NCBI C++ Toolkit Book PDF Format Disclaimer:

Please note that this PDF does not offer internal hyperlinks from the table of contents to the individual chapters, nor is any other navigation feature available in this PDF. Please consult the online version of the book at http://www.ncbi.nlm.nih.gov/toolkit/doc/book/ if you require these navigation features.

Revised: 23-September-2013

The NCBI C++ Toolkit

The NCBI C++ Toolkit

Book Information

Contributing Authors

For list of contributors, see Table 1.

Table 1

List of Contributors

Full-time developers NOTE: This table is always a draft and virtually never up-to-date. Last updated: 19 Sep 2013	
Special thanks to Jim Ostell	**Established the biological and bibliographic data model supported by the C++ Toolkit. He also established the overall design, priorities, and goals for C++ Toolkit based on experience building and using the NCBI C Toolkit which preceeded it. He continues to cheer on the list of talented software developers and scientists below who are primarily responsible for making the C++ Toolkit a reality and for introducing most of its nicer features.**
Denis Vakatov (since Oct 1998)	Fathered the Toolkit. Coordinate all works on the portable (core, non-internal) projects of the Toolkit, actively participate in the design (and sometimes implementation details) of all core APIs. CONFIGURE -- orig.author and eventual supporter. CORELIB -- orig.author of many modules and active supporter/developer. CGI -- orig.author of "CGI Request" and "CGI Application". DBAPI -- a massive code and API revision on incorporating DBAPI into the Toolkit (with the orig.author, V.Soussov); participate in the core (exception, diagnostic, driver manager) code support and development. CONNECT -- orig.author of the core, abstract connection(CONN) and portable socket(SOCK) APIs, and FW-daemon. GUI -- helped setup the project structure, namespace and common definitions. DOC -- "Reference Manual", "FAQ", mailing lists; snapshots, announcements.
Eugene Vasilchenko (Nov 1999 - Feb 2001) (Aug 2002 - current)	CORELIB -- "CObject, CRef<>", multi-threading CGI -- orig.author of "CGI Response", "Fast-CGI module" HTML -- orig.author SERIAL -- orig.author DATATOOL -- orig.author OBJMGR -- taking over the client-side "loader" code; revising some "user" APIs
Anton Lavrentiev (since Mar 2000)	CONNECT -- *[principal developer]* author of "NCBI Services": network client API, load balancer, service mapper, dispatcher and launcher; daemons' installation, configuration and monitoring. CTOOLS -- *[principal developer]* connectivity with the NCBI C Toolkit. MSVC++ project mutli-configuration *[principal developer]*. Help with the internal wxWindows installations on Windows and Solaris. DOC -- documentation on all of the above Tune-up of online docs and source browsers.
Aleksey Grichenko (since Jan 2001)	CORELIB -- orig.author of the thread library SERIAL -- support and further development DATATOOL -- support and further development OBJMGR -- *[principal developer]* client-side API and implementation Incorporation of MT-safety and "safe-static" features to all of the above
Aaron Ucko (since Aug 2001)	ID1_FETCH -- *[principal developer]* developed from a test/demo application to a real client. CONFIGURE -- *[principal developer]*; active support and development of the Unix building framework CORELIB -- generalized error handlers, implemented E-mail and CGI/HTML ones UTIL,CONNECT -- blocking-queue; multi-threaded network server API OBJECTS -- adding new functionality, QA'ing other people's additions ALNMGR -- participated in the design PubMed (internal) -- *[principal developer]* developing C++ bio-sequence indexer framework Toolkit builds on Unix'es (internal) -- support of the building and installation framework
Andrei Gourianov (since Nov 2001)	CORELIB -- major revamp of the exception API -- structure, standartize. OBJMGR -- client-side API, implementation, and docs. DATATOOL -- adding DTD/XML support for the code generator
Vladimir Ivanov (since Apr 2001)	HTML -- further support and development CORELIB, UTIL -- porting of some very platform-dependent extensions Tune-up of online docs and source browsers. Internal wxWindows installations on Windows and Solaris.

Full-time developers NOTE: This table is always a draft and virtually never up-to-date. Last updated: 19 Sep 2013	
David McElhany (since Jan 2009)	DOC -- Toolkit book
Victoria Serova (since Dec 2005)	DOC -- Toolkit book
Diane Zimmerman (2000 only)	DOC -- "Programming Manual"
Chris Lanczycki (summer 2002 only)	DOC -- major reorganization of the docs structure and appearance
Major contributors	
Anton Butanaev	OBJMGR -- helped to implement ID1 loader DBAPI (in progress) -- driver for MySQL
Cliff Clausen	OBJECTS -- ported various bio-sequence related code and utilities (from C Toolkit)
Mike DiCuccio	GBENCH -- (in progress) extendable C++ FLTK/OpenGL based GUI tool for the retrieval, visualization, analysis, editing, and submitting of biological sequences
Jonathan Kans	OBJECTS -- helped port seq. validator (from C Toolkit). Provide MAC platform support. Contributed code (which sometimes other people ported) for fast sequence alphabet conversion and for translation of coding regions. Also writing the 5-column feature table reader.
Michael Kholodov	DBAPI -- author of the "user-level" database API based on Vladimir Soussov's portable "driver-level" API. SERIAL, DATATOOL -- provided eventual support of (in the beginning of 2001)
Michael Kimelman	OBJMGR (in progress) -- server-side API and implementation, client-side loader (both generic and its implementation for ID)
Vladimir Lebedev	GUI_SEQ -- the first FLTK/OpenGL based GUI widgets for bio-seq visualization Provide MAC platform support.
Peter Meric	GBENCH (in progress) -- extendable C++ FLTK/OpenGL based GUI tool for the retrieval, visualization, analysis, editing, and submitting of biological sequences and maps (eg. MapViewer data)
Vsevolod Sandomirskiy	CORELIB, CGI -- draft-authored some application- and context- classes
Victor Sapojnikov	DBAPI -- participated in the implementation of the Microsoft DBLIB driver on Windows; (in progress) multiplatform "network bridge" driver
Vladimir Soussov	DBAPI -- *[principal developer]* author of the portable DB driver API and its implemementations for CTLIB(Sybase for Unix and Windows), DBLIB (Sybase and Microsoft), FreeTDS and ODBC
Kamen Todorov	ALNMGR -- library to deal with bio-sequence alignments
Paul Thiessen	APP/CN3D -- Cn3D: graphical protein and alignment viewing, editing, and annotation. ALGO/STRUCTURE/STRUCT_DP -- Block-based dynamic programming sequence alignments. OBJTOOLS/CDDALIGNVIEW -- HTML sequence alignment displays.
Charlie (Chunlei) Liu, Chris Lanczycki	ALGO/STRUCTURE/CD_UTILS -- These contain numerous algorithms used by the structure group and the CDD project.
Thomas Madden, Christiam Camacho, George Coulouris, Ning Ma, Vahram Avagyan, Jian Ye	BLAST -- Basic Local Alignment Search Tool
Greg Boratyn, Richa Agarwala	COBALT -- Constraint Based Alignment Tool

Full-time developers NOTE: This table is always a draft and virtually never up-to-date. Last updated: 19 Sep 2013	
Jonathan Kans	5-column feature table reader; Defline generator function; GenBank flatfile generator; Basic and Extended sequence cleanup; Sequence record validator; Alignment readers; Various format readers (e.g., BED, WIGGLE)

License

DISCLAIMER: This (book-located) copy of the license may be out-of-date - please see the up-to-date version at: http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/doc/public/LICENSE

```
License: GNU LGPL
Location: {src,include}/gui/widgets/FLU
Authors: Jason Bryan
License: GNU LGPL
Location: {src,include}/gui/widgets/Fl_Table
Authors: Greg Ercolano
License: GNU LGPL
Location: include/util/bitset
Author: Anatoliy Kuznetsov
License: MIT [include/util/bitset/license.txt]
Location: {src,include}/util/compress/bzip2
Author: Julian R Seward
License: BSDish [src/util/compress/bzip2/LICENSE]
Location: {src,include}/util/compress/zlib
Authors: Jean-loup Gailly and Mark Adler
License: BSDish [include/util/compress/zlib/zlib.h]
Location: {src,include}/util/regexp
Author: Philip Hazel
License: BSDish [src/util/regexp/doc/LICENCE]
Location: {src,include}/misc/xmlwrapp
Author: Peter J Jones at al. [src/misc/xmlwrapp/AUTHORS]
License: BSDish [src/misc/xmlwrapp/LICENSE]
==============================================================
Copyright F.A.Q.
--------------------------------------------------------------
Q. Our product makes use of the NCBI source code, and we did changes
and additions to that version of the NCBI code to better fit it to
our needs. Can we copyright the code, and how?
A. You can copyright only the *changes* or the *additions* you made to the
NCBI source code. You should identify unambiguously those sections of
the code that were modified, e.g. by commenting any changes you made
in the code you distribute. Therefore, your license has to make clear
to users that your product is a combination of code that is public domain
within the U.S. (but may be subject to copyright by the U.S. in foreign
countries) and code that has been created or modified by you.
--------------------------------------------------------------
Q. Can we (re)license all or part of the NCBI source code?
A. No, you cannot license or relicense the source code written by NCBI
since you cannot claim any copyright in the software that was developed
at NCBI as a 'government work' and consequently is in the public domain
within the U.S.
--------------------------------------------------------------
Q. What if these copyright guidelines are not clear enough or are not
applicable to my particular case?
A. Contact us. Send your questions to 'toolbox@ncbi.nlm.nih.gov'.
```

Part 1: Overview

Part 1 provides an introduction to the C++ Toolkit. The first chapter, "Introduction to the C++ Toolkit" provides a broad overview of the capabilties in the C++ Toolkit with links to other chapters that cover topics in more detail. The second chapter "Getting Started" provides a description of how to obtain the C++ Toolkit, the layout of the source distribution tree, and how to get started. The following is a list of chapters in this part:

1: Introduction to the C++ Toolkit

Last Update: February 25, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

One difficulty in understanding a major piece of software such as the C++ Toolkit is knowing where to begin in understanding the overall framework or getting the 'big picture' of how all the different components relate to each other. One approach is to dive into the details of one component and understand it in sufficient detail to get a roadmap of the rest of the components, and then repeat this process with the other components. Without a formal road map, this approach can be very time consuming and it may take a long time to locate the functionality one needs.

When trying to understand a major piece of software, it would be more effective if there is a written text that explains the overall framework without getting too lost in the details. This chapter is written with the intent to provide you with this broader picture of the C++ Toolkit.

This chapter provides an introduction to the major components that make up the toolkit. You can use this chapter as a roadmap for the rest of the chapters that follow.

Chapter Outline

The following is an outline of the topics presented in this chapter:

The CORELIB Module

The C++ Toolkit can be seen as consisting of several major pieces of code that we will refer to as *module*. The core module is called, appropriately enough, CORELIB, and provides a portable way to write C++ code and many useful facilities such as an application framework, argument processing, template utilities, threads, etc. The CORELIB facilities are used by other major modules. The rest of the sections that follow discusses the CORELIB and the other C++ Toolkit modules in more detail.

The following is a list of the CORELIB facilities. Note that each facility may be implemented by a number of C++ classes spread across many files.

- Application Framework
- Argument processing
- Diagnostics
- Environment Interface
- Files and Directories
- MT Test wrappers
- Object and Ref classes
- Portability definitions
- Portable Exception Handling
- Portable Process Pipes
- Registry
- STL Use Hints
- String Manipulations
- Template Utilities
- Threads
- Time

A brief description of each of each of these facilities are presented in the subsections that follow:

Application Framework

The Application framework primarily consists of an abstract class called CNcbiApplication which defines the high level behavior of an application. For example, every application upon loading seems to go through a cycle of doing some initialization, then some processing, and upon completion of processing, doing some clean up activity before exiting. These three phases are modeled in the CNcbiApplication class as interface methods Init(), Run(), and Exit().

A new application is written by deriving a class from the CNcbiApplication base class and writing an implementation of the Init(), Run(), and Exit() methods. Execution control to the new application is passed by calling the application object's AppMain() method inherited from the CNcbiApplication base class (see Figure 1). The AppMain() method is similar to the main() method used in C/C++ programs and calls the Init(), Run(), and Exit() methods.

More details on using the CNcbiApplication class are presented in a later chapter.

Argument processing

In a C++ program, control is transferred from the command line to the program via the main() function. The main() function is passed a count of the number of arguments (int argc), and an array of character strings containing arguments to the program (char** argv). As long as the argument types are simple, one can simply set up a loop to iterate through the array of argument values and process them. However, with time applications evolve and grow more complex. Often there is a need to do some more complex argument checking. For example, the application may want to enforce a check on the number and position of arguments, check the argument type (int, string, etc.), check for constraints on argument values, check for flags, check for arguments that follow a keyword (-logfile mylogfile.log), check for mandatory arguments, display usage help on the arguments, etc.

To make the above tasks easier, the CORELIB provides a number of portable classes that encapsulate the functionality of argument checking and processing. The main classes that provide this functionality are the CArgDescriptions, CArgs, CArgValue classes.

Argument descriptions such as the expected number, type, position, mandatory and optional attributes are setup during an application's initilization such as the application object's Init() method (see previous section) by calling the CArgDescriptions class methods. Then, the arguments are extracted by calling the CArgs class methods.

More details on argument processing are presented in a later chapter.

Diagnostics

It is very useful for an application to post messages about its internal state or other diagnostic information to a file, console or for that matter any output stream. The CORELIB provides a portable diagnostics facility that enables an application to post diagnostic messages of various severity levels to an output stream. This diagnostic facility is provided by the CNcbiDiag class. You can set the diagnostic stream to the standard error output stream (NcbiErr) or to any other output stream.

You can set the severity level of the message to Information, Warning, Error, Critical, Fatal, or Trace. You can alter the severity level at any time during the use of the diagnostic stream.

More details on diagnostic streams and processing of diagnostic messages are presented in later chapters.

Environment Interface

An application can read the environment variable settings (such as PATH) that are in affect when the application is run. CORELIB defines a portable CNcbiEnvironment class that stores the environment variable settings and provides applications with methods to get the environment variable values.

More details on the environment interface are presented in a later chapter.

Files and Directories

An application may need access to information about a file or directory. The CORELIB provides a number of portable classes to model a system file and directory. Some of the important classes are CFile for modeling a file, CDir for modeling a directory, and CMemoryFile for memory mapped file.

For example, if you create a CFile object corresponding to a system file, you can get the file's attribute settings such as file size, permission settings, or check the existence of a file. You can get the directory where the file is located, the base name of the file, and the file's extension. There are also a number of useful functions that are made available through these classes to parse a file path or build a file path from the component parts such as a directory, base name, and extension.

More details on file and directory classes are presented in later chapters.

MT Test wrappers

The CNcbiApplication class which was discussed earlier provides a framework for writing portable applications. For writing portable multi-threaded applications, the CORELIB provides a CThreadedApp class derived from CNcbiApplication class which provides a framework for building multi-threaded applications.

Instead of using the Init(), Run(), Exit() methods for the CNcbiApplication class, the CThreadedApp class defines specialized methods such as Thread_Init(), Thread_Run(), Thread_Exit(), Thread_Destroy() for controlling thread behavior. These methods operate on a specific thread identified by a thread index parameter.

Object and Ref classes

A major cause of errors in C/C++ programs is due to dynamic allocation of memory. Stated simply, memory for objects allocated using the new operator must be released by a corresponding delete operator. Failure to delete allocated memory results in memory leaks. There may also be programming errors caused by references to objects that have never been allocated or improperly allocated. One reason these types of problems crop up are because a programmer may dynamically allocate memory as needed, but may not deallocate it due to unanticipated execution paths.

The C++ standard provides the use of a template class, auto_ptr , that wraps memory management inside constructors and destructors. Because a destructor is called for every constructed object, memory allocation and deallocation can be kept symmetrical with respect to each other. However, the auto_ptr does not properly handle the issue of ownership when multiple auto pointers, point to the same object. What is needed is a reference counted smart pointer that keeps a count of the number of pointers pointing to the same object. An object can only be released when its reference count drops to zero.

The CORELIB implements a portable reference counted smart pointer through the CRef and CObject classes. The CRef class provides the interface methods to access the pointer and the CObject is used to store the object and the reference count.

More CObject classes are presented in a later chapter.

Portability definitions

To help with portability, the CORELIB uses only those C/C++ standard types that have some guarantees about size and representation. In particular, use of long, long long, float is not recommended for portable code.

To help with portability, integer types such as Int1, Uint1, Int2, Uint2, Int4, Uint4, Int8, Uint8 have been defined with constant limits. For example, a signed integer of two bytes size is defined as type Int2 with a minimum size of kMin_I2 and a maximum size of kMax_I2. There are minimum and maximum limit constants defined for each of the different integer types.

More details on standard portable data types are presented in a later chapter.

Portable Exception Handling

C++ defines a structured exception handling mechanism to catch and process errors in a block of code. When the error occurs an exception is thrown and caught by an exception handler. The exception handler can then try to recover from the error, or process the error. In the C++ standard, there is only one exception class (std::exception), that stores a text message that can be printed out. The information reported by the std::exception may not be enough for a complex system. The CORELIB defines a portable CException class derived from std::exception class that remedies the short comings of the standard exception class

The CORELIB defines a portable CException class derived from std::exception class. The CException class in turn serves as a base class for many other exception classes specific to an application area such as the CCoreException, CAppException, CArgException, CFileException, and so on. Each of these derived classes can add facilities specific to the application area they deal with.

These exception classes provides many useful facilities such as a unique identification for every exception that is thrown, the location (file name and line number) where the exception occurred, references to lower-level exceptions that have already been thrown so that a more complete picture of the chain of exceptions is available, ability to report the exception data into an arbitrary output channel such as a diagnostic stream, and format the message differently for each channel.

More details on exceptions and exception handling are presented in a later chapter.

Portable Process Pipes

A pipe is a common mechanism used to establish communications between two separate processes. The pipe serves as a communication channel between processes.

The CORELIB defines the CPipe class that provides a portable inter-process communications facility between a parent process and its child process. The pipe is created by specifying the command and arguments used to start the child process and specifying the type of data channels (text or binary) that will connect the processes. Data is sent across the pipe using the CPipe read and write methods.

Registry

N.B. The preferred way to define configuration parameters for an application is to use the macros in the CParam class (e.g. NCBI_PARAM_DECL). More details on the CParam class and its macros are presented in a later chapter. If the CParam class cannot be used, then the registry may be used instead.

The settings for an application may be read from a configuration or initialization file (the "registry"). This configuration file may define the parameters needed by the application. For example, many Unix programs read their parameter settings from configuration files. Similarly, Windows programs may read and store information in an internal registry database, or an initialization file.

The CNcbiRegistry class provides a portable facility to access, modify and store runtime information read from a configuration file. The configuration file consists of sections. A section is defined by a section header of the form [***section-header-name***]. Within each section, the parameters are defined using (name, value) pairs and represented as ***name=value*** strings. The syntax closely resembles the '.ini' files used in Windows and also by Unix tools such as Samba.

More details on the Registry are presented in a later chapter.

STL Use Hints

To minimize naming conflicts, all NCBI code is placed in the ncbi name space. The CORELIB defines a number of portable macros to help manage name space definitions. For example, you can use the BEGIN_NAME_SPACE macro at the start of a section of code to place that code in the specified name space. The END_NAME_SPACE macros is used to indicate the end the of the name space definition. To declare the use of the NCBI namespace, the macros USING_NCBI_SCOPE is used.

A number of macros have been defined to handle non-standard behavior of C++ compilers. For example, a macro BREAK is defined, that is used to break out of a loop, instead of using the break statement directly. This is done to handle a bug in the Sun WorkShop (pre 5.3 version) compiler that fails to call destructors for objects created in for-loop initializers. Another example is that some compilers (example, Sun Pro 4.2) do not understand the using namespace std; statement. Therefore, for portable code, the using namespace statement should be prohibited.

More details on the use of portable macros are presented in a later chapter.

String Manipulations

C++ defines the standard string class that provides operations on strings. However, compilers may exhibit non-portable string behavior especially with regards to multi-threaded programs. The CORELIB provides portable string manipulation facilities through the NStr class that provides a number of class-wide functions for string manipulation.

NStr portable functions include the string-to-X and X-to-string conversion functions where X is a data type including a pointer type, string comparisons with and without case, pattern searches within a string, string truncation, substring replacements, string splitting and join operations, string tokenization, etc.

Template Utilities

The C++ Template classes support a number of useful template classes for data structures such as vectors, lists, sets, maps, and so on.

The CORELIB defines a number of useful utility template classes. Some examples are template classes and functions for checking for equality of objects through a pointer, checking for non-null values of pointers, getting and setting map elements, deleting all elements from a container of pointers where the container can be a list, vector, set, multiset, map or multimap.

More details on the template utilities are presented in a later chapter.

Threads

Applications can run faster, if they are structured to exploit any inherent parallelism in the application's code execution paths. Code execution paths in an application can be assigned to separate threads. When the application is run on a multiprocessor system, there can be significant improvements in performance especially when threads run in parallel on separate processors.

The CORELIB defines a portable CThread class that can be used to provide basic thread functionality such as thread creation, thread execution, thread termination, and thread cleanup.

To create user defined threads you need to derive your class from CThread, and override the thread's Main() method and, and if necessary the OnExit() method for thread-specific cleanup. Next, you create a thread object by instantiating the class you derived from CThread. Now you are ready to launch thread execution by calling the thread's Run() method. The Run() method starts thread execution and the thread will continue to run until it terminates. If you want the thread to run independently of the parent thread you call the thread's Detach() method. If you want to wait till the thread terminates, you call the thread's Join() method.

Synchronization between threads is provided through mutexes and read/write locks.

More details on threads and synchronization are presented in a later chapter.

Time

The CTime class provides a portable interface to date and time functions. CTime can operate with both local and UTC time, and can be used to store data and time at a particular moment or elapsed time. The time epoch is defined as Jan 1, 1900 so you cannot use CTime for storing timestamps before Jan 1, 1900.

The CTime class can adjust for daylight savings time. For display purposes, the time format can be set to a variety of time formats specified by a format string. For example, "M/D/Y h:m:s" for a timestamp of "5/6/03 14:07:09". Additional time format specifiers are defined for full month name (B), abbreviated month name (b), nanosecond (S), timezone format (Z), full weekday name (W) and abbreviated weekday name (w).

A class CStopWatch is also available that acts as a stop watch and measures elapsed time via the Elapsed() method, after its Start() method is called.

More details on the CTime class are presented in a later chapter.

The ALGORITHM Module

The ALGORITHM module is a collection of rigorously defined, often computationally intensive algorithms performed on sequences. It is divided into three groups:

- ALIGN. A set of global alignment algorithms, including generic Needleman-Wunsch, a linear-space Hirschberg's algorithm and a spliced (cDna/mRna-to-Genomic) alignment algorithm.
- BLAST. Basic Local Alignment Tool code and interface.
- SEQUENCE. Various algorithms on biological sequences, including antigenic determinant prediction, CPG-island finder, ORF finder, string matcher and others.

The CGI Module

The CGI module provides an integrated framework for writing CGI applications. It consists of classes that implement the CGI (Common Gateway Interface). These classes are used to retrieve and parse an HTTP request, and then compose and deliver an HTTP response.

The CGI module consists of a number of classes. The interaction between these classes is fairly complex, and therefore, not covered in this introductory chapter. We will attempt to only identify the major classes in this overview, and cover the details of their interaction in later chapters. Among the more important of the CGI classes are the CCgiApplication, CCgiContext, CCgiRequest, CCgiResponse, and CCgiCookie.

The CCgiApplication is used to define the CGI application and is derived from the CNcbiApplication discussed eariler. You write a CGI application by deriving application class from CCgiApplication and providing an adoption of the Init(), Run(), and Exit() methods inherited from the CNcbiApplication class. Details on how to implement the Init(), Run() and Exit() methods for a CGI application are provided in a later chapter.

The CCgiRequest class is defined to receive and parse the request, and the CCgiResponse class outputs the response to an output stream.

The CCgiCookie class models a *cookie*. A cookie is a name, value string pair that can be stored on the user's web browser in an attempt to remember a session state. All incoming CCgiCookies are parsed and stored by the CCgiRequest object, and the outgoing cookies are sent along with the response by the CCgiResponse object.

The CGI application executes in a 'context' defined by the CCgiContext class. The CCgiContext class provides a wrapper for the CCgiApplication, CCgiRequest and CCgiResponse objects and drives the processing of input requests.

More details on CGI classes and their interactions are presented in a later chapter.

The CONNECT Module

The CONNECT module implements a variety of interfaces and classes dealing with making connections to a network services. The core of the Connection Library is written in C which provides a low level interface to the communication protocols. The CONNECT module provides C++ interfaces so that the objects have diagnostic and error handling capabilities that are consistent with the rest of the toolkit. The standard sockets (SOCK) API is implemented on a variety of platforms such as Unix, MS-Windows, MacOS, Darwin. The CONNECT module provides a higher level access to the SOCK API by using C++ wrapper classes.

The following is a list of topics presented in this section:

- Socket classes
- Connector and Connection Handles
- Connection Streams
- Sendmail API
- Threaded Server

Socket classes

The C++ classes that implement the socket interface are CSocket, CDatagramSocket, CListeningSocket, and CSocketAPI. The socket defines an end point for a connection which consists of an IP address (or host name) of the end point, port number and transport protocol used (TCP, UDP).

The CSocket class encapsulates the descriptions of both local and remote end points. The local end point, which is the end point on the client issuing a connection request, is defined as parameters to the CSocket constructor. The remote end point on which the network service is running is specified as parameters to the Connect() method for the CSocket class. The CSocket class defines additional methods to manage the connection such as Reconnect() to reconnect to the same end point as the Connect() method; the Shutdown() method to terminate the connection; the Wait() method to wait on several sockets at once; the Read() and Write() methods to read and write data via the socket; and a number of other support methods.

The CSocket is designed for connection-oriented services such as those running over the TCP transport protocol. For connectionless, or datagram services, such as those running over the UDP transport protocol, you must use the CDatagramSocket class. The local end point is defined as parameters to the CDatagramSocket constructor. The remote end point is specified as parameters to the Connect() method for the CDatagramSocket class. Unlike the case of the connection-oriented services, this Connect() method only specifies the default destination address, and does not restrict the source address of the incoming messages. The methods Send () and Recv() are used to send the datagram, and the method SetBroadcast() sets the socket to broadcast messages sent to the datagram socket. The CDatagramSocket is derived from the

CSocket class but methods such as Shutdown() and Reconnect() that apply to connection-oriented services are not available to users of the CDatagramSocket class.

The CListeningSocket is used by server-side applications to listen for connection requests. The CListeningSocket constructor specifies the port to listen to and the size of the connection request queue. You can change the port that the server application listens to any time by using the Listen() method. The Accept() method accepts the connection request, and returns a CSocket object through which data is transferred.

The CSocketAPI is a C++ wrapper for class-wide common socket utility functions available for sockets such as the gethostname(), gethostbyaddr(), ntoa(), aton(), and so on.

Connector and Connection Handles

The SOCK interface is a relatively low-level interface for connection services. The CONNECT module provides a generalization of this interface to connection services using a connection type and specialized connectors.

A connection is modeled by a connection type and a connector type. The connector type models the end point of the connection, and the connection type, the actual connection. Together, the connector and connection objects are used to define the following types of connections: socket, file, http, memory, and a general service connection.

The connector is described by a connector handle, CONNECTOR. CONNECTOR is a typedef and defined as a pointer to an internal data structure.

The connection is described by a connection handle CONN. CONN is a typedef and defined as a pointer to an internal structure. The CONN type is used as a parameter to a number of functions that handle the connection such as CONN_Create(), CONN_ReInit(), CONN_Read (), CONN_Write(), etc.

The CONNECTOR socket handle is created by a call to the SOCK_CreateConnector() function and passed the host name to connect to, the port number on the host to connect to, and maximum number of retries. The CONNECTOR handle is then passed as an argument to the CONN_Create() which returns a CONNECTION handle. The CONNECTION handle is then used with the connection functions (that have the prefix CONN_) to process the connection. The connection so created is bi-directional (full duplex) and input and output data can be processed simultaneously.

The other connector types, file, http, memory are similar to the socket connector type. In the case of a file connector, the connector handle is created by calling the FILE_CreateConnector () function and passed an input file and an output file. This connector could be used for both reading and writing files, when input comes from one file, and output goes to another file. This differs from normal file I/O when a single handle is used to access only one file, but resembles data exchange via sockets, instead. In the case of the HTTP connection, the HTTP_CreateConnector type is called and passed a pointer to network information structure, a pointer to a user-header consisting of HTTP tag-values, and a bitmask representing flags that affect the HTTP response.

The general service connector is the most complex connector in the library, and can model any type of service. It can be used for data transfer between an application and a named service. The data can be sent via HTTP or directly as a byte stream (using SOCK directly). In the former case it uses the HTTP connectors and in the latter the SOCK connectors. The general service connector is used when the other connector types are not adequate for implementing the task on hand.

More details on connector classes are presented in a later chapter.

Connection Streams

The CONNECT module provides a higher level of abstraction to connection programming in the form of C++ connection stream classes derived from the standard iostream class. This makes the familiar stream I/O operators, manipulators available to the connection stream. The main connection stream classes are the CConn_IOStream, CCon_SocketStream, CCon_HttpStream, CCon_ServiceStream, and CCon_MemoryStream.

Figure 2 shows the relationship between the different stream classes. From this figure we can see that CConn_IOStream is derived from the C++ iostream class and serves as a base class for all the other connection stream classes. The CCon_IOStream allows input operations to be tied to the output operations so that any input attempt first flushes the output queue from the internal buffers.

The CCon_SocketStream stream models a stream of bytes in a bi-directional TCP connection between two end points specified by a host/port pair. As the name suggests the socket stream uses the socket interface directly. The CCon_HttpStream stream models a stream of data between and HTTP client and an HTTP server (such as a web server). The server end of the stream is identified by a URL of the form http://host[:port]/path[?args]. The CCon_ServiceStream stream models data transfer with a named service that can be found via dispatcher/load-balancing daemon and implemented as either HTTP CGI, standalone server, or NCBI service. The CCon_MemoryStream stream models data transfer in memory similar to the C++ strstream class.

More details on connection stream classes are presented in a later chapter.

Sendmail API

The CONNECT module provides an API that provides access to SMTP protocol. SMTP (Simple Mail Transfer Protocol) is a standard email relaying protocol used by many popular MTAs (Message Transfer Agents), such as sendmail, smail, etc, found on many systems. SMTP passes (relays) email messages between hosts in the Internet all the way from sender to recipient.

To initiate the use of the sendmail API, you must call the SendMailInfo_Int() function that initializes structure SSendMailInfo, passed by a pointer. Your code then modifies the structure to contain proper information such as that expected in a mail header (To, From, CC, BCC fields) and other communication settings from their default values set at initialization. Then, you can send email using the CORE_SendMail() or CORE_SendMailEx() functions.

Threaded Server

The CONNECT module provides support for multithreaded servers through the CThreadedServer class. The CThreadedServer class is an abstract class for network servers and uses thread pools. This class maintains a pool of threads, called worker threads, to process incoming connections. Each connection gets assigned to one of the worker threads, allowing the server to handle multiple requests in parallel while still checking for new requests.

You must derive your threaded server from the CThreadedServer class and define the Process () method to indicate what to do with each incoming connection. The Process() method runs asynchronously by using a separate thread for each request.

More details on threaded server classes are presented in a later chapter.

The CTOOL Module

The CTOOL module provides bridge mechanisms and conversion functions. More specifically, the CTOOL module provides a number of useful functions such as a bridge between the NCBI C++ Toolkit and the older C Toolkit for error handling, an ASN.1 connections stream that builds on top of the connection stream, and an ASN converter that provides templates for converting ASN.1-based objects between NCBI's C and C++ in-memory layouts.

The ASN.1 connections support is provides through functions CreateAsnConn() for creating an ASN stream connection; CreateAsnConn_ServiceEx() for creating a service connection using the service name, type and connection parameters; and CreateAsnConn_Service() which is a specialized case of CreateAsnConn_ServiceEx() with some parameters set to zero.

The DBAPI Module

The DBAPI module supports object oriented access to databases by providing user classes that model a database as a data source to which a connection can be made, and on which ordinary SQL queries or stored procedure SQL queries can be issued. The results obtained can be navigated using a result class or using the 'cursor' mechanism that is common to many databases.

The user classes are used by a programmer to access the database. The user classes depend upon a database driver to allow low level access to the underlying relational database management system (RDBMS). Each type of RDBMS can be expected to have a different driver that provides this low level hook into the database. The database drivers are architected to provide a uniform interface to the user classes so that the database driver can be changed to connect to a different database without affecting the program code that makes use of the user classes. For a list of the database drivers for different database that are supported, see the Supported DBAPI Drivers section.

The following is a list of topics presented in this section:

- Database User Classes
- Database Driver Architecture

Database User Classes

The interface to the database is provided by a number of C++ classes such as the IDataSource, IDbConnection, IStatement, ICallableStatement, ICursor, IResultSet, IResultSetMetaData . The user does not use these interfaces directly. Instead, the DBAPI module provides concrete classes that implement these interface classes. The corresponding concrete classes for the above mentioned interfaces are CDataSource, CDbConnection, CStatement, CCallableStatement, CCursor, CResultSet, CResultSetMetaData.

Before accessing to a specific database, the user must register the driver with the CDriverManager class which maintains the drivers registered for the application. The user does this by using the CDriverManager class' factory method GetInstance() to create an instance of the CDriverManager class and registering the driver with this driver manager object. For details on how this can be done, see the Choosing the Driver section.

After the driver has been registered, the user classes can be used to access that database. There are a number of ways this can be done, but the most common method is to call the IDataSource factory method CreateDs() to create an instance of the data source. Next, the CreateConnection () method for the data source is called, to return a connection object that implements the IConnection interface. Next, the connection object's Connect() method is called with the user name, password, server name, database name to make the connection to the database. Next, the connection object's CreateStatement() method is called to create a statement object that implements the IStatement interface. Next, the statement object's Execute() method is called to execute the query. Note that additional calls to the IConnection::CreateStatement() results in cloning the connection for each statement which means that these connections inherit the database which was specified in the Connect() or SetDatabase() method.

Executing the statement objects' Execute() method returns the result set which is stored in the statement object and can be accessed using the statement object's GetResultSet() method. You can then call the statement object's HasRows() method which returns a Boolean true if there are rows to be processed. The type of the result can be obtained by calling the IResultSet::GetResultType() method. The IStatement::ExecuteUpdate() method is used for SQL statements that do not return rows (UPDATE or DELETE SQL statement), in which case the method IStatement::GetRowCount() returns the number of updated or deleted rows.

Calling the IStatement::GetResultSet() returns the rows via the result set object that implements the IResultSet interface. The method IResultSet::Next() is used to fetch each row in the result set and returns a false when no more fetch data is available; otherwise, it returns a true. All column data, except BLOB data is represented by a CVariant object. The method IResultSet::GetVariant() takes the column number as its parameter where the first column has the start value of 1.

The CVariant class is used to describe the fields of a record which can be of any data type. The CVariant has a set of accessory methods (GetXXX()) to extract a value of a particular type. For example, the GetInt4(), GetByte(), GetString(), methods will extract an Int4, Byte data value from the CVariant object. If data extraction is not possible because of incompatible types, the CVariantException is thrown. The CVariant has a set of factory methods for creating objects of a particular data type, such as CVariant::BigInt() for Int8, CVariant::SmallDateTime() for NCBI's CTime, and so on.

For sample code illustrating the above mentioned steps, see the Data Source and Connections and Main Loop sections.

Database Driver Architecture

The driver can use two different methods to access the particular RDBMS. If RDBMS provides a client library (CTLib) for a given computer system, then the driver utilizes this library. If there is no client library, then the driver connects to RDBMS through a special gateway server which is running on a computer system where such library does exist.

The database driver architecture has two major groups of the driver's objects: the RDBMS independent objects, and the RDBMS dependent objects specific to a RDBMS. From a user's perspective, the most important RDBMS dependent object is the driver context object. A connection to the database is made by calling the driver context's Connect() method. All driver contexts implement the same interface defined in the I_DriverContext class.

If the application needs to connect to RDBMS libraries from different vendors, there is a problem trying to link statically with the RDBMS libraries from different vendors. The reason for this is that most of these libraries are written in C, and may use the same names which cause name collisions. Therefore, the C_DriverMgr is used to overcome this problem and allow the creation of a mixture of statically linked and dynamically loaded drivers and use them together in one executable.

The low level connection to an RDBMS is specific to that RDBMS. To provide RDBMS independence, the connection information is wrapped in an RDBMS independent object CDB_Connection. The commands and the results are also wrapped in an RDBMS independent object. The user is responsible for deleting these RDBMS independent objects because the life spans of the RDBMS dependent and RDBMS independent objects are not necessarily the same.

Once you have the CDB_Connection object, you can use it as a factory for the different types of command objects. The command object's Result() method can be called to get the results. To send and to receive the data through the driver you must use the driver provided datatypes such as CDB_BigInt, CDB_Float, CDB_SmallDateTime. These driver data types are all derived from CDB_Object class.

More details on the database driver architecture is presented in a later chapter.

The GUI Module

The C++ Toolkit does not include its own GUI Module. Instead, Toolkit-based GUI applications make use of third party GUI packages. Depending on the requirements, we recommend either wxWidgets or FOX.

More details on developing GUI application in conjunction with the C++ Toolkit are presented in a later chapter.

The HTML Module

The HTML module implements a number of HTML classes that are intended for use in CGI and other programs. The HTML classes can be used to generate HTML code dynamically.

The HTML classes can be used to represent HTML page internally in memory as a graph. Each HTML element or tag is represented by a node in the graph. The attributes for an HTML element are represented as attributes in the node. A node in the graph can have other elements as children. For example, for an HTML page, the top HTML element will be described by an HTML node in the graph. The HTML node will have the HEAD and BODY nodes as its children. The BODY node will have text data and other HTML nodes as its children. The graph structure representation of an HTML page allows easy additions, deletions and modification of the page elements.

Note that while the HTML classes can be used to represent the HTML page internally in memory as a graph there is no provision for parsing of existing HTML pages to generate these classes.

The following is a list of topics presented in this section:

- Relationships between HTML classes
- HTML Processing

Relationships between HTML classes

The base class for all nodes in the graph structure for an HTML document is the CNCBINode. The CNCBINode class is derived from CObject and provides the ability to add, delete, and modify the nodes in the graph. The ability to add and modify nodes is inherited by all the classes derived from CNCBINode (see Figure 3). The classes derived from CNCBINode represent the HTML elements on an HTML page. You can easily identify the HTML element that a class handles from the class names such as CHTMLText, CHTMLButtonList, etc.

The text node classes CHTMLText and CHTMLPlainText are intended to be used directly by the user. Both CHTMLText and CHTMLPlainText are used to insert text into the generated html, with the difference that CHTMLPlainText class performs HTML encoding before generation. A number of other classes such as CHTMLNode, CHTMLElement, CHTMLOpenElement, and CHTMLListElement are base classes for the elements actually used to construct an HTML page, such as CHTML_head, CHTML_form (see Figure 4).

The CHTMLNode class is the base class for CHTMLElement and CHTMLOpenElement and is used for describing the HTML elements that are found in an HTML page such as HEAD, BODY, H1, BR, etc. The CHTMLElement tag describes those tags that have a close tag and are well formed. The CHTMLOpenElement class describes tags that are often found without the corresponding close tag such as the BR element that inserts a line break. The CHTMLListElement class is used in lists such as the OL element.

Important classes of HTML elements used in forms to input data are the input elements such as checkboxes, radio buttons, text fields, etc. The CHTML_input class derived from the CHTML_OpenElement class serves as the base class for a variety of input elements (see Figure 5).

More details on HTML classes and their relationships is presented in a later chapter.

HTML Processing

The HTML classes can be used to dynamically generate pages. In addition to the classes described in the previous section, there are a number of page classes that are designed to help with HTML processing. The page classes serve as generalized containers for collections of other HTML components, which are mapped to the page. Figure 6 describes the important classes in page class hierarchy.

The CHTMLBasicPage class is as a base class whose features are inherited by the CHTMLPage derived class - it is not intended for direct usage. Through the methods of this class, you can access or set the CgiApplication, Style, and TagMap stored in the class.

The CHTMLPage class when used with the appropriate HTML template file, can generate the 'bolier plate' web pages such as a standard corporate web page, with a corporate logo, a hook for an application-specific logo, a top menubar of links to several databases served by a query program, a links sidebar for application-specific links to relevant sites, a VIEW tag for an application's web interface, a bottom menubar for help links, disclaimers, and other boiler plate links. The template file is a simple HTML text file with named tags (<@tagname@>) which allow the insertion of new HTML blocks into a pre-formatted page.

More details on CHTMLBasicPage, CHTMLPage and related classes is presented in a later chapter.

The OBJECT MANAGER Module

The Object Manager module is a library of C++ classes, which facilitate access to biological sequence data. It makes it possible to transparently download data from the GenBank database, investigate biological sequence data structure, retrieve sequence data, descriptions and annotations.

The Object Manager has been designed to present an interface to users and to minimize their exposure to the details of interacting with biological databases and their underlying data structures. The Object Manager, therefore, coordinates the use of biological sequence data objects, particularly the management of the details of loading data from different data sources.

The NCBI databases and software tools are designed around a particular model of biological sequence data. The data model must be very flexible because the nature of this data is not yet fully understood, and its fundamental properties and relationships are constantly being revised. NCBI uses Abstract Syntax Notation One (ASN.1) as a formal language to describe biological sequence data and its associated information.

The bio sequence data may be huge and downloading all of this data may not be practical or desirable. Therefore, the Object Manager transparently transmits only the data that is really needed and not all of it at once. There is a datatool that generates corresponding data objects (source code and header files) from the object's ASN.1 specification. The Object Manager is able to manipulate these objects.

Biological sequences are identified by a Seq_id, which may have different forms.

The most general container object of bio sequence data, as defined in NCBI data model, is Seq_entry. A great deal of NCBI software is designed to accept a Seq_entry as the primary unit of data. In general, the Seq_entry is defined recursively as a tree of Seq_entry objects, where each node contains either Bioseq or list of other Seq_entry objects and additional data like sequence description, sequence annotations.

Two important concepts in the Object Manager are scope and reference resolution. The client defines a scope as the sources of data where the system uses only "allowed" sources to look for data. Scopes may contain several variants of the same bio sequence (Seq_entry). Since sequences refer to each other, the scope sets may have some data that is common to both scopes. In this case changing data in one scope should be reflected in all other scopes, which "look" at the same data.

The other concept a client uses is reference resolution. Reference resolution is used in situations where different biological sequences can refer to each other. For example, a sequence of amino acids may be the same as sequence of amino acids in another sequence. The data retrieval system should be able to resolve such references automatically answering what amino acids are actually here. Optionally, at the client's request, such automatic resolution may be turned off.

The Object Manager provides a consistent view of the data despite modifications to the data. For example, the data may change during a client's session because new biological data has been uploaded to the database while the client is still processing the old data. In this case, when the client for additional data, the system should retrieve the original bio sequence data, and not the most recent one. However, if the database changes between a client's sessions, then the next time the client session is started, the most recent data is retrieved, unless the client specifically asks for the older data.

The Object Manager is thread safe, and supports multithreading which makes it possible to work with bio sequence data from multiple threads.

The Object Manager includes numerous classes for accessing bio sequence data such as CDataLoader and CDataSource which manage global and local accesses to data, CSeqVector and CSeqMap objects to find and manipulate sequence data, a number of specialized iterators to parse descriptions and annotations, among others. The CObjectManager and

CScope classes provide the foundation of the library, managing data objects and coordinating their interactions.

More details on the Object Manager and related classes is presented in a later chapter.

The SERIAL Module

Click here to see Full Documentation on the Data Serialization Library.

Serial library provides means for loading, accessing, manipulating, and serialization of data in a formatted way. It supports serialization in ASN.1 (text or BER encoding), XML, and JSON formats.

The structure of data is described by some sort of formal language. In our case it can be ASN. 1, DTD or XML Schema. Based on such specification, DATATOOL application, which is part of NCBI C++ toolkit, generates a collection of data storage classes that can be used to store and serialize data. The design purpose was to make these classes as lightweight as possible, moving all details of serialization into specialized classes - “object streams”. Structure of the data is described with the help of “type information”. Data objects contain data and type information only. Any such data storage object can be viewed as a node tree that provides random access to its data. Serial library provides means to traversing this data tree without knowing its structure in advance – using only type information; C++ code generated by DATATOOL makes it possible to access any child node directly.

“Object streams” are intermediaries between data storage objects and input or output stream. They perform encoding or decoding of data according to format specifications. Guided by the type information embedded into data object, on reading they allocate memory when needed, fill in data, and validate that all mandatory data is present; on writing they guarantee that all relevant data is written and that the resulting document is well-formed. All it takes to read or write a top-level data object is one function call – all the details are handled by an object stream.

Closely related to serialization is the task of converting data from one format into another. One approach could be reading data object completely into memory and then writing it in another format. The only problem is that the size of data can be huge. To simplify this task and to avoid storing data in memory, serial library provides “object stream copier” class. It reads data by small chunks and writes it immediately after reading. In addition to small memory footprint, it also works much faster.

Input data can be very large in size; also, reading it completely into memory could not be the goal of processing. Having a large file of data, one might want to investigate information containers only of a particular type. Serial library provides a variety of means for doing this. The list includes read and write hooks, several types of stream iterators, and filter templates. It is worth to note that, when using read hooks to read child nodes, one might end up with an invalid top-level data object; or, when using write hooks, one might begin with an invalid object and fill in missing data on the fly – in hooks.

In essence, “hook” is a callback function that client application provides to serial library. Client application installs the hook, then reads (or writes) data object, and somewhere from the depths of serialization processing, the library calls this hook function at appropriate times, for example, when a data chunk of specified type is about to be read. It is also possible to install context-specific hooks. Such hooks are triggered when serializing a particular object type in a particular context; for example, for all objects of class A which are contained in object B.

The UTIL Module

The UTIL module is collection of some very useful utility classes that implement I/O related functions, algorithms, container classes; text related and thread related functions. Individual facilities include classes to compute checksums, implement interval search trees, lightweight strings, string search, linked sets, random number generation, UTF-8 conversions, registry based DNS, rotating log streams, thread pools, and many others.

The following sections give an overview of the utility classes:

- Checksum
- Console Debug Dump Viewer
- Diff API
- Floating Point Comparison
- Lightweight Strings
- Range Support
- Linked Sets
- Random Number Generator
- Registry based DNS
- Resizing Iterator
- Rotating Log Streams
- Stream Support
- String Search
- Synchronized and blocking queue
- Thread Pools
- UTF 8 Conversion

Checksum

The Checksum class implements CRC32 (Cyclic Redundancy Checksum 32-bit) calculation. The CRC32 is a 32-bit polynomial checksum that has many applications such as verifying the integrity of a piece of data. The CChecksum class implements the CRC32 checksum that can be used to compute the CRC of a sequence of byte values.

The checksum calculation is set up by creating a CChecksum object using the CChecksum constructor and passing it the type of CRC to be calculated. Currently only CRC32 is defined, so you must pass it the enumeration constant eCRC32 also defined in the class.

Data on which the checksum is to be computed is passed to the CChecksum'sAddLine() or AddChars() method as a character array. As data is passed to these methods, the CRC is computed and stored in the class. You can get the value of the computed CRC using the GetChecksum() method. Alternatively, you can use the WriteChecksum() method and pass it a CNcbiOstream object and have the CRC written to the output stream in the following syntax:

/* Original file checksum: lines: *nnnn*, chars: *nnnn*, CRC32: *xxxxxxxx* */

Console Debug Dump Viewer

The UTIL module implements a simple Console Debug Dump Viewer that enables the printing of object information on the console, through a simple console interface. Objects that can be debugged must be inherited from CDebugDumpable class. The CObject is derived from CDebugDumpable, and since most other objects are derived from CObject this makes these objects 'debuggable'.

The Console Debug Dump Viewer is implemented by the CDebugDumpViewer class. This class implements a breakpoint method called Bpt(). This method is called with the name of the object and a pointer to the object to be debugged. This method prompts the user for commands that the user can type from the console:

```
Console Debug Dump Viewer
Stopped at testfile.cpp(120)
current object: myobj = xxxxxx
Available commands:
 t[ypeid] address
 d[ump] address depth
 go
```

The CDebugDumpViewer class also permits the enabling and disabling of debug dump breakpoints from the registry.

Diff API

The Diff API includes the CDiff class for character-based diffs and the CDiffText class for line-based diffs. The API is based on the open source Diff, Match and Patch Library and the Diff Template Library.

To use the Diff API, include xdiff in the LIB line of your application makefile, and include <util/diff/diff.hpp> in your source.

The following sample code shows how to perform both character- and line-based diffs:

```
// Print difference list in human readable format
static void s_PrintDiff(const string& msg, const string& s1, const string&
s2,
 const CDiffList& diff)
{
 NcbiCout << msg << NcbiEndl
 << "Comparing '" << s1 << "' to '" << s2 << "':" << NcbiEndl;
 ITERATE(CDiffList::TList, it, diff.GetList()) {
 string op;
 size_t n1 = 0;
 size_t n2 = 0;

 if (it->IsDelete()) {
 op = "-";
 n1 = it->GetLine().first;
 } else if (it->IsInsert()) {
 op = "+";
 n2 = it->GetLine().second;
 } else {
 op = "=";
 n1 = it->GetLine().first;
 n2 = it->GetLine().second;
 }
 NCbiCout << op << " ("
 << n1 << "," << n2 << ")"
 << ": " << "'" << it->GetString() << "'" << NCbiEndl;
 }
}

// Perform a character-based diff:
{{
 CTempString s1("how now");
 CTempString s2("brown cow");
 CDiff d;
 CDiffList& diffs(d.Diff(s1, s2));
 s_PrintDiff("Line-based diff:", s1, s2, diffs);
}}

// Perform a line-based diff:
{{
 CTempString s1("group 1\nasdf asf\ntttt\nasdf asd");
 CTempString s2("group 2\nqwerty\n\nasdf\nasf asd");
 CDiffText d;
 CDiffList& diffs(d.Diff(s1, s2));
 s_PrintDiff("Line-based diff:", s1, s2, diffs);
}}
```

For more detailed usage, see the test program:

http://www.ncbi.nlm.nih.gov/viewvc/v1/trunk/c%2B%2B/src/util/diff/test/

Floating Point Comparison

For technical reasons, direct comparison of "close" floating point values is simply not reliable on most computers in use today. Therefore, in cases where the values being compared might be close, it is advisable to apply a tolerance when making comparisons to avoid unexpected results.

The UTIL module defines a function, g_FloatingPoint_Compare(), that implements floating point comparison using a tolerance. In practice this means that code like:

```
 if (a < b) {
 if (c == d ) {
 if (e > f) {
```

should be rewritten as:

```
#include <util/floating_point.hpp>
//...
 if (g_FloatingPoint_Compare(a, eFP_LessThan, b,
 eFP_WithPercent, percent) {
 if (g_FloatingPoint_Compare(c, eFP_EqualTo, d,
 eFP_WithFraction, fraction) {
 if (g_FloatingPoint_Compare(e, eFP_GreaterThan, f,
 eFP_WithPercent, percent) {
```

Note that compared variables must be of the same floating point type, otherwise a compile error will be generated.

For further details on this function, see its Doxygen documentation.

For technical details on the subject, including what it means to be close, see "Comparing floating point numbers" by Bruce Dawson.

Lightweight Strings

Class CTempString implements a light-weight string on top of a storage buffer whose lifetime management is known and controlled.

CTempString is designed to perform no memory allocation but provide a string interaction interface congruent with std::basic_string<char>.

As such, CTempString provides a const-only access interface to its underlying storage. Care has been taken to avoid allocations and other expensive operations wherever possible.

CTempString has constructors from std::string and C-style string, which do not copy the string data but keep char pointer and string length.

This way the construction and destruction are very efficient.

Take into account, that the character string array kept by CTempString object must remain valid and unchanged during whole lifetime of the CTempString object.

It's convenient to use the class CTempString as an argument of API functions so that no allocation or deallocation will take place on of the function call.

Linked Sets

The UTIL module defines a template container class, CLinkedMultiset, that can hold a linked list of multiset container types.

The CLinkedMultiset defines iterator methods begin(), end(), find(), lower_bound(), upper_bound(), to help traverse the container. The method get(), fetches the contained value, the method insert() inserts a new value into the container, and the method erase(), removes the specified value from the container.

Random Number Generator

The UTIL module defines the CRandom class that can be used for generating 32-bit unsigned random numbers. The random number generator algorithm is the Lagged Fibonacci Generator (LFG) algorithm.

The random number generator is initialized with a seed value, and then the GetRandom() method is called to get the next random number. You can also specify that the random number value that is returned be in a specified range of values.

Range Support

The UTIL module provides a number of container classes that support a *range* which models an interval consisting of a set of ordered values. the CRange class stores information about an interval, **[*from*, *to*]**, where the ***from*** and ***to*** points are inclusive. This is sometimes called a *closed interval*.

Another class, the CRangeMap class, is similar to the CRange class but allows for the storing and retrieving of data using the interval as key. The time for iterating over the interval is proportional to the amount of intervals produced by the iterator and may not be efficient in some cases.

Another class, the CIntervalTree class, has the same functionality as the CRangeMap class but uses a different algorithm; that is, one based on McCreight's algorithm. Unlike the CRangeMap class, the CIntervalTree class allows several values to have the same key interval. This class is faster and its speed is not affected by the type of data but it uses more memory (about three times as much as CRangeMap) and, as a result, is less efficient when the amount of interval in the set is quite big. For example, the CIntervalTree class becomes less efficient than CRangeMap when the total memory becomes greater than processor cache.

More details on range classes are presented in a later chapter.

Registry based DNS

The UTIL module defines the CSmallDns class that implements a simple registry based DNS server. The CSmallDns class provides DNS name to IP address translations similar to a standard DNS server, except that the database used to store DNS name to IP address mappings is a non-standard local database. The database of DNS names and IP address mappings are kept in a registry-like file named by local_hosts_file using section [LOCAL_DNS].

The CSmallDns has two methods that are responsible for providing the DNS name to IP address translations: the LocalResolveDNS method and the LocalBackResolveDNS method. The LocalResolveDNS method does 'forward' name resolution. That is, given a host name, it returns a string containing the IP address in the dotted decimal notation. The LocalBackResolveDNS method does a 'reverse lookup'. That is, given an IP address as a dotted decimal notation string, it returns the host name stored in the registry.

Resizing Iterator

The UTIL module defines two template classes, the CResizingIterator and the CConstResizingIterator classes that handle sequences represented as packed sequences of elements of different sizes For example, a vector <char> might actually hold 2-bit values, such as nucleotides, or 32-bit integer values.

The purpose of these iterator classes is to provide iterator semantics for data values that can be efficiently represented as a packed sequence of elements regardless of the size.

Rotating Log Streams

The UTIL module defines the CRotatingLogStream class that can be used to implement a rotating log file. The idea being that once the log of messages gets too large, a 'rotation' operation can be performed. The default rotation is to rename the existing log file by appending it with a timestamp, and opening a new log.

The rotating log can be specified as a file, with an upper limit (in bytes) to how big the log will grow. The CRotatingLogStream defines a method called Rotate() that implements the default rotation.

Stream Support

The UTIL module defines a number of portable classes that provide additional stream support beyond that provided by the standard C++ streams. The CByteSource class acts as an abstract base class (see Figure 7), for a number of stream classes derived from it. As the name of the other classes derived from CByteSource suggests, each of these classes provides the methods from reading from the named source. To list a few examples: CFileByteSource is a specialized class for reading from a named file; CMemoryByteSource is a specialized class for reading from a memory buffer; CResultByteSource is a specialized class for reading database results; CStreamByteSource is a specialized class from reading from the C++ input stream (istream); CFStreamByteSource is a specialized class from reading from the C++ input file stream (ifstream).

The classes such as CSubFileByteSource are used to define a slice of the source stream in terms of a start position and a length. The read operations are then confined to this slice.

Additional classes, the CIStreamBuffer and the COStreamBuffer have been defined for standard input and output buffer streams. These can be used in situations where a compiler's implementation of the standard input and output stream buffering is inefficient.

More details on the stream classes are presented in a later chapter.

String Search

The UTIL module defines the CBoyerMooreMatcher class and the CTextFsm class which are used for searching for a single pattern over varying texts.

The CBoyerMooreMatcher class, as the name suggests, uses the Boyer-Moore algorithm for string searches. The CTextFsm is a template class that performs the search using a finite state automaton for a specified to be matched data type. Since the matched data type is often a string, the CTextFsa class is defined as a convenience by instantiating the CTextFsm with the matched type template parameter set to string.

The search can be setup as a case sensitive or case insensitive search. The default is case sensitive search. In the case of the CBoyerMooreMatcher class, the search can be setup for any pattern match or a whole word match. A whole word match means that a pattern was found to be between white spaces. The default is any pattern match.

Synchronized and blocking queue

The UTIL module defines class CSyncQueue which implements a thread-safe queue that has “blocking” semantics: when queue is empty Pop() method will effectively block execution until some elements will be added to the queue; when queue have reached its maximum size Push() method will block execution until some elements will be extracted from queue. All these operations can be controlled by timeout. Besides that CSyncQueue is not bound to first-in-first-out queue paradigm. It has underlying stl container (deque by default) which will define the nature of queue. This container is set via template parameter to CSyncQueue and can be deque, vector, list, CSyncQueue_set, CSyncQueue_multiset and CSyncQueue_priority_queue (the latter three are small addons to STL set, multiset and priority_queue for the sake of compatibility with CSyncQueue).

There is also CSyncQueue::TAccessGuard class which can lock the queue for some bulk operations if during them queue should not be changed by other threads.

For more details on CSyncQueue look here: http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/doxyhtml/CSyncQueueDescription.html.

Thread Pools

The UTIL module defines a number of classes implementing pool of threads.

CThreadPool is the main class. It executes any tasks derived from the CThreadPool_Task class. The number of threads in pool is controlled by special holder of this policy — object derived from CThreadPool_Controller (default implementation is CThreadPool_Controller_PID based on Proportional-Integral-Derivative algortithm). All threads executing by CThreadPool are the instances of CThreadPool_Thread class or its derivatives.

More details on threaded pool classes are presented in a later chapter.

UTF 8 Conversion

The UTIL module provides a number of functions to convert between UTF-8 representation, ASCII 7-bit representation and Unicode representations. For example, StringToCode() converts the first UTF-8 character in a string to a Unicode symbol, and StringToVector() converts a UTF-8 string into a vector of Unicode symbols.

The result of a conversion can be success, out of range, or a two character sequence of the skip character (0xFF) followed by another character.

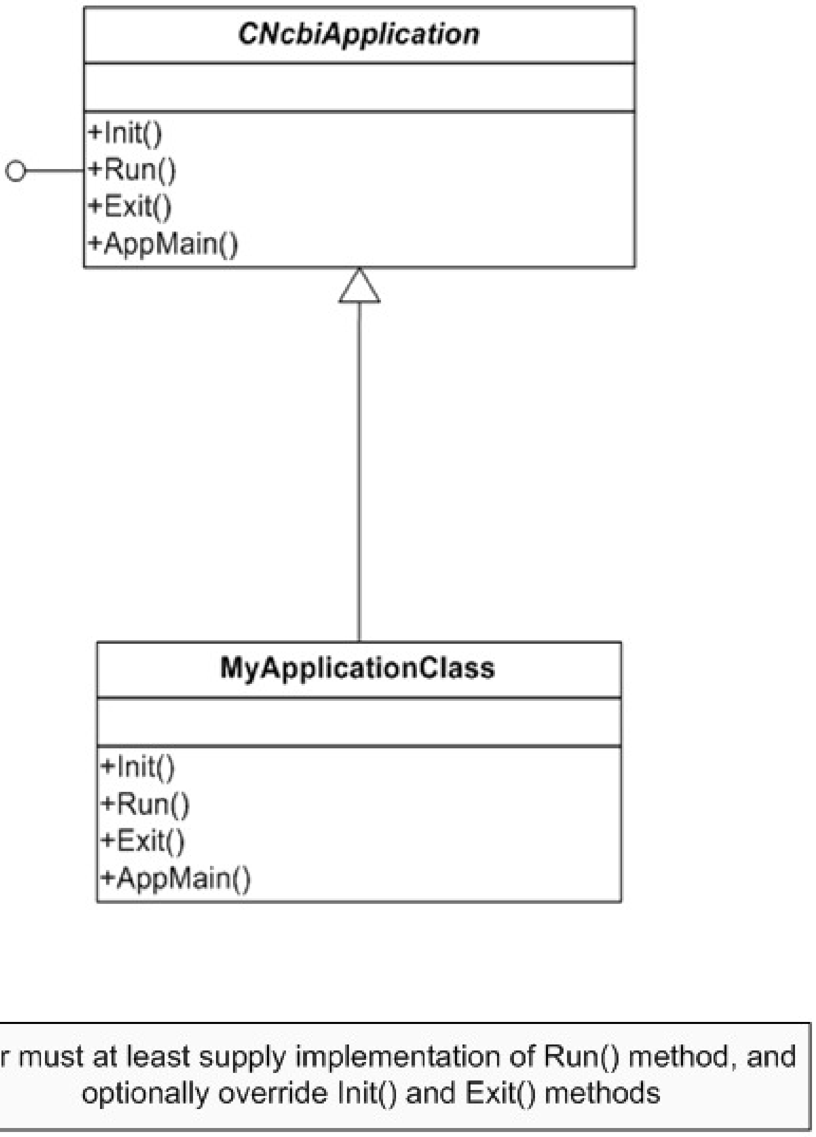

CNcbiApplication
+Init()
+Run()
+Exit()
+AppMain()
MyApplicationClass
+Init()
+Run()
+Exit()
+AppMain()
User must at least supply implementation of Run() method, and optionally override Init() and Exit() methods

Figure 1. The CNcbiApplication class

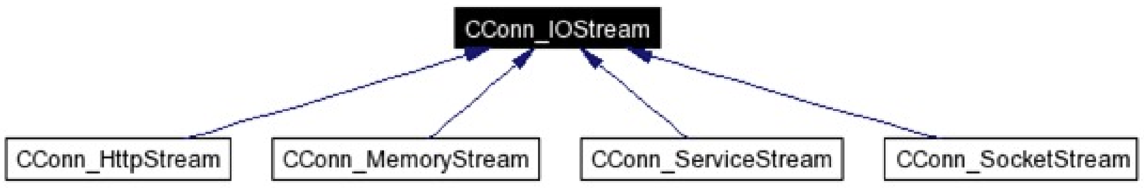


Figure 2. Connection stream classes

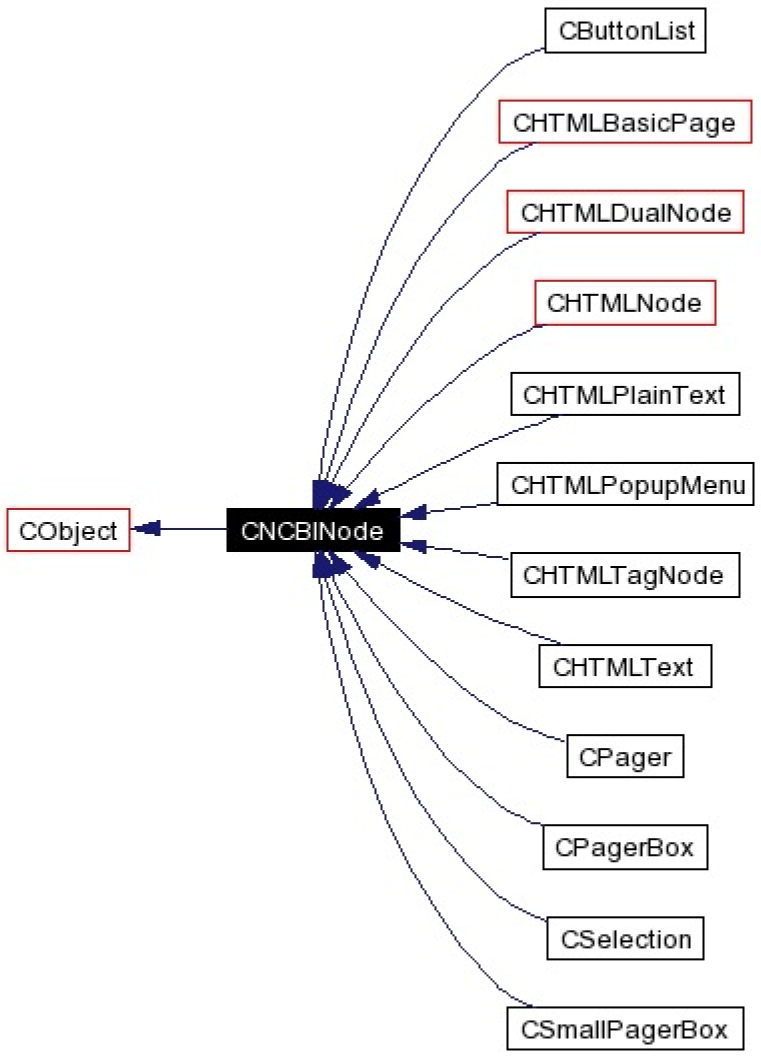


Figure 3. HTML classes derived from CNCBINode

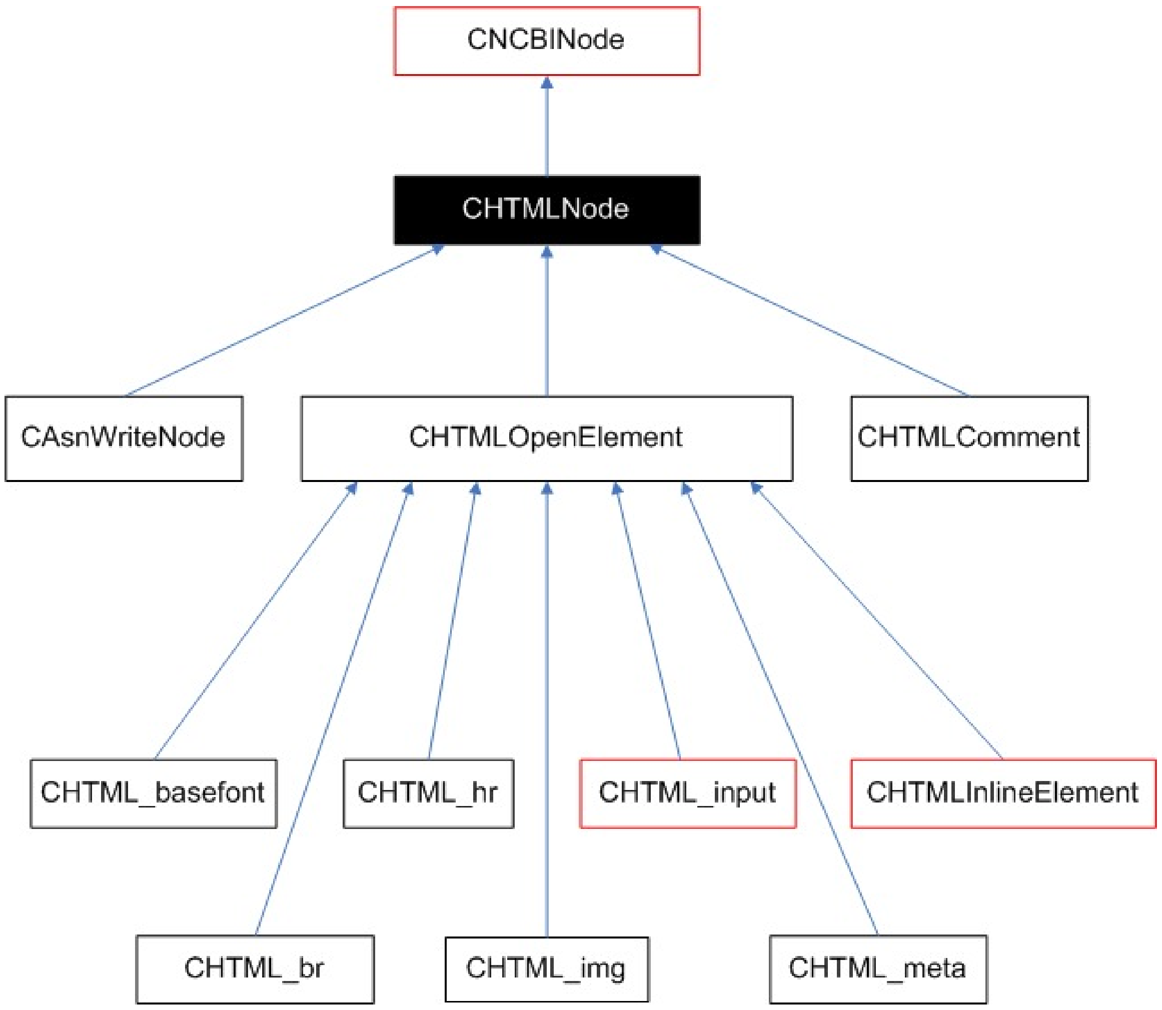


Figure 4. The CHTMLNode class and its derived classes

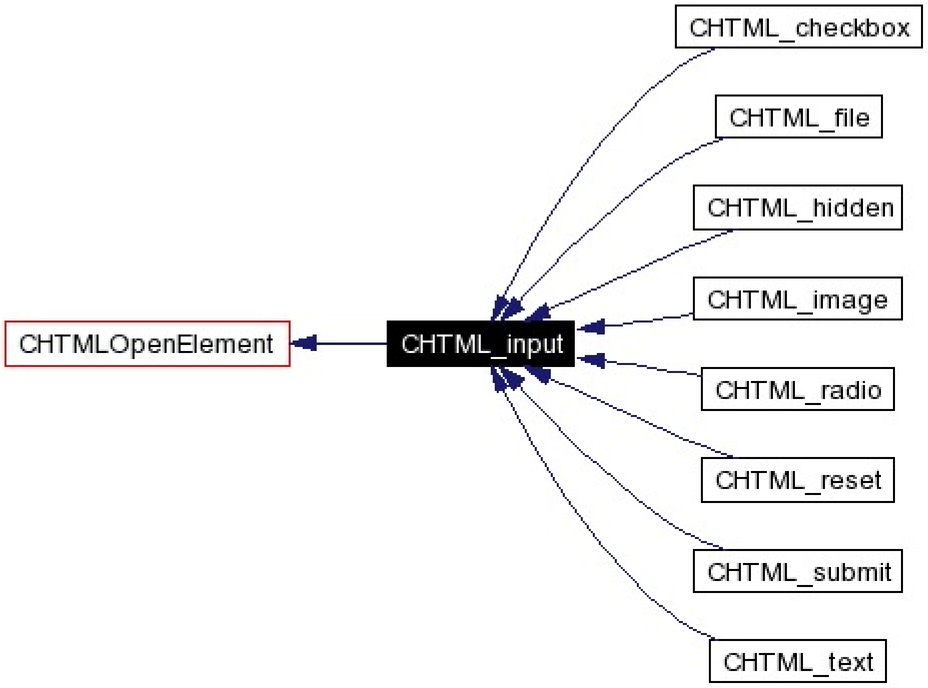


Figure 5. The CHTML_input class and its derived classes

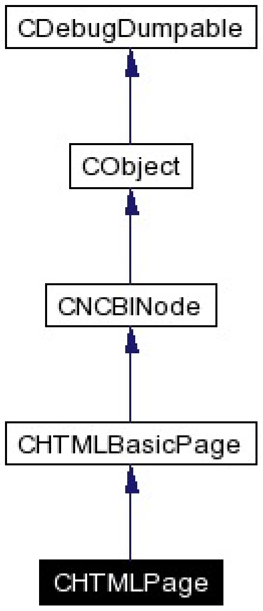


Figure 6. HTML page classes

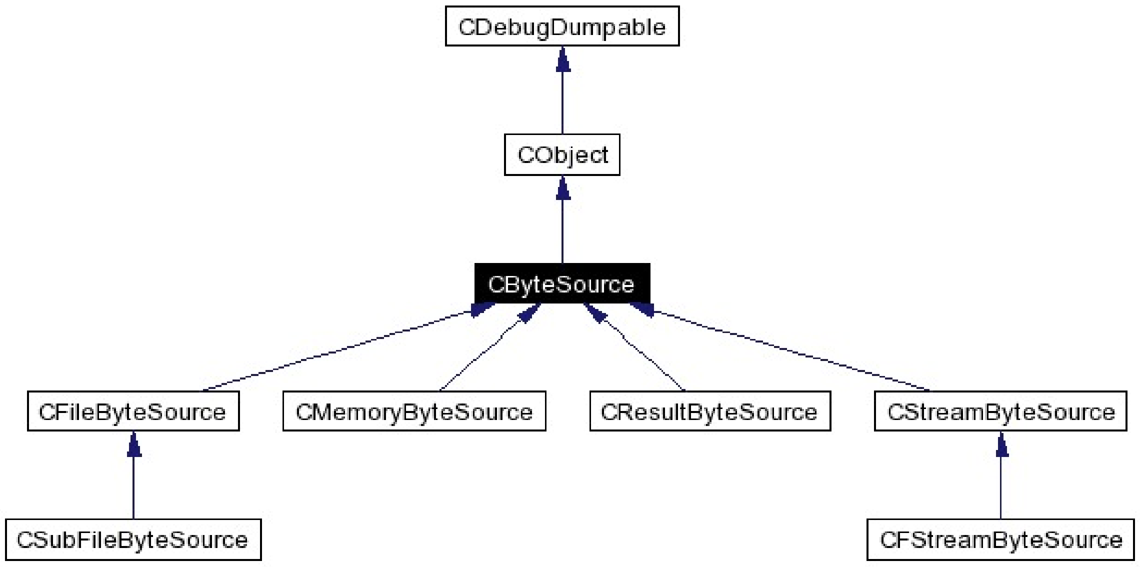


Figure 7. Relationship between CByteSource and its derived classes

2: Getting Started

Last Update: June 29, 2012.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This section is intended as a bird's-eye view of the Toolkit for new users, and to give quick access to important reference links for experienced users. It lays out the general roadmap of tasks required to get going, giving links to take the reader to detailed discussions and supplying a number of simple, concrete test applications.

Note: Much of this material is platform-neutral, although the discussion is platform-centric. Users would also benefit from reading the instructions specific to those systems and, where applicable, how to use Subversion (SVN) with MS Windows and Mac OS.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Quick Start

A good deal of the complication and tedium of getting started has thankfully been wrapped by a number of shell scripts. They facilitate a 'quick start' whether starting anew or within an existing Toolkit work environment. ('Non-quick starts' sometimes cannot be avoided, but they are considered elsewhere.)

- **Get the Source Tree (see** Figure 1**)**
  - Retrieve via SVN (in-house | public), **or**
  - Download via FTP, **or**
  - Run svn_core *(requires a SVN repository containing the C++ Toolkit; for NCBI users)*
- **Configure the build tree (see** Figure 2**)**
  - Use the configure script, **or**
  - Use a compiler-specific wrapper script (e.g. compilers/unix/*.sh).
- **Build the C++ Toolkit from** makefiles **and** meta-makefiles(if required)
  - make all_r for a recursive make, **or**
  - make all to make only targets for the current directory.
- **Work on your new or existing application or library** the scripts new_project and (for an existing Toolkit project) import_project help to set up the appropriate makefiles and/or source.

In a nutshell, that's all it takes to get up and running. The download, configuration, installation and build actions are shown for two cases in this sample.

The last item, employing the Toolkit in a project, completely glosses over the substantial issue of how to use the installed Toolkit. Where does one begin to look to identify the functionality to solve your particular problem, or indeed, to write the simplest of programs? "Basics of Using the C++ Toolkit" will deal with those issues. Investigate these and other topics with the set of sample applications. See Examples for further cases that employ specific features of the NCBI C++ Toolkit.

Example Applications

The suite of application examples below highlight important areas of the Toolkit and can be used as a starting point for your own development. Note that you may generate the sample application code by running the new_project script for that application. The following examples are now available:

- app/basic - This example builds two applications: a generic application (basic_sample) to demonstrate the use of key Toolkit classes, and an example program (multi_command) that accepts multiple command line forms.
- app/alnmgr - Creates an alignment manager application.
- app/asn - Creates a library based on an ASN.1 specification, and a test application.

- app/blast - Creates an application that uses BLAST.
- app/cgi - Creates a Web-enabled CGI application.
- app/dbapi - Creates a database application.
- app/eutils - Creates an eUtils client application.
- app/lds - Creates an application that uses local data storage (LDS).
- app/netcache - Creates an application that uses NetCache.
- app/netschedule - Creates an NCBI GRID application that uses NetSchedule.
- app/objects - Creates an application that uses ASN.1 objects.
- app/objmgr - The Toolkit manipulates biological data objects in the context of an Object Manager class (CObjectManager). This example creates an application that uses the object manager.
- app/sdbapi - Creates a database application that uses SDBAPI.
- app/serial - Creates a dozen applications that demonstrate using serial library hooks, plus a handful of other applications that demonstrate other aspects of the serial library.
- app/soap/client - Creates a SOAP client application.
- app/soap/server - Creates a SOAP server application.
- app/unit_test - Creates an NCBI unit test application.

To build an example use its accompanying Makefile.

Example Libraries

The following example libraries can be created with new_project and used as a starting point for a new library:

- lib/basic - Creates a trivial library (it finds files in PATH) for demonstrating the basics of the build system for libraries. This example library includes a simple test application.
- lib/asn - Creates an ASN.1 object project.
- lib/dtd - Creates an XML DTD project.
- lib/xsd - Creates an XML Schema project.

Source Tree Availability

The source tree is available through FTP, SVN and by running special scripts. The following subsections discuss these topics in more detail:

- FTP Availability
- SVN Availability
- Availability via Shell Scripts

FTP Availability

The Toolkit source is available via ftp at ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/CURRENT/, and the archives available, with unpacking instructions, are listed on the download page. If you plan to modify the Toolkit source in any way with the ftp code, it is strongly advised that it be placed under a source code control system (preferably SVN) so that you can rollback to an earlier revision without having to ftp the entire archive once again.

SVN Availability

NCBI users can obtain the source tree directly from the internal SVN repository.

A read-only repository is also available to the public.

Availability via Shell Scripts

For NCBI users, the various shell scripts in $NCBI/c++/scripts tailor the working codebase and can prepare the work environment for new projects. Except where noted, an active Toolkit SVN repository is required, and obviously in all cases a version of the Toolkit must be accessible.

- svn_core. Details on svn_core are discussed in a later chapter.
- import_project. Details on import_project are discussed in a later chapter.
- new_project. Details on new_project are discussed in a later chapter.
- update_projects. Details on update_core and update_projects are covered in later chapter.

Source Tree Contents

The following topics are discussed in this section:

- Top-Level Source Organization
- The Core NCBI C++ Toolkit
- Source Tree for Individual Projects
- The Makefile Templates
- The New Module Stubs

Top-Level Source Organization

The NCBI C++ Toolkit source tree (see Figure 1) is organized as follows:

- src/ -- a hierarchical directory tree of NCBI C++ projects. Contained within src are all source files (*.cpp, *.c), along with private header files (*.hpp, *.h), makefiles (Makefile.*, including Makefile.mk), scripts (*.sh), and occasionally some project-specific data
- include/ -- a hierarchical directory tree whose structure mirrors the src directory tree. It contains only public header files (*.hpp, *.h).

Example:include/corelib/ contains public headers for the sources located in src/corelib/

- scripts/ -- auxiliary scripts, including those to help manage interactions with the NCBI SVN code repository, such as import_project, new_project, and svn_core.
- files for platform-specific configuration and installation:
  - compilers/ -- directory containing compiler-specific configure wrappers (unix/*.sh) and miscellaneous resources and build scripts for MS Windows/MacOS platforms
  - configure -- a multi-platform configuration shell script (generated from template configure.ac using autoconf)
  - various scripts and template files used by configure, autoconf
- doc/ -- NCBI C++ documentation, including a library reference, configuration and installation instructions, example code and guidelines for **everybody** writing code for the NCBI C++ Toolkit.

The Core NCBI C++ Toolkit

The 'core' libraries of the Toolkit provide users with a highly portable set of functionality. The following projects comprise the portable core of the Toolkit:

corelib connect cgi html util

Consult the library reference (Part 3 of this book) for further details.

Source Tree for Individual Projects

For the overall NCBI C++ source tree structure see Top-Level Source Organization above.

An individual project contains the set of source code and/or scripts that are required to build a Toolkit library or executable. In the NCBI source tree, projects are identified as sub-trees of the src, and include directories of the main C++ tree root. For example, corelib and objects/objmgr are both projects. However, note that a project's code exists in two sibling directories: the public headers in include/ and the source code, private headers and makefiles in src.

The contents of each project's source tree are:

- *.cpp, *.hpp -- project's source files and private headers
- Makefile.in -- a meta-makefile to specify which local projects (described in Makefile.*.in) and sub-projects(located in the project subdirectories) must be built
- Makefile.*.lib, Makefile.*.app -- customized makefiles to build a library or an application
- Makefile.* -- "free style" makefiles
- sub-project directories (if any)

The Makefile Templates

Each project is built by customizing a set of generic makefiles. These generic makefile templates (Makefile.*.in) are found in src and help to control the assembly of the entire Toolkit via recursive builds of the individual projects. (The usage of these makefiles and other configurations issues are summarized below and detailed on the Working with Makefiles page.)

- Makefile.in -- makefile to perform a recursive build in all project subdirectories
- Makefile.meta.in -- included by all makefiles that provide both local and recursive builds
- Makefile.mk.in -- included by all makefiles; sets a lot of configuration variables
- Makefile.lib.in -- included by all makefiles that perform a "standard" library build, when building only static libraries.
- Makefile.dll.in -- included by all makefiles that perform a "standard" library build, when building only shared libraries.
- Makefile.both.in -- included by all makefiles that perform a "standard" library build, when building both static and shared libraries.
- Makefile.lib.tmpl.in -- serves as a template for the project customized makefiles (Makefile.*.lib[.in]) that perform a "standard" library build
- Makefile.app.in -- included by all makefiles that perform a "standard" application build
- Makefile.app.tmpl.in -- serves as a template for the project customized makefiles (Makefile.*.app[.in]) that perform a "standard" application build
- Makefile.rules.in, Makefile.rules_with_autodep.in -- instructions for building object files; included by most other makefiles

The New Module Stubs

A Toolkit module typically consists of a header (*.hpp) and a source (*.cpp) file. Use the stubs provided, which include boilerplate such as the NCBI disclaimer and SVN revision information, to easily start a new module. You may also consider using the sample code described above for your new module.

Decide Where You Will Work (in-tree, in a subtree, out-of-tree)

Depending on how you plan to interact with the NCBI C++ Toolkit source tree, the Toolkit has mechanisms to streamline how you create and manage projects. The simplest case is to work out-of-tree in a private directory. This means that you are writing new code that needs only to link with pre-built Toolkit libraries. If your project requires the source for a limited set of Toolkit projects it is often sufficient to work in a subtree of the Toolkit source distribution.

Most users will find it preferable and fully sufficient to work in a subtree or a private directory. Certain situations and users (particularly Toolkit developers) do require access to the full Toolkit source tree; in such instances one must work in-tree.

Basic Installation and Configuration Considerations

Note: Much of this discussion is Unix-centric. Windows and Mac users would also benefit from reading the instructions specific to those systems.

The configuration and installation process is automated with the configure script and its wrappers in the compilers directory. These scripts handle the compiler- and platform-dependent Toolkit settings and create the build tree (see Figure 2) skeleton. The configured build tree, located in <builddir>, is populated with customized meta-makefile, headers and source files. Most system-dependence has been isolated in the <builddir>/inc/ncbiconf.h header. By running make all_r from <builddir>, the full Toolbox is built for the target platform and compiler combination.

Summarized below are some basic ways to control the installation and configuration process. More comprehensive documentation can be found at config.html.

- **A Simple Example Build**
- configure Options View the list of options by running ./configure --help
- **Enable/Disable Debugging**
- Building Shared and/or Static Libraries Shared libraries (DLL's) can be used in Toolkit executables and libraries for a number of tested configurations. Note that to link with the shared libraries at run time a valid runpath must be specified.
- If you are outside NCBI, make sure the paths to your third party libraries are correctly specified. See Site-Specific Third Party Library Configuration for details.
- Influencing configure via Environment Variables Several environment variables control the tools and flags employed by configure. The generic ones are: CC, CXX, CPP, AR, RANLIB, STRIP, CFLAGS, CXXFLAGS, CPPFLAGS, LDFLAGS, LIBS. In addition, you may manually set various localization environment variables.
- Multi-Thread Safe Compilation
- **Controlling Builds of Optional Projects** You may selectively build or not build one of the optional projects ("serial", "ctools", "gui", "objects", "internal") with configure flags. If an optional project is not configured into your distribution, it can be added later using the import_projects script.

- Adjust the Configuration of an Existing Build If you need to update or change the configuration of an existing build, use the reconfigure.sh or relocate.sh script.
- Working with Multiple build trees Managing builds for a variety of platforms and/or compiler environments is straightforward. The configure/install/build cycle has been designed to support the concurrent development of multiple builds from the same source files. This is accomplished by having independent build trees that exist as sibling directories. Each build is configured according to its own set of configuration options and thus produces distinct libraries and executables. All builds are nonetheless constructed from the same source code in $NCBI/c++/{src, include}.

Basics of Using the C++ Toolkit

The following topics are discussed in this section:

- Compiling and Linking with make
- Makefile Customization
- Basic Toolkit Coding Infrastructure
- Key Classes
- The Object Manager and datatool
- Debugging and Diagnostic Aids
- Coding Standards and Guidelines

Compiling and Linking with make

The NCBI C++ Toolkit uses the standard Unix utility make to build libraries and executable code, using instructions found in makefiles. More details on compiling and linking with make can be found in a later chapter.

To initiate compilation and linking, run make:

```
make -f <Makefile_Name> [<target_name>]
```

When run from the top of the build tree, this command can make the entire tree (with target all_r). If given within a specific project subdirectory it can be made to target just that project. The Toolkit has in its src directory templates (e.g., Makefile.*.in) for makefiles and meta-makefiles that define common file locations, compiler options, environment settings, and standard make targets. Each Toolkit project has a specialized meta-makefile in its src directory. The relevant meta-makefile templates for a project, e.g., Makefile.in, are customized by configure and placed in its build tree. For new projects, whether in or out of the C++ Toolkit tree, the programmer must provide either makefiles or meta-makefiles.

Makefile Customization

Fortunately, for the common situations where a script was used to set up your source, or if you are working in the C++ Toolkit source tree, you will usually have correctly customized makefiles in each project directory of the build tree. For other cases, particularly when using the new_project script, some measure of user customization may be needed. The more frequent customizations involve (see "Working with Makefiles" or "Project makefiles" for a full discussion):

- meta-makefile macros: APP_PROJ, LIB_PROJ, SUB_PROJ, USR_PROJ Lists of applications, libraries, sub-projects, and user projects, respectively, to make.

- Library and Application macros: APP, LIB, LIBS, OBJ, SRC List the application name to build, Toolkit library(ies) to make or include, non-Toolkit library(ies) to link, object files to make, and source to use, respectively.
- Compiler Flag Macros: CFLAGS, CPPFLAGS, CXXFLAGS, LDFLAGS Include or override C compiler, C/C++ preprocessor, C++ compiler, and linker flags, respectively. Many more localization macros are also available for use.
- Altering the Active Version of the Toolkit You can change the active version of NCBI C++ toolkit by manually setting the variable $(builddir) in Makefile.foo_[app|lib] to the desired toolkit path, e.g.: builddir = $(NCBI)/c++/GCC-Release/build. Consult this list or, better, look at the output of 'ls -d $NCBI/c++/*/build' to see those pre-built Toolkit builds available on your system.

Basic Toolkit Coding Infrastructure

Summarized below are some features of the global Toolkit infrastructure that users may commonly employ or encounter.

- ***The NCBI Namespace Macros*** The header ncbistl.hpp defines three principal namespace macros: NCBI_NS_STD, NCBI_NS_NCBI and NCBI_USING_NAMESPACE_STD. Respectively, these refer to the standard C++ std:: namespace, a local NCBI namespace ncbi:: for Toolkit entities, and a namespace combining the names from NCBI_NS_STD and NCBI_NS_NCBI.
- ***Using the NCBI Namespaces*** Also in ncbistl.hpp are the macros BEGIN_NCBI_SCOPE and END_NCBI_SCOPE. These bracket code blocks which define names to be included in the NCBI namespace, and are invoked in nearly all of the Toolkit headers (see example). To use the NCBI namespace in a code block, place the USING_NCBI_SCOPE macro before the block references its first unqualified name. This macro also allows for unqualified use of the std:: namespace. Much of the Toolkit source employs this macro (see example), although it is possible to define and work with other namespaces.
- ***Configuration-Dependent Macros and*** ncbiconf.h #ifdef tests for the configuration-dependent macros, for example _DEBUG or NCBI_OS_UNIX, etc., are used throughout the Toolkit for conditional compilation and accommodate your environment's requirements. The configure script defines many of these macros; the resulting #define's appear in the ncbiconf.h header and is found in the <builddir>/inc directory. It is not typically included explicitly by the programmer, however. Rather, it is included by other basic Toolkit headers (e.g., ncbitype.h, ncbicfg.h, ncbistl.hpp) to pick up configuration-specific features.
- ***NCBI Types (***ncbitype.h, ncbi_limits.[h|hpp]***)*** To promote code portability developers are strongly encouraged to use these standard C/C++ types whenever possible as they are ensured to have well-defined behavior throughout the Toolkit. Also see the current type-use rules. The ncbitype.h header provides a set of fixed-size integer types for special situations, while the ncbi_limits.[h| hpp] headers set numeric limits for the supported types.
- ***The*** ncbistd.hpp ***header*** The NCBI C++ standard #include's and #defin'itions are found in ncbistd.hpp, which provides the interface to many of the basic Toolkit modules. The explicit NCBI headers included by ncbistd.hpp are: ncbitype.h, ncbistl.hpp, ncbistr.hpp, ncbidbg.hpp, ncbiexpt.hpp and ncbi_limits.h.
- ***Portable Stream Handling*** Programmers can ensure portable stream and buffer I/O operations by using the NCBI C++ Toolkit stream wrappers, typedef's and #define's declared in the ncbistre.hpp. For example, always use CNcbiIstream instead of

YourFavoriteNamespace::istream and favor NcbiCin over cin. A variety of classes that perform case-conversion and other manipulations in conjunction with NCBI streams and buffers are also available. See the source for details.

- ***Use of the C++ STL (Standard Template Library) in the Toolkit*** The Toolkit employs the STL's set of template container classes, algorithms and iterators for managing collections of objects. Being standardized interfaces, coding with them provides portability. However, one drawback is the inability of STL containers to deal with reference objects, a problem area the Toolkit's CRef and CObject classes largely remedy.
- ***Serializable Objects, the ASN.1 Data Types and*** datatool The ASN.1 data model for biological data underlies all of the C and C++ Toolkit development at NCBI. The C++ Toolkit represents the ASN.1 data types as serializable objects, that is, objects able to save, restore, or transmit their state. This requires knowledge of an object's type and as such a CTypeInfo object is provided in each class to encapsulate type information.
  Additionally, object stream classes (CObject[IO]Stream, and subclasses) have been designed specifically to perform data object serialization. The nuts-and-bolts of doing this has been documented on the Processing Serial Data page, with additional information about the contents and parsing of ASN.1-derived objects in Traversing a Data Structure.Each of the serializable objects appears in its own subdirectory under [src| include]/objects. These objects/* projects are configured differently from the rest of the Toolkit, in that header and source files are auto-generated from the ASN.1 specifications by the datatool program. The --with-objects flag to configure also directs a build of the user classes for the serializable objects.

Key Classes

For reference, we list some of the fundamental classes used in developing applications with the Toolkit. Some of these classes are described elsewhere, but consult the library reference (Part 3 of this book) and the source browser for complete details.

- CNcbiApplication (abstract class used to define the basic functionality and behavior of an NCBI application; **this application class effectively supersedes the C-style main() function**)
- CArgDescriptions, CArgs, and CArgValue (command-line argument processing)
- CNcbiEnvironment (store, access, and modify environment variables)
- CNcbiRegistry (load, access, modify and store runtime information)
- CNcbiDiag (error handling for the Toolkit;)
- CObject (base class for objects requiring a reference count)
- CRef (a reference-counted smart pointer; particularly useful with STL and template classes)
- CObject[IO]Stream (serialized data streams)
- CTypeInfo and CObjectTypeInfo (Runtime Object Type Information; extensible to user-defined types)
- CObjectManager, etc. (classes for working with biological sequence data)
- CCgiApplication, etc. (classes to create CGI and Fast-CGI applications and handle CGI Diagnostics)
- CNCBINode, etc. (classes representing HTML tags and Web page content)
- Iterator Classes (easy traversal of collections and containers)

- Exception Handling (classes, macros and tracing for exceptions)

The Object Manager and datatool

The datatool processes the ASN.1 specifications in the src/objects/directories and is the C++ Toolkit's analogue of the C Toolkit's asntool. The goal of datatool is to generate the class definitions corresponding to each ASN.1 defined data entity, including all required type information. As ASN.1 allows data to be selected from one of several types in a choice element, care must be taken to handle such cases.

The Object Manager is a C++ Toolkit library whose goal is to transparently download data from the GenBank database, investigate bio sequence data structure, and retrieve sequence data, descriptions and annotations. In the library are classes such as CDataLoader and CDataSource which manage global and local accesses to data, CSeqVector and CSeqMap objects to find and manipulate sequence data, a number of specialized iterators to parse descriptions and annotations, among others. The CObjectManager and CScope classes provide the foundation of the library, managing data objects and coordinating their interactions.

Jump-start and Object Manager FAQ are all available to help new users.

Debugging and Diagnostic Aids

The Toolkit has a number of methods for catching, reporting and handling coding bugs and exceptional conditions. During development, a debug mode exists to allow for assertions, traces and message posting. The standard C++ exception handling (which should be used as much as possible) has been extended by a pair of NCBI exception classes, CErrnoException and CParseException and additional associated macros. Diagnostics, including an ERR_POST macro available for routine error posting, have been built into the Toolkit infrastructure.

For more detailed and extensive reporting of an object's state (including the states of any contained objects), a special debug dump interface has been implemented. All objects derived from the CObject class, which is in turn derived from the abstract base class CDebugDumpable, automatically have this capability.

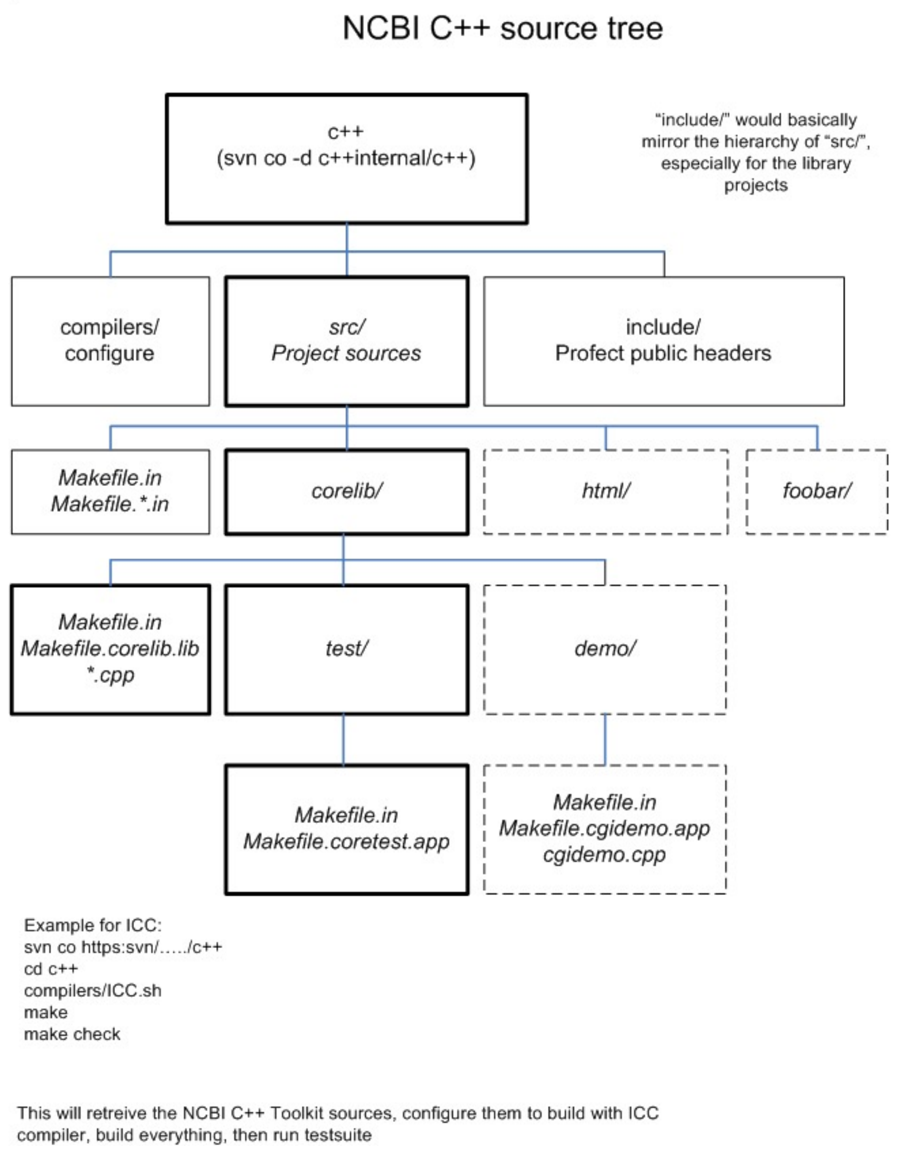
Coding Standards and Guidelines

All C++ source in the Toolkit has a well-defined coding style which shall be used for new contributions and is highly encouraged for all user-developed code. Among these standards are

- variable naming conventions (for types, constants, class members, etc.)
- using namespaces and the NCBI name scope
- code indentation (4-space indentation, **no** tab symbols)
- declaring and defining classes and functions

Noteworthy Files

A list of important files is given in Table 1.


NCBI C++ source tree
c++
(svn co -d c++internal/c++)
"include/" would basically mirror the hierarchy of "src/", especially for the library projects
compilers/ configure
src/ Project sources
include/ Profect public headers
Makefile.in Makefile.*.in
corelib/
html/
foobar/
Makefile.in Makefile.corelib.lib *.cpp
test/
demo/
Makefile.in Makefile.coretest.app
Makefile.in Makefile.cgidemo.app cgidemo.cpp

```
Example for ICC:
svn co https:svn/...../c++
cd c++
compilers/ICC.sh
make
make check
```

This will retreive the NCBI C++ Toolkit sources, configure them to build with ICC compiler, build everything, then run testsuite

Figure 1. NCBI C++ Source Tree

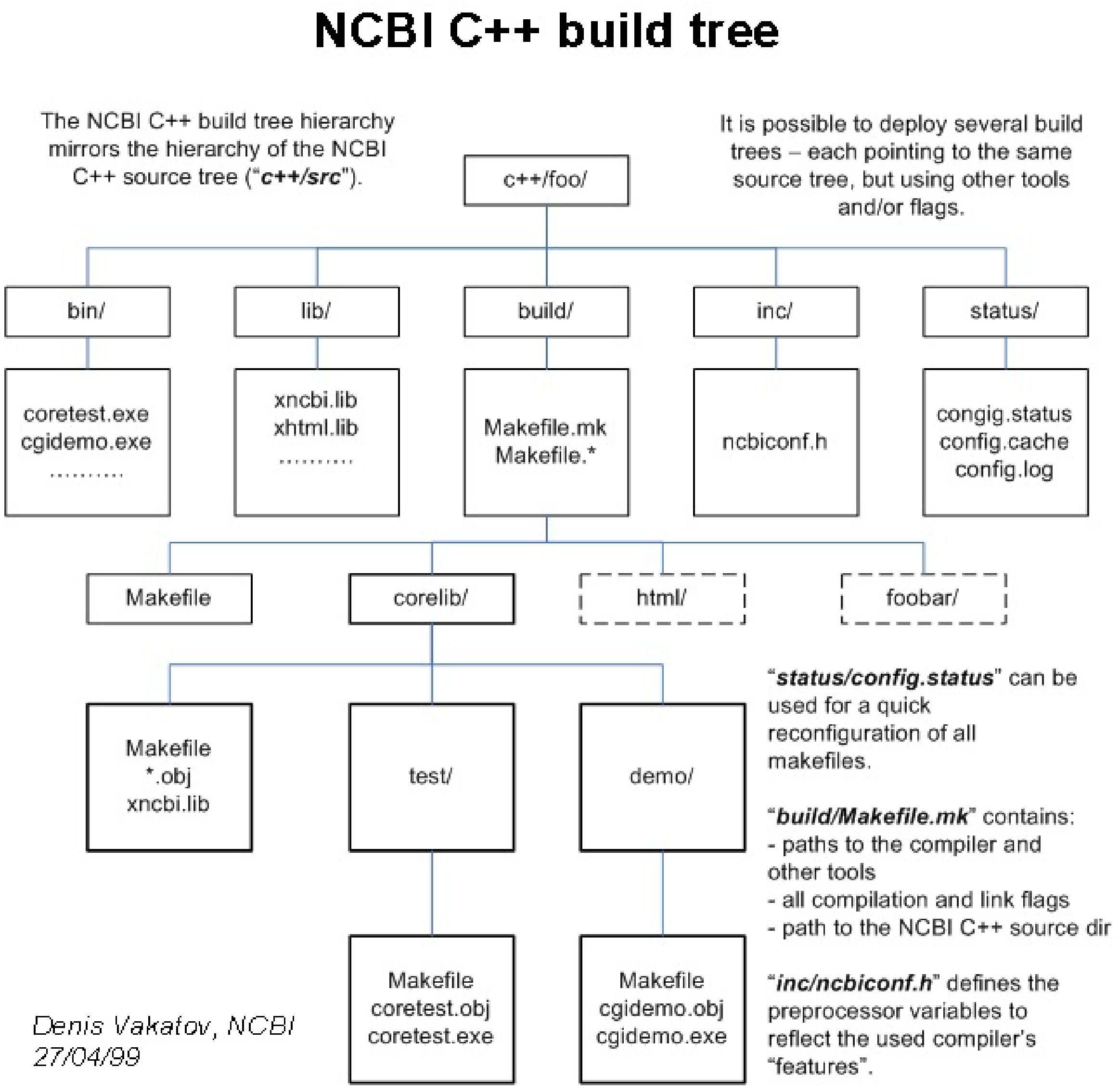

NCBI C++ build tree
The NCBI C++ build tree hierarchy mirrors the hierarchy of the NCBI C++ source tree ("***c++/src***").
It is possible to deploy several build trees – each pointing to the same source tree, but using other tools and/or flags.
c++/foo/
bin/
lib/
build/
inc/
status/
coretest.exe
cgidemo.exe
……….
xncbi.lib
xhtml.lib
……….
Makefile.mk
Makefile.*
ncbiconf.h
congig.status
config.cache
config.log
Makefile
corelib/
html/
foobar/
Makefile
*.obj
xncbi.lib
test/
demo/
Makefile
coretest.obj
coretest.exe
Makefile
cgidemo.obj
cgidemo.exe
"***status/config.status***" can be used for a quick reconfiguration of all makefiles.
"***build/Makefile.mk***" contains:
- paths to the compiler and other tools
- all compilation and link flags
- path to the NCBI C++ source dir
"***inc/ncbiconf.h***" defines the preprocessor variables to reflect the used compiler's "features".
Denis Vakatov, NCBI
27/04/99

Figure 2. NCBI C++ Build Tree

Table 1. Noteworthy Files

Filename (relative to $NCBI/c++)	**Description**
compilers/*/<compiler_name>.sh	Use the configure shell script, or one of its compiler-specific wrappers, to fully configure and install all files required to build the Toolkit.
import_project	Import only an existing Toolkit project into an independent subtree of your current Toolkit source tree. (Requires a SVN source repository.)
update_{core|projects}	Update your local copy of either the core Toolkit or set of specified projects. (Requires a SVN source repository.)
new_project	Set up a new project outside of the NCBI C++ Toolkit tree to access pre-built version of the Toolkit libraries. Sample code can be requested to serve as a template for the new module.
src/<project_dir>/Makefile.in src/<project_dir>/ Makefile.<project>.{app, lib}	Customized meta-makefile template and the corresponding datafile to provide project-specific source dependencies, libraries, compiler flags, etc. This information is accessed by configure to build a projects's meta-makefile (see below).
doc/framewrk.{cpp|hpp}	Basic templates for source and header files that can be used when starting a new module. Includes common headers, the NCBI disclaimer and SVN keywords in a standard way.
CHECKOUT_STATUS	This file summarizes the local source tree structure that was obtained when using one of the shell scripts in scripts. (Requires a SVN source repository.)
Build-specific Files (relative to $NCBI/c++/<builddir>)	**Description**
Makefile Makefile.mk Makefile.meta	These are the primary makefiles used to build the entire Toolkit (when used recursively). They are customized for a specific build from the corresponding *.in templates in $NCBI/c++/src. Makefile is the master, top-level file, Makefile.mk sets many make and shell variables and Makefile.meta is where most of the make targets are defined.
<project_dir>/Makefile <project_dir>/ Makefile.<project>_{app, lib}	Project-specific custom meta-makefile and makefiles, respectively, configured from templates in the src/ hierarchy and any pertinent src/<project_dir>/Makefile.<project>.{app, lib} files (see REF TO OLD ANCHOR: get_started.html_ref_TmplMetaMake<secref rid="get_started.html_ref_ImptFiles">above</secref>).
inc/ncbiconf.h	Header that #define's many of the build-specific constants required by the Toolkit. This file is auto-generated by the configure script, and some pre-built versions do exist in compilers.
reconfigure.sh	Update the build tree due to changes in or the addition of configurable files (*.in files, such as Makefile.in or the meta-makefiles) to the source tree.
relocate.sh	Adjust paths to this build tree and the relevant source tree.
corelib/ncbicfg.c	Define and manage the runtime path settings. This file is auto-generated by the configure script.
status/config.{cache|log|status}	These files provide information on configure's construction of the build tree, and the cache of build settings to expedite future changes.

Part 2: Development Framework

Part 2 deals with the development framework, and discusses how to download the Toolkit code and configure the source code for different platforms, how to build the libraries and executables, how to setup projects, and the recommended style for writing code. The following is a list of chapters in this part:

3: Retrieve the Source Code (FTP and Subversion)

Created: April 1, 2003.
Last Update: May 16, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The first step in working with the C++ Toolkit is getting the source code, which can be either downloaded from anonymous FTP or checked out from a Subversion repository. This chapter describes both methods and the use of utility scripts that can help getting only the necessary source code components.

If you are interested in downloading source code from the C Toolkit instead of the C++ Toolkit, please see Access to the C Toolkit source tree Using CVS.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Public Access to the Source Code via FTP

- FTP Download Now

- **Available FTP Archives**: Select the archive for your system. When the dialog box appears, choose the destination in your file system for the downloaded archive. Note: With some browsers, you may need to right-click-and-hold with your mouse and use the 'Save Link As...', 'Copy to Folder...', or similar options from the drop-down menu to properly save the archive. For a current list of the source code archives for different operating system/compiler combinations consult the current Release Notes available at ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/CURRENT/RELEASE_NOTES.html
- **Unpack the Source Archive**
  - *Unix and Macintosh Systems*
    The Unix distributions have been archived using the standard tar command and compressed using gzip. When unpacked, all files will be under the directory ncbi_cxx--<version_number>, which will be created in the current directory. (Caution: If ncbi_cxx--<version_number> already exists, tar extraction will overwrite existing files.) To unpack the archive: gunzip -c ncbi_cxx--*.tar.gz | tar xvf -
  - *Windows Systems*
    The Microsoft Windows versions of the source distribution have been prepared as self-extracting executables. By default a sub-folder ncbi_cxx--<version_number > will be created in the current folder to contain the extracted source. If ncbi_cxx--<version_number > already exists in the folder where the executable is launched, user confirmation is required before files are overwritten. To actually perform the extraction, do one of the following:
    - Run the executable from a command shell. This will create the sub-folder in the shell's current directory, even if the executable is located somewhere else.
    - Double-click on the archive's icon to create ncbi_cxx--<version_number > in the current folder.
    - Right-click on the archive's icon, and select 'Extract to...' to unpack the archive to a user-specified location in the filesystem.

Read-Only Access to the Source Code via Subversion

The following options for read-only access to the C++ Toolkit Subversion repository are available to the public:

- Checking out the source tree directly from the repository (e.g. svn co http://anonsvn.ncbi.nlm.nih.gov/repos/v1/trunk/c++).
- Browsing the repository with an HTTP browser (e.g. http://www.ncbi.nlm.nih.gov/viewvc/v1/trunk/c++).
- Accessing the repository with a WebDAV client (also using http://anonsvn.ncbi.nlm.nih.gov/repos/v1/trunk/c++ – although some clients may require dav:// instead of http://).

Read-Write Access to the Source Code via Subversion (NCBI only)

Note: This section discusses read-write access to the Subversion repository, which is only available to users inside NCBI. For public access, see the section on read-only access.

Subversion client installation and usage instructions are available on separate pages for UNIX, MS Windows, and Mac OS systems.

For a detailed description of the Subversion Version Control System please download the book "Version Control with Subversion" or run the command svn help on your workstation for quick reference.

The following is an outline of the topics presented in this section. Select the instructions appropriate for your **development** environment.

NCBI Source Tree Contents

The NCBI C++ Toolkit Subversion repository contains all source code, scripts, utilities, tools, tests and documentation required to build the Toolkit on the major classes of operating systems (Unix, MS Windows and Mac OS).

Source Code Retrieval under Unix

Retrieval of the C++ Toolkit Source Code Tree

This section discusses the methods of checking out the entire source tree or just the necessary portions. An important point to note is that the entire NCBI C++ tree is very big because it contains a lot of internal projects. There are also numerous platform-specific files, and even platform-specific sub-trees, which you will never need unless you work on those platforms. Therefore it is frequently sufficient, and in fact, usually advisable, to retrieve only files of interest using the shell scripts from the path (it is in the default $PATH):

/am/ncbiapdata/bin

They can also be checked out directly from the Subversion repository at:

https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/internal/scripts/common

The auxiliary script svn_core checks out only the core NCBI C++ Toolkit sources for a desired platform. A similar auxiliary script, import_project, can be used to import the source from a single project. To facilitate the creation of a new project, use the script new_project which generates new directories and makefiles for the new project from templates. This script also checks out a specified sample application from the source tree that may be adapted for the new project or built directly as a demonstration.

Checking out the whole Toolkit source tree using a Subversion client can take 15 minutes or more. However, the script svn_toolkit_tree (available to NCBI users via the default PATH on most UNIX hosts) produces the same result in only 10-30 seconds. The svn_toolkit_tree script combines a daily archive with an update of the working copy to bring it up-to-date. This produces the same set of files and revisions as running svn checkout, but in much less time. Besides speed, the differences between using a Subversion client and the svn_toolkit_tree script include:

- The svn_toolkit_tree script may not be compatible with your Subversion client. If your client is older than the version used to create the archive, you may not be able to access the archive.
- The svn_toolkit_tree script requires that your current directory does not contain a subdirectory with the name that the script is about to create (see below for the subdirectory names created by the script).

There are three archives currently available:

- trunk
- trunk-core
- production

which correspond to the following flavors of the C++ Toolkit trees:

- https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/internal/c++
- https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/c++
- https://svn.ncbi.nlm.nih.gov/repos/toolkit/production/candidates/production.HEAD/c++

which the script will deploy to the local subdirectory named, respectively:

- toolkit-trunk/
- toolkit-trunk-core/
- toolkit-production/

For example, to retrieve the current TRUNK version of the "core" part of the C++ Toolkit tree (the part without the GUI and INTERNAL projects), run:

```
$ svn_toolkit_tree trunk-core
/net/snowman/vol/projects/ncbisoft/toolkit_trees/trunk-core.tar.gz ->
toolkit-trunk-core/
Updating toolkit-trunk-core/...

$ ls toolkit-trunk-core/
compilers configure doc include scripts src
```

Checking Out the Development NCBI C++ Toolkit Source Tree

You can check out the entire development NCBI C++ source tree from the repository to your local directory (e.g., foo/c++/) just by running:

```
cd foo
svn checkout https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/c++
```

For internal projects use:

```
cd foo
svn checkout https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/internal/c++
```

Caution: Be aware that sources checked out through the development source tree have the latest sources and are different from the public release that is done at periodic intervals. These sources are relatively unstable "development" sources, so they are not guaranteed to work properly or even compile. Use these sources at your own risk (and/or to apply patches to stable releases).The sources are usually better by the end of day and especially by the end of the week (like Sunday evening).

Checking Out the Production NCBI C++ Toolkit Source Tree

Besides the development NCBI C++ source tree, there is the C++ Toolkit "production" source tree that has been added to the public Subversion repository. This tree contains stable snapshots of the "development" C++ Toolkit tree. Please note that these sources are lagging behind, sometimes months behind the current snapshot of the sources.

You can check out the entire "production" NCBI C++ source tree from the public repository to your local directory by running:

```
svn co https://svn.ncbi.nlm.nih.gov/repos/toolkit/production/latest/c++
```

This repository path corresponds to the latest production build of the Toolkit. If you want to check out sources for an older production build, please specify the exact date of that build as follows:

```
svn co https://svn.ncbi.nlm.nih.gov/repos/toolkit/production/20031212/c++
```

where 20031212 is the date when this specific build was originated. You can easily find out the available production builds by running

```
svn ls https://svn.ncbi.nlm.nih.gov/repos/toolkit/production
```

This command will print directories under production/, which correspond to the production builds.

svn_core: Retrieving core components

The NCBI C++ Toolkit has many features and extensions beyond the core of portable functionality. However, one often wants to obtain a set of core sources that is free of non-portable elements, and the svn_core script performs this task across the range of supported platforms. Options to the basic command allow the developer to further tailor the retrieved sources by including (or excluding) certain portions of the Toolkit.

For usage help, run svn_core without arguments.

Note: svn_core is not available on Windows.

Table 1 describes the arguments of svn_core. Only the target directory and SVN branch arguments are mandatory.

Some directories are always checked out, regardless of command-line arguments. These are shown in Table 2. (All paths are relative to the repository path https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/c++.)

Other directories may or may not be checked out, depending on the <branch> and <platform> options. These are shown in Table 3.

Still other directories may be checked out depending on the --with/--without-<feature> options. These are shown in Table 4.

import_project: Retrieve Source for an Existing Project

Usage:

```
import_project <SVN_relative_tree_path> [builddir]
```

In many cases, you work on your own project which is a part of the NCBI C++ tree, and you do not want to check out, update and rebuild the entire NCBI C++ tree. Instead, you just want to use headers and libraries of the pre-built NCBI C++ Toolkit to build your project.

The shell script import_project will check out your project's src and include directories from the repository and create temporary makefiles based on the project's customized makefiles. The new makefiles will also contain a reference to the pre-built NCBI C++ Toolkit.

For example:

```
import_project serial/datatool
```

will check out the datatool project from the NCBI C++ tree (trunk/c++/{src,include}/serial/datatool/), and create a makefile Makefile.datatool_app that uses the project's customized makefile Makefile.datatool.app. Now you can just go to the created working directory c++/src/serial/datatool/ and build the application datatool using:

```
make -f Makefile.datatool_app
```

update_core: Update the Portable and Core Components

Usage:

```
update_core [--no-projects] [<dirs>]
```

Once you have obtained the core C++ Toolkit sources, with svn_core or otherwise, the local copies will become out of sync with the master SVN repository contents when other developers commit their changes. update_core will update your local core source tree with any changed files without the side-effect of simultaneously checking out non-core portions of the tree. Subdirectory */internal does not get updated by this script.

The --no-projects switch excludes any Windows or MacOS project files from the update. Specifically, those subdirectory names of the form *_prj are skipped during the update when this flag is set.

The list [<dirs>], when present, identifies the set of directories relative to the current directory to update. The default list of updated directories is:

- .
- compilers
- doc
- include
- scripts
- src

Note that the default list is not pushed onto a user-supplied list of directories.

update_projects: Check out and update Source of Selected Projects

Usage:

```
update_projects <project-list> [<directory>]
```

Script update_projects facilitates the original retrieval and subsequent updates of selected parts of the Toolkit tree. Because the source code and makefiles are distributed over more than one subdirectory under repository path trunk/c++, this script assembles the set of required files and places them in your local C++ source tree.

The projects to be retrieved (or updated) must be specified in the command line as the <project-list> parameter. Its value can be either of the following:

- Explicit specification of the pathname of the project listing file. This project listing file can contain project directory names as well as references to other project listings and must be formatted according to the simple syntax used by the configure script.
- Specify one of the standard project names. Standard projects are those whose project listing files are located in one of the system directories, which are trunk/c++/scripts/projects and trunk/c++/scripts/internal/projects. When a project name is specified on the command line, the ".lst" extension is added to it and the resulting file name is searched for in the above mentioned system directories.

The parameter to update_projects indicates the target directory where the sources will be checked out to and where the project will be configured and built. This parameter is optional and is set to the current directory by default.

Source Code Retrieval under MS Windows

1. In NCBI, the SVN clients must be set up and ready to use. Ask Systems if you don't have the client installed on your workstation. If you are working outside of NCBI, then you can download the latest version of Subversion from http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91. Run the Subversion installer and follow the instructions. The latest version may not come with an executable installer though. In this case, please unpack the zip archive with the latest Subversion binaries to a local directory, for example C:\Program Files\svn-win32-1.4.2. Change the PATH environment variable so that it points to the bin subdirectory under your Subversion installation directory, for example set PATH=%PATH%;C:\Program Files\svn-win32-1.4.2\bin
2. Start your favorite command shell. Change current directory to the designated working directory. At the command prompt, type:svn co https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/c++
3. Modify source files as required. Refer to Svnbook for the documentation on particular Subversion commands. Monitor your changes using svn diff, synchronize your working copy with the trunk using svn update, and finally commit them using svn commit.

The rest should be the same as when using Subversion under UNIX systems. See Source Code Retrieval under Unix.

Source Code Retrieval under Mac OS X

Download and install the latest Subversion binaries for MacOSX from http://subversion.tigris.org/.

The rest should be the same as when using Subversion under UNIX systems. See Source Code Retrieval under Unix.

Source Tree Structure Summary

To summarize the Getting Started page, the source tree is organized as follows:

- The top-level has configuration files and the directories include/, src/, scripts/, compilers/ and doc/
- The src and include directories contain "projects" as subdirectories. Projects may contain sub-projects in a hierarchical fashion.
- src/ additionally contains makefile and meta-makefile templates.
- Projects contain "modules" and various customized makefiles and meta-makefiles to control their compilation.

Table 1. svn_core Arguments

Argument	Description	Permitted Values
<dir>	Path to where the source tree will be checked out. This argument is **required**.	A valid writable directory name (must not exist already); name cannot start with "-".
<branch>	Which branch of the source tree to check out. This argument is **r equired**.	core - toolkit/trunk/c++ development - toolkit/trunk/internal/c++ production - toolkit/production/candidates/trial/c++ prod-head - toolkit/production/candidates/production.HEAD/c++ frozen-head - toolkit/production/candidates/frozen.HEAD/c++ trial - toolkit/production/candidates/trial/c++ release - toolkit/release/public/current/c++ gbench - gbench/branches/1.1 gbench2 - gbench/trunk (See c++-branches.txt for an up-to-date list of branches.)
--date	Check out as at the start of the specified timestamp. If the --date flag is missing, today's date and current time are used.	A date in a format acceptable to the svn -r argument, for example --date="2013-03-29 19:49:48 +0000". (Do not include curly braces and quote the timestamp if it contains spaces.) See the Revision Dates section in the Subversion manual for details.
--export	Get a clean source tree without .svn directories.	n/a
--<platform>	Obtain sources for the specified platform(s).	--unix - Unix systems; --msvc - Microsoft Visual C++ environment; --mac - Macintosh systems; --cygwin - Cygwin UNIX environment for Windows; --all - all platforms. If no value is supplied, --all is used.
--with-ctools	Check out core projects responsible for working together with the NCBI C Toolkit (the ctools directory). This option is effective by default unless --without-ctools is used.	n/a
--without-ctools	Do not check out core projects responsible for working together with the NCBI C Toolkit (the ctools directory).	n/a
--with-gui	Check out core projects responsible for providing cross-platform graphic user interface capability (the gui directory). This option is effective by default unless --without-gui is used.	n/a
--without-gui	No not check out core projects responsible for providing cross-platform graphic user interface capability (the gui directory).	n/a
--with-internal	Check out a selection of NCBI-internal core projects. See Table 4 for a detailed list of affected directories.	n/a
--without-internal	Do not check out NCBI-internal core projects.	n/a

--with-objects	Check out the objects, objmgr, and objtools directories and generate serialization code from the ASN.1 specifications. If this flag is not present, those directories are still checked out (unless overridden by the --without-objects flag) but no serialization code is generated.	n/a
--without-objects	Do not check out the objects, objmgr, and objtools directories or generate ASN.1 serialization code. (On Unix platforms the code generation can be done later, during the build.)	n/a

Table 2. List of the directories that are always checked out

Checked out directories	**Recursive?**
(include|src)	no
(include|src)/algo	yes
src/app	yes
src/build-system	yes
(include|src)/cgi	yes
include/common	yes
(include|src)/connect	no
(include|src)/connect/ext	yes
include/connect/impl	yes
src/connect/test	yes
(include|src)/connect/services	yes
(include|src)/corelib	yes
(include|src)/db	yes
(include|src)/dbapi	yes
(include|src)/html	yes
(include|src)/misc	yes
(include|src)/sample	yes
(include|src)/serial	yes
include/test	yes
(include|src)/util	yes
scripts	yes

Table 3. Directories that may be checked out depending on branch and platform options

Checked out directories	**Recursive?**	**Options**
compilers	yes	<platform> = all
compilers	no	<platform> != all
compilers/cygwin	yes	<platform> = cygwin
compilers/msvc1000_prj	yes	<platform> = msvc
compilers/unix	yes	<platform> = cygwin or mac or unix
compilers/xCode	yes	<platform> = max
compilers/xcode90_prj	yes	<platform> = mac
doc	yes	<branch> = development
include/connect/daemons	yes	<platform> = all or unix
src/check	yes	<platform> != mac
src/connect/daemons	yes	<platform> = all or unix
src/connect/mitsock	yes	<platform> = mac
src/dll	yes	<platform> = all or mac or msvc

Table 4. Directories that may be checked out depending on --with/--without options

Checked out directories	**Recursive?**	**Options**
(include|src)/ctools	yes	--with-ctools or not --without-ctools
(include|src)/gui	yes	--with-gui or not --without-gui
(include|src)/internal	no	--with-internal
(include|src)/internal/algo	no	--with-internal
(include|src)/internal/algo/id_mapper	yes	--with-internal
(include|src)/internal/align_model	yes	--with-internal
include/internal/asn_cache	yes	--with-internal
src/internal/asn_cache	no	--with-internal
src/internal/asn_cache/lib	yes	--with-internal
(include|src)/internal/blast	no	--with-internal
(include|src)/internal/blast/DistribDbSupport	yes	--with-internal
(include|src)/internal/contigdb	no	--with-internal
src/internal/demo	yes	--with-internal
(include|src)/internal/ID	no	--with-internal
(include|src)/internal/ID/utils	no	--with-internal
(include|src)/internal/mapview	no	--with-internal
(include|src)/internal/mapview/objects	yes	--with-internal
(include|src)/internal/mapview/util	yes	--with-internal
(include|src)/internal/myncbi	yes	--with-internal
include/internal/objects	no	--with-internal
(include|src)/objects	yes	--with-objects or not --without-objects
(include|src)/objmgr	yes	--with-objects or not --without-objects
(include|src)/objtools	yes	--with-objects or not --without-objects
src/internal/objects	yes	--with-internal
(include|src)/internal/sra	yes	--with-internal
src/internal/test	yes	--with-internal
(include|src)/internal/txclient	yes	--with-internal
(include|src)/internal/txserver	yes	--with-internal
(include|src)/internal/txxmldoc	yes	--with-internal

4: Configure, Build, and Use the Toolkit

Last Update: July 18, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter describes in detail how to configure, build, and use the NCBI C++ Toolkit (or selected components of it) on supported platforms. See the Getting Started chapter for a general overview of the process. A list of all supported platforms can be seen here.

Note: Users insde NCBI who just want to use the Toolkit don't need to configure and build it - there are various configurations of the Toolkit prebuilt and ready to use. See the new_project script for more information.

Configuring is the process of creating configuration files that define exactly what can be built and what options may be used in the build process. The created configuration files include C headers that define suitable preprocessor macros, as well makefiles (for UNIX) or project solutions (for MS Visual C++ or for Xcode) used in the build step.

With some compilers that include an Integrated Development Environment (e.g. MS Visual C++), a top-level build target, called CONFIGURE, is available. On UNIX-like systems it is necessary to execute a configuration script *configure* – sometimes via a special wrapper script that first performs some platform-specific pre-configuration steps and then runs the configuration process.

The configuration process defines the set of targets that can be built. It is up to the user to choose which of those targets to build and to choose the desired build options. For more details on the build system and the Makefiles created by the configuration process, see the chapter on Working with Makefiles.

Successful builds result in immediately usable libraries and applications, and generally there is no need for a separate installation step on any platform.

In addition to building the Toolkit libraries and applications, this chapter also discusses building test suites and sample applications. You might want to build and run a test suite if you are having trouble using the Toolkit and you aren't sure if it is working properly. While it isn't necessary to build a test suite to use the Toolkit, it can be useful for ensuring that the Toolkit has been properly configured and built. Building a sample application may be a good first step toward learning how to build your own applications.

Chapter Outline

General Information for All Platforms

Using the Toolkit on any platform requires these basic high-level steps:

- Prepare the development environment.
- Get the source files from NCBI and place them in your working directory.
- Choose a build scope.
- Configure the build.
- Build.
- Use the Toolkit from your application.

Choosing a Build Scope

After preparing the development environment, you'll need to choose a build scope. Choosing a build scope means deciding whether you want to build the entire Toolkit or just some portion of it. The build system includes methods on most platforms for building pre-defined scopes, such as just the core libraries and applications, the Genome Workbench, pre-defined lists of one or more projects, etc. Choosing a build scope must be done before configuring on some platforms. On other platforms it can be done either before or after configuring. See the section for your platform for more details on pre-defined build scope choices.

Reducing Build Scope with Project Tags

The pre-defined build scopes mentioned above may be unnecessarily broad for your task. You can reduce the build scope by using project tags.

There are two complementary parts to using project tags. First, project tags are defined and associated with selected projects. Second, a tag filter is supplied to the configuration process. The configuration process then filters the list of projects that will be built, based on each project's tags and the supplied tag filter.

An important benefit of using project tags is that all dependencies for the projects that match the tag filter will be automatically deduced and added to the build list.

Defining Project Tags

All project tags must be defined in src\build-system\project_tags.txt prior to use. Tag names should be easily recognizable and classifiable, like ‘proj[_subproj]’, e.g. “pubchem” or “pubchem_openeye”.

Once defined in project_tags.txt, project tags can then be associated with any number of projects by using the PROJ_TAG macro in the Makefile.in or Makefile.*.{app|lib} for the selected projects. Project tag definitions apply recursively to subprojects and subdirectories (similar to a REQUIRES definition), thereby removing the need to define tags in all makefiles in a subtree. Subprojects may define additional tags, or undefine inherited tags by prefixing a hyphen '-' to the tag.

The syntax for defining (or undefining) a project tag is:

```
PROJ_TAG = [-]mytag1 [[-]mytag2...]
```

For example, if Makefile.in has this line:

```
PROJ_TAG = foo bar
```

and a project beneath it in the tree hierarchy (say Makefile.*.app) has this line:

```
PROJ_TAG = xyz -bar
```

then the latter project's effective tag definition is:

```
PROJ_TAG = foo xyz
```

Filtering with Project Tags

A tag filter can be constructed from one or more project tags – either as a single tag or as a Boolean expression of tags. Boolean expressions of tags can include grouping (parentheses) and the '&&' (AND), '||' (OR), and '!' (NOT) operators, for example: (core || web) && !test

Note: An asterisk '*' or an empty string can be used in place of a tag filter in the "Allowed project tags" field on the Configuration tab of the configuration GUI. These values are not filters, but simply indicate that all projects in the build scope will be passed to the configuration process without filtering.

The following places are searched in the order given for the tag filter to use (if any) in the configuration process:

1. The "Allowed project tags" field in the configuration GUI (if the configuration GUI is being used).
2. A tag filter definition line in a project list file (if one is being used).
   a. To use a project list file for configuration, either specify the project list file in the "Subtree, or LST file" field on the Configuration tab of the configuration GUI or use the --with-projects=FILE argument for the configure script.
   b. When one project list file includes another, only the original will be scanned for a filter. This applies to both interactive (i.e. with the configuration GUI) and non-interactive configuring.
   c. The syntax for the tag filter definition line in a project list file is: #define TAGS [tag_filter]
3. For MSVC, the -projtag option of the PTB_FLAGS macro in the compilers \msvc1000_prj\static\build\UtilityProjects\configure._ file for non-interactive configuring, or the same option in the configure_dialog._ file for interactive configuring.

If a significant tag filter (i.e. something besides an asterisk or empty field) is found in one of the above places, then that tag filter will be supplied to the configuration process. Otherwise, there will be no filtering of the projects.

Configure the Build

Prior to configuring, users outside NCBI should make sure the paths to their third party libraries are correctly specified.

For the configuration step you can specify whether to use static or dynamically-linked libraries; whether to generate multithread-safe code; whether to look for various third-party libraries at alternative locations; whether or not to include debugging information; etc.

Configuration can be done in one of three ways:

- Using the Configuration GUI.
- Using a "native" IDE – MSVC on Windows or Xcode on Mac OS X.
- Using the command-line on UNIX, Cygwin/Windows, or Mac OS X.

Site-Specific Third Party Library Configuration

Users outside NCBI should check the file src/build-system/config.site to see if it correctly specifies the paths to their third party libraries. If not, it can be edited using src/build-system/config.site.ex as a guide.

Note: The configure --with-PACKAGE options take precedence over the config.site and PACKAGE_PATH settings.

Using the Configuration GUI

The configuration GUI can be launched from a command shell or from an IDE (MSVC or Xcode). It is Java-based and requires the Java Platform Standard Edition.

The following sections describe how to use the configuration GUI:

- Starting the configuration GUI
- Configuration tab
- Advanced tab
- Third party libraries tab
- Projects tab
- Done tab

See the UNIX, Windows, and Mac OS X sections for OS-specific configuration information.

Starting the configuration GUI

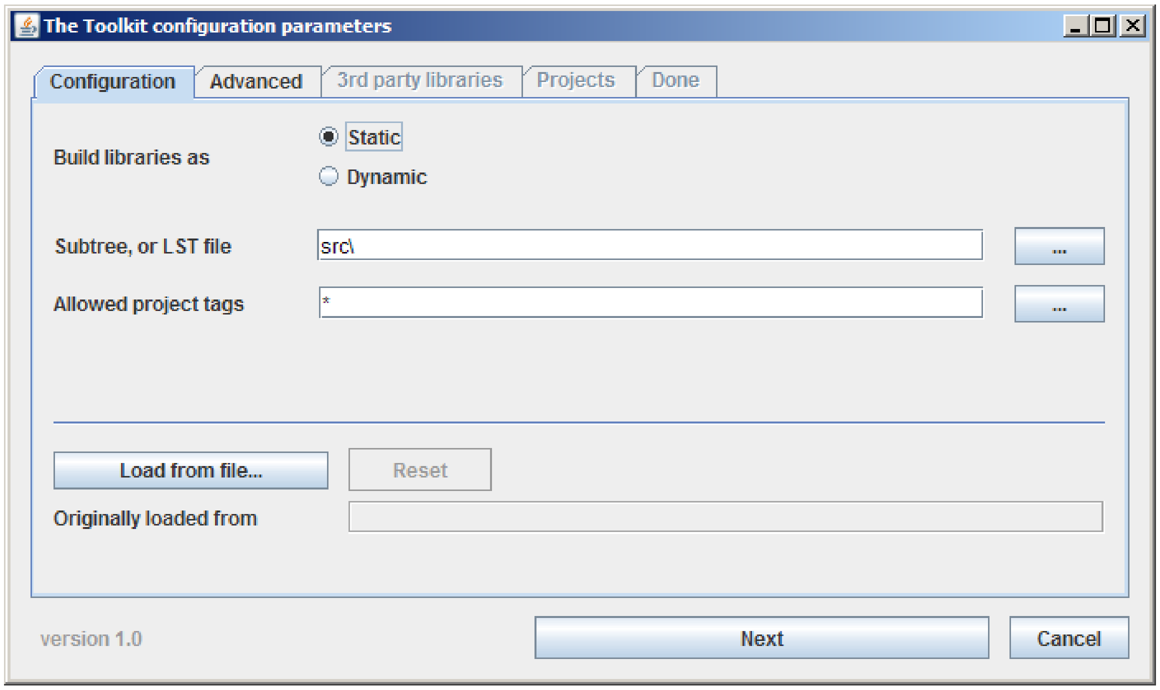
To launch the configuration GUI:

- From the command-line: ./configure --with-configure-dialog
- From the MSVS IDE: build the -CONFIGURE-DIALOG- project
- From the Xcode IDE: build the CONFIGURE-DIALOG target

The configuration GUI has a "Wizard" style design – selections are made in a sequence of steps, followed by clicking the Next button. After each step additional tabs may be enabled, depending on the specific data. It opens with initial values set by the invoking program (the configure script for command-line invocation or the project_tree_builder program for IDE's).

Configuration tab

The Configuration tab looks like:



Build libraries as: [✓ Static] Dynamic

Subtree, or LST file: src\

Allowed project tags: *

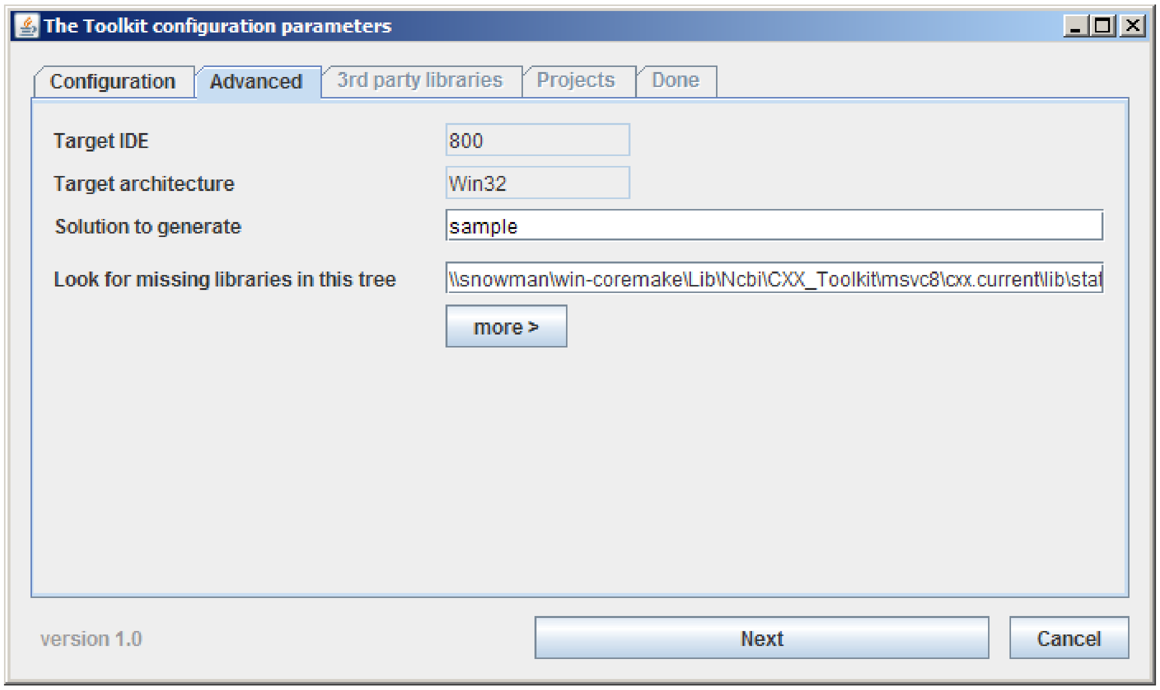
Originally loaded from:

The Configuration tab allows you to:

- Choose between static and dynamically-linked libraries.
- Specify the subset of the Toolkit that you want to build, using either a path for a subtree (e.g. src\) or a project list file (*.lst) for specific projects. Clicking on the "..." button opens a file selection dialog, which can be used to navigate to the desired subtree or to select a project list file.
- Specify one or more project tags (which will restrict the scope of the build to the specified projects). Clicking on the "..." button simply displays the valid choices for project tags (it isn't used for selecting tags). More than one project tag can be combined in a Boolean expression, for example:
  (code || web) && !test
- Load a configuration from a file. This requires having previously saved a configuration, from the Done tab. If you load a configuration from a file, the file path is shown in the "Originally loaded from" text field and the Reset button becomes enabled. Clicking the Reset button resets all configuration settings to the values that were used to invoke the configuration GUI.

Advanced tab

The Advanced tab looks like:

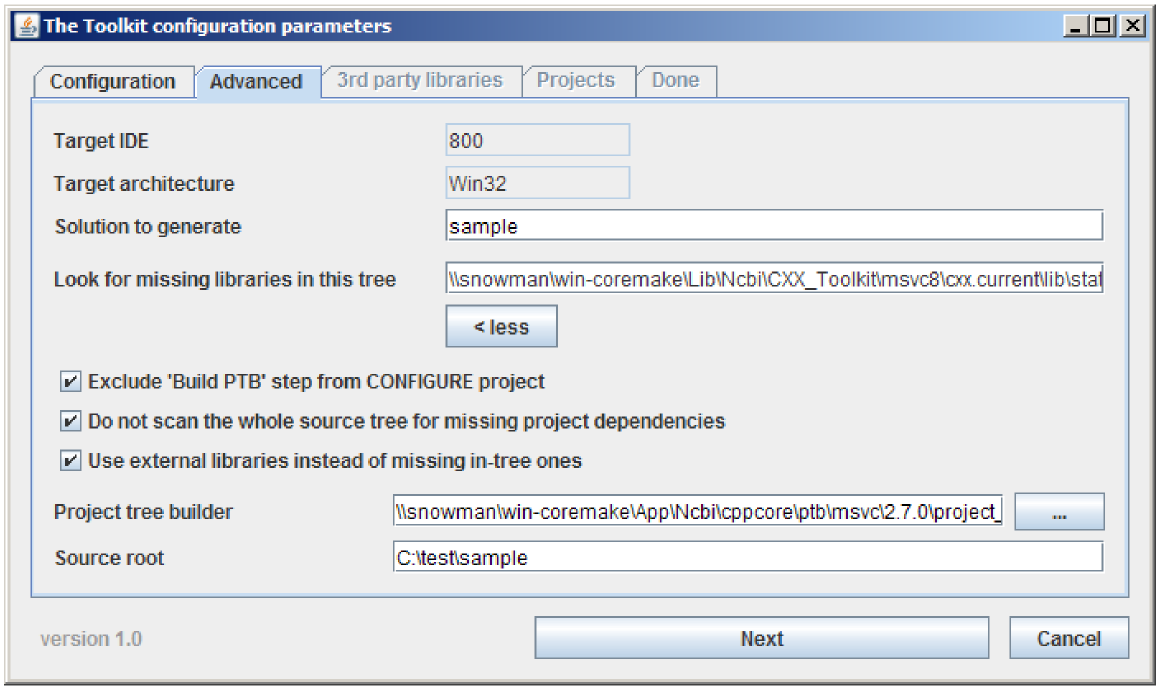


Target IDE: 800
Target architecture: Win32
Solution to generate: sample
Look for missing libraries in this tree: \\snowman\win-coremake\Lib\Ncbi\CXX_Toolkit\msvc8\cxx.current\lib\stat

The Advanced tab allows you to:

- View the current version of the IDE (currently only applicable to Windows / Microsoft Visual Studio).
- View the current architecture (currently only applicable to Windows / Microsoft Visual Studio).
- Specify the name of a solution file to generate. You can use this to create different solution files for different configurations.
- Specify where to look for missing libraries. This can be used to change the build – for example, from cxx.current to cxx.potluck.

In addition, by clicking "more" you will see:



Target IDE: 800

Target architecture: Win32

Solution to generate: sample

Look for missing libraries in this tree: \\snowman\win-coremake\Lib\Ncbi\CXX_Toolkit\msvc8\cxx.current\lib\stat

☑ Exclude 'Build PTB' step from CONFIGURE project

☑ Do not scan the whole source tree for missing project dependencies

☑ Use external libraries instead of missing in-tree ones

Project tree builder: \\snowman\win-coremake\App\Ncbi\cppcore\ptb\msvc\2.7.0\project_

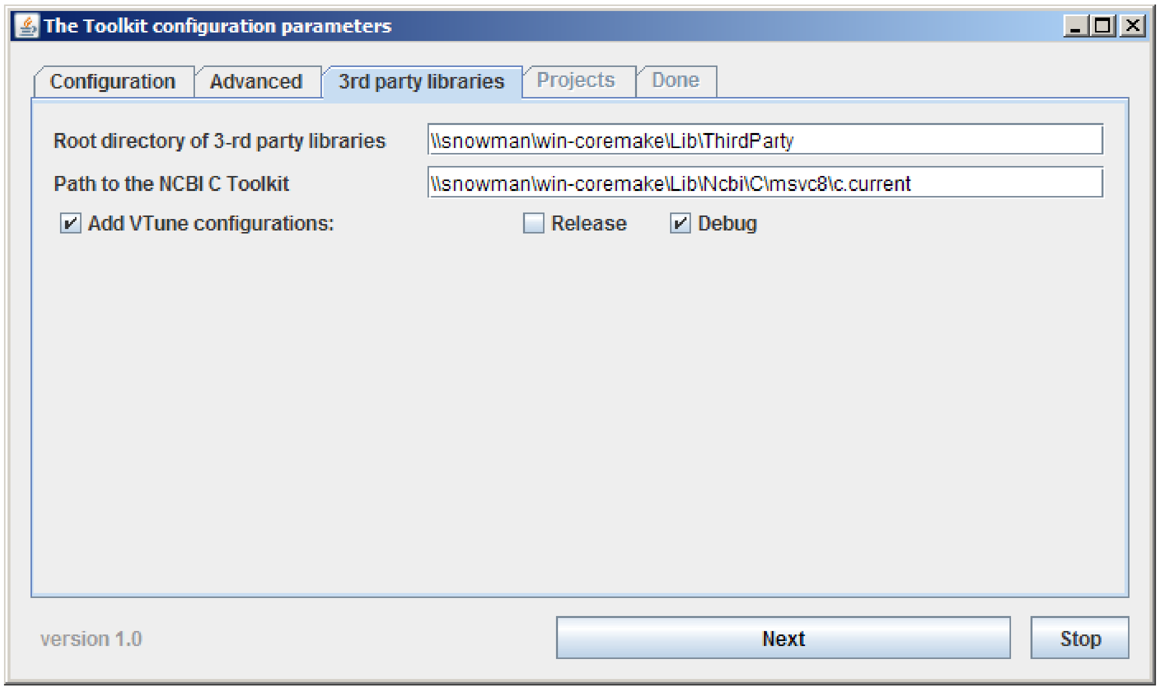
Source root: C:\test\sample

These additional options generally don't need to be changed, but they allow you to:

- Exclude the "Build PTB" step from the configure process. This should be selected if the PTB (project tree builder) source is not available. Even if the PTB source is available, it usually makes sense to exclude building the PTB because building it will take longer and generally won't have a benefit.
- Prevent whole-tree scanning for missing project dependencies. A project dependency may be missing if, for example, import_project was used and the configuration was changed to something other than simply Debug or Release (e.g. DebugMT).
- Use external libraries instead of missing in-tree ones.
- Select a different project tree builder. In most cases this won't be needed, but it could be useful for tasks such as debugging the build system.
- Select a different location to use as the root of the source tree.

Third party libraries tab

The Third party libraries tab looks like:



Root directory of 3-rd party libraries: \\snowman\win-coremake\Lib\ThirdParty

Path to the NCBI C Toolkit: \\snowman\win-coremake\Lib\Ncbi\C\msvc8\c.current

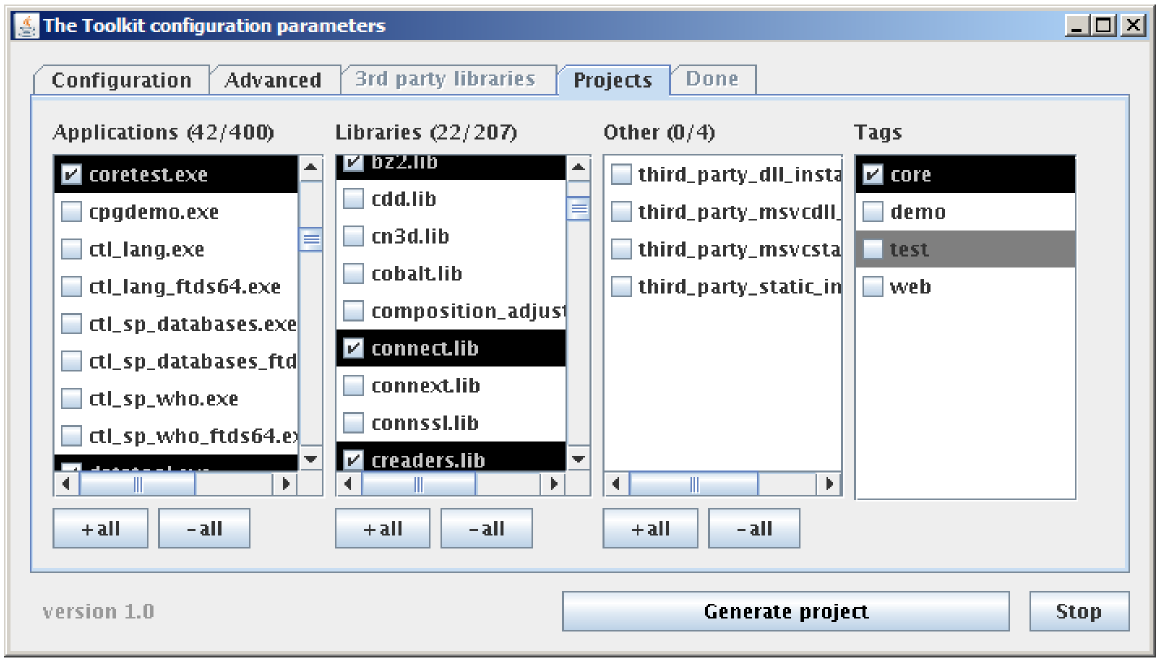
[✓] Add VTune configurations: [ ] Release [✓] Debug

The Third party libraries tab allows you to:

- Select a different location for third-party libraries.
- Select a different location for the NCBI C Toolkit.
- Add VTune configurations. If selected, new VTune configurations will be added to the list of available configurations – for example, VTune_DebugDLL.

Projects tab

The Projects tab looks like:



Applications (42/400): [✓] coretest.exe, [ ] cpgdemo.exe, [ ] ctl_lang.exe, [ ] ctl_lang_ftds64.exe, [ ] ctl_sp_databases.exe, [ ] ctl_sp_databases_ftd, [ ] ctl_sp_who.exe, [ ] ctl_sp_who_ftds64.e

Libraries (22/207): [✓] bz2.lib, [ ] cdd.lib, [ ] cn3d.lib, [ ] cobalt.lib, [ ] composition_adjust, [✓] connect.lib, [ ] connext.lib, [ ] connssl.lib, [✓] creaders.lib

Other (0/4): [ ] third_party_dll_insta, [ ] third_party_msvcdll_, [ ] third_party_msvcsta, [ ] third_party_static_in

Tags: [✓] core, [ ] demo, [ ] test, [ ] web

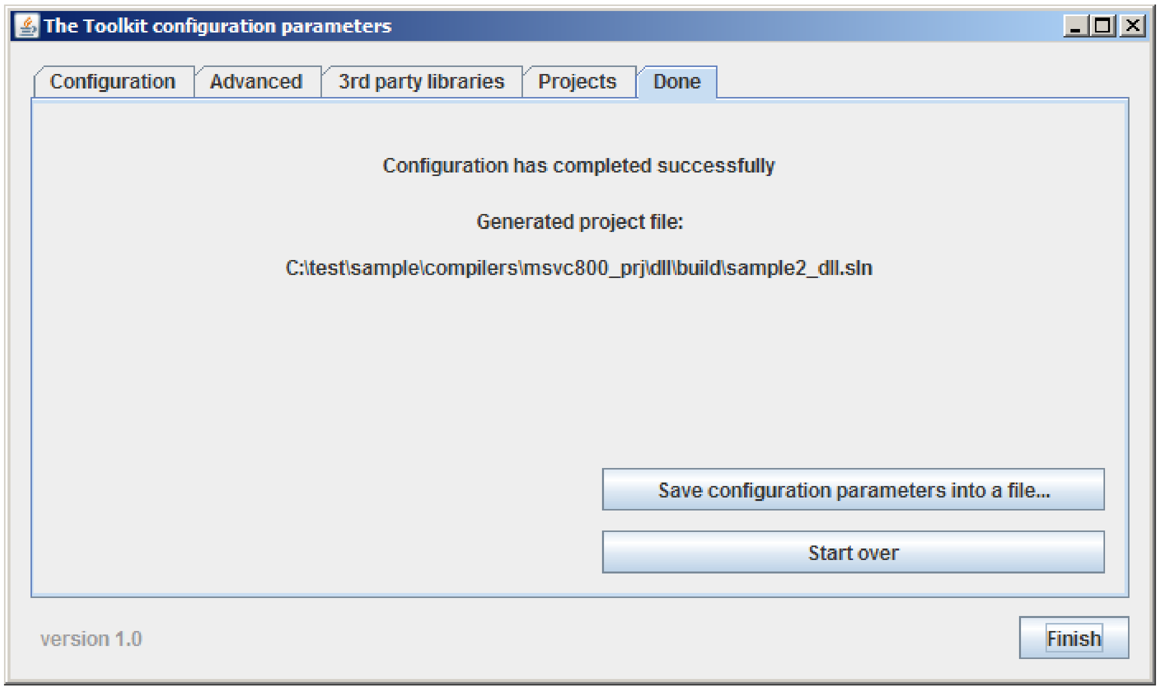
The Projects tab allows you to select exactly which applications and libraries will be built. If an item is not selected, but at least one selected item depends on it, then it will also be built. This provides a convenient way for developers to simply pick the top-level items to build.

The "-all" and "+all" buttons uncheck or check all the items in a column.

The Tags column allows you to quickly select all items having the selected project tag(s). Also, selecting items in the other columns will update the selection status of the tags column.

Done tab

The Done tab looks like:


The Toolkit configuration parameters
Configuration
Advanced
3rd party libraries
Projects
Done
Save configuration parameters into a file...
Start over
version 1.0
Finish

Configuration has completed successfully

Generated project file:

C:\test\sample\compilers\msvc800_prj\dll\build\sample2_dll.sln

The Done tab:

- Reports whether the project was generated successfully.
- Shows the path for the generated solution file.
- Gives the option to save the configuration parameters. Once saved, the same parameters can be loaded again from the Configuration tab.
- Gives the option to start over and create a new set of configuration parameters.
- Gives the option to close the tool, via the Finish button. Closing the tool will return you to the configuration process, which will continue based on the parameters set in the configuration GUI.

Use the Toolkit

After choosing a build scope, configuring, and building the Toolkit, you can now use it. The Toolkit itself includes useful applications, demo programs, and sample code – in addition to the libraries you can use from your own applications. You can also build a suite of test applications and/or sample applications if desired.

Supported Platforms

The term “platform” in this chapter has a specific meaning: the combination of operating system, architecture, and compiler. A supported platform is one for which the Toolkit has been configured, built, tested, and used by other applications.

The list of supported platforms may change with new releases. For the platforms supported in the release you are using, see the Supported Platforms section in the release notes. Note that some platforms are only partially supported.

UNIX

Note: Please also see the General Information for All Platforms section, as it contains relevant information that is not repeated here.

This section covers the following topics:

- General Information for UNIX Platforms
  - Choosing a Build Scope
  - Configuring
  - Building
  - Using
- Special Considerations for Specific UNIX Platforms
  - Linux / ICC
  - Cygwin / GCC

General Information for UNIX Platforms

This section provides information on configuring, building, and using the Toolkit that is applicable to all UNIX platforms. The section Special Considerations for Specific UNIX Platforms addresses platform-specific details.

Note, however, that the sections on specific platforms do not address the level of support for specific compilers. See the Supported Platforms section in the release notes for information on partially supported compilers.

The following topics are discussed in this section:

- Choosing a Build Scope
- Configuring
  - Configuration Script configure
  - Structure of the Build Tree Produced by configure
  - Options for Fine-Tuning the configure Script
  - Quick Reconfiguration
- Building
  - General Principles for Building with UNIX
  - Building Only Core Libraries and Applications
  - Building GUI Libraries and Applications
  - Building the Genome Workbench
  - Building the Entire Toolkit

Choosing a Build Scope with UNIX

The Toolkit is very large and you may not want to retrieve and build the entire Toolkit if you don't need to. Therefore, after preparing the development environment and getting the source files, you'll need to choose a build scope. Several mechanisms are provided to enable working with only a portion of the Toolkit.

The first thing you can do is to limit the source code retrieved from the repository:

- using the shell script import_project; or
- using the shell script update_projects.

Next, you can limit what is built:

- by configuring with the *--with-projects* option; or
- by running *make* only within directories of interest; or
- by building only a selected list of end targets using *flat makefile*

You can also choose between static and shared libraries - or build both. Building with static libraries will result in much larger applications and require much more disk space.

Configuring with UNIX

The following topics are discussed in this section:

Configuration Script configure

Different build setups compile C++ (and even C!) code differently; they may vary in the OS standard and 3rd-party libraries and header files, completeness of the C++ implementation, and in compiler bugs. There are also different versions of *make* and other tools and different file naming conventions on different platforms.

Thus, configuration is needed to use the platform- and compiler-specific features. For this purpose, we are using a script produced by the GNU autoconf utility to automatically generate the build-specific header file ncbiconf.h and makefiles that would work for the given platform.

The user performs configuration by merely running platform-independent (*sh, bash*) shell script ***configure*** (which we pre-generate in-house from the template configure.ac using autoconf).

During the configuration process, many compiler features are tested, and the results of this testing are recorded in the configuration header ncbiconf.h by the means of C preprocessor variables. For example, the preprocessor variable NO_INCLASS_TMPL indicates whether the compiler supports template class methods. Also contained in the ncbiconf.h file are preprocessor variables used to define sized integer and BigScalar types.

The ***configure*** script will create a build tree, a hierarchy of directories where object modules, libraries, and executables are to be built. It will also configure all *.in template files located in the NCBI C++ source tree (src/) and deploy the resultant configured files in the relevant places of the build tree. This way, all platform- and compiler-specific tools and flags will be "frozen" inside the configured makefiles in the build tree. The ncbiconf.h (described above, also configured for the given compiler) will be put to the inc/ sub-directory of the resultant build tree.

You can create as many build trees as needed. All build trees refer to the same source tree, but contain their own platform/compiler-specific ncbiconf.h header and/or different set of compilation/linking flags and tools ("frozen" in the makefiles, particularly in Makefile.mk). This allows building libraries and executables using different compilers and/or flags, yet from the same source, and in a uniform way.

A configuration tool with a Java-based GUI is also available and can be launched from the command-line:

```
./configure --with-configure-dialog
```

Additional parameters can also be passed to configure, just as without the configuration GUI.

For more information on using the configuration GUI, see the general section on configuring.

Structure of the Build Tree Produced by configure

Each configuration process results in a new build tree. The top-level directories in the tree are:

inc/ - contains the ncbiconf.h configuration header generated by the ***configure*** script.

build/ - contains a hierarchy of directories that correspond to those in the src/ (in NCBI C++ original sources). These directories will contain makefiles (Makefile.*) generated by the ***configure*** script from the makefile templates (Makefile.*.in) of the corresponding project located in the source tree. The resultant scripts and makefiles will keep references to the original

NCBI C++ source directories. There is a "very special" file, Makefile.mk, that contains all configured tools, flags, and local paths. This file is usually included by other makefiles. All build results (object modules, libraries, and executables, as well as any auxiliary files and directories created during the build) will go exclusively into the *build tree* and not to the original NCBI C++ source directories. This allows for several build trees to use the same source code while compiling and linking with different flags and/or compilers.

lib/ - contains the libraries built by the build/-located projects.

bin/ - contains the executables built by the build/-located projects.

status/ - contains:

- config.cache, a cache file;
- config.log, a log file;
- config.status, a secondary configuration script produced by ***configure***;
- *.enabled files, with package and feature availability; and
- .*.dep files, with timestamps of the built Toolkit libraries.

Options for Fine-Tuning the configure Script

The configure script is highly customizable. The following sections describe some of the configuration options:

- Getting a Synopsis of Available Configuration Options
- Debug vs. Release Configuration
- Multi-Thread Safe Compilation and Linking with MT Libraries
- Building Shared Libraries (DLLs)
- Finer-grained Control of Projects: --with-projects
- Building in the 64-bit mode
- Localization for the System and Third-Party Packages
- Naming the Build Tree
- Hard-Coding Run-Time DLL Path into Executables and DLLs
- Automatic Generation of Dependencies (for GNU make Only)
- After-Configure User Callback Script
- Tools and Flags
- Prohibiting the Use of Some of the System and Third-party Packages
- Optional Projects
- Miscellaneous: --without-exe, --without-execopy, --with-lib-rebuilds(=ask)

To get the full list of available configuration options, run ./configure --help. The NCBI-specific options are at the end of the printout.

Note: Do not use the "standard" configure options listed in the *"Directory and file names:"* section of the help printout (such as --prefix= , --bindir=, etc.) because these are usually not used by the NCBI ***configure*** script.

The following two ***configure*** flags control whether to target for the *Debug* or *Release* version. These options (the default being *--with-debug*) control the appearance of preprocessor flags -*D_DEBUG* and -*DNDEBUG* and compiler/linker flags -*g* and -*O*, respectively:

--with-debug -- engage -*D_DEBUG* and -*g*, strip -*DNDEBUG* and -*O* (if not *--with-optimization*)

--without-debug -- strip -*D_DEBUG* and -*g*, engage -*DNDEBUG* and -*O* (if not *--without-optimization*)

--with-optimization -- unconditionally engage -*DNDEBUG* and -*O*

--without-optimization -- unconditionally strip -*DNDEBUG* and -*O*

default: *--with-debug --without-optimization*

--with-mt - compile all code in an MT-safe manner; link with the system thread library.

--without-mt - compile with no regard to MT safety.

default: *--without-mt*

On the capable platforms, you can build libraries as *shared (dynamic)*.

--with-dll --with-static -- build libraries as both *dynamic* and *static*; however, if the library project makefile specifies LIB_OR_DLL = lib, then build the library as *static* only, and if the library project makefile specifies LIB_OR_DLL = dll, then build the library as *dynamic* only. Note that the resulting static libraries consist of position-independent objects.

--with-dll -- build libraries as *dynamic*; however, if the library project makefile specifies LIB_OR_DLL = lib, then build the library as *static*

--without-dll -- always build *static* libraries, even if the library project makefile specifies LIB_OR_DLL = dll

default: build libraries as *static* (albeit with position-independent code); however, if the library project makefile specifies LIB_OR_DLL = dll, then build the library as *dynamic*

If the above options aren't specific enough for you, you can also tell ***configure*** which projects you want to build by passing the flag *--with-projects=FILE*, where *FILE* contains a list of extended regular expressions indicating which directories to build in. With this option, the *make* target all_p will build all selected projects under the current directory. If there is a project that you want to keep track of but not automatically build, you can follow its name with "update-only". To **exclude** projects that would otherwise match, list them explicitly with an initial hyphen. (Exclusions can also be regular expressions rather than simple project names.) If no *FILE* argument is supplied then ***configure*** expects to find a project list file named "projects" in the top-level c++ directory.

For instance, a file containing the lines

```
corelib$
util
serial
```

```
-serial/test
test update-only
```

would request a non-recursive build in corelib and a recursive build in util, and a recursive build in serial that skipped serial/test. It would also request keeping the test project up-to-date (for the benefit of the programs in util/test).

Note: The flags listed above still apply; for instance, you still need --*with-internal* to enable internal projects. However, update_projects can automatically take care of these for you; it will also take any lines starting with two hyphens as explicit options.

Project list files may also define a project tag filter, with the syntax:

```
#define TAGS [ tag_filter ]
```

See the section on filtering with project tags for more information.

--*with-64* - compile all code and build executables in 64-bit mode.

default: depends on the platform; usually --*without-64* if both 32-bit and 64-bit build modes are available.

There is some configuration info that usually cannot be guessed or detected automatically, and thus in most cases it must be specified "manually" for the given local host's working environment. This is done by setting the localization environment variables (see Table 2) in addition to the "generic" ones (CC, CXX, CPP, AR, RANLIB, STRIP, CFLAGS, CXXFLAGS, CPPFLAGS, LDFLAGS, LIBS).

On the basis of Table 2, ***configure*** will derive the variables shown in Table 3 to use in the generated makefiles.

Note: The file src/build-system/config.site may also be edited to simplify localization of third party libraries, especially for users outside NCBI.

The configuration process will produce the new *build tree* in a subdirectory of the root source directory. The default base name of this subdirectory will reflect the compiler name and a *Release/Debug* suffix, e.g., *GCC-Release/*. The default *build tree* name can be alternated by passing the following flags to the ***configure*** script:

--*without-suffix* - do not add *Release/Debug*, **MT**, and/or **DLL** suffix(es) to the *build tree* name. **Example:** *GCC/* instead of *GCC-ReleaseMT/*

--*with-hostspec* - add full host specs to the *build tree* name. **Example:** *GCC-Debug--i586-pc-linux-gnu/*

--*with-build-root=/home/foo/bar* - specify your own *build tree* path and name.

With --*with-build-root=*, you still can explicitly use --*with-suffix* and --*with-hostspec* to add suffix(s) to your *build tree* name in a manner described above.

Example: --*with-build-root=/home/foo/bar--with-mt* --*with-suffix* would deploy the new *build tree* in */home/foo/bar-DebugMT*.

There is also a special case with "*--with-build-root=.*" for those who prefer to put object files, libraries, and executables in the same directory as the sources. But be advised that this will not allow you to configure other *build trees*.

To be able to run executables linked against dynamic libraries (DLLs), you have to specify the location (runpath) of the DLLs. It can be done by hard-coding (using linker flags such as-*R.....*) the *runpath* into the executables.

--with-runpath - hard-code the path to the *lib/* dir of the Toolkit *build tree*.

--with-runpath=/foo/bar - hard-code the path to the user-defined */foo/bar* dir.

--without-runpath - do not hard-code any *runpath*.

default: if *--without-dll* flag is specified, then act as if *--without-runpath* was specified; otherwise, engage the *--with-runpath* scenario.

The makefile macro ncbi_runpath will be set to the resulting runpath, if any.

Note: When running an executable you also can use environment variable $LD_LIBRARY_PATH to specify the runpath, like this:

```
env LD_LIBRARY_PATH="/home/USERNAME/c++/WorkShop6-ReleaseDLL/lib" \
/home/USERNAME/c++/WorkShop6-ReleaseDLL/bin/coretest
```

HINT: The *--with-runpath=....* option can be useful to build production DLLs and executables, which are meant to use production DLLs. The latter are usually installed not in the lib/ dir of your development tree (*build tree*) but at some well-known dir of your production site. Thus, you can do the development in a "regular" manner (i.e., in a *build tree* configured using only *--with-runpath*); then, when you want to build a production version (which is to use, let's say, DLLs installed in "/some_path/foo/ "), you must reconfigure your C++ build tree with just the same options as before, plus *"--with-runpath=/some_path/foo"*. Then rebuild the DLLs and executables and install them into production. Then re-reconfigure your *build tree* back with its original flags (without the "*--with-runpath* =/some_path/foo ") and continue with your development cycle, again using local in-tree DLLs.

--with-autodep - add build rules to automatically generate dependencies for the compiled C/C ++ sources.

--without-autodep - do not add these rules.

default: detect if the *make* command actually calls GNU *make*; if it does, then *--with-autodep*, else *--with-autodep*

Also, you can always switch between these two variants "manually", after the configuration is done, by setting the value of the variable Rules in Makefile.mk to either rules or rules_with_autodep.

Note: You **must** use GNU *make* if you configured with *--with-autodep*, because in this case the makefiles would use very specific GNU *make* features!

You can specify your own script to call from the ***configure*** script after the configuration is complete:

--with-extra-action="<some_action>"

where <*some_action*> can be some script with parameters. The trick here is that in the <*some_action*> string, all occurrences of "**{}**" will be replaced by the build dir name.

Example:

```
configure --with-extra-action="echo foobar {}"
```

will execute (after the configuration is done):

```
echo foobar /home/user/c++/GCC-Debug
```

There is a predefined set of tools and flags used in the build process. The user can customize these tools and flags by setting the environment variables shown in Table 1 for the ***configure*** script. For example, if you intend to debug the Toolkit with Insure++, you should run ***configure*** with CC and CXX set to insure.

Later, these tools and flags will be engaged in the makefile build rules, such as:

- To compile C sources: $(CC) -c $(CFLAGS) $(CPPFLAGS)....
- To compile C++ sources: $(CXX) -c $(CXXFLAGS) $(CPPFLAGS)....
- To compose a library: $(AR) libXXX.a xxx1.o xxx2.o xxx3.o .....$(RANLIB) libXXX.a
- To link an executable: $(LINK) $(LDFLAGS) ..... $(LIBS)

For more information on these and other variables, see the GNU autoconf documentation. The specified tools and flags will then be "frozen" inside the makefiles of *build tree* produced by this ***configure*** run.

Some of the above system and third-party packages can be prohibited from use by using the following ***configure*** flags:

--without-sybase (Sybase)

--without-ftds (FreeTDS)

--without-fastcgi (FastCGI)

--without-fltk (FLTK)

--without-wxwin (wxWindows)

--without-ncbi-c (NCBI C Toolkit)

--without-sssdb (NCBI SSS DB)

--without-sssutils (NCBI SSS UTILS)

--without-sss (both *--without-sssdb* and *--without-sssutils*)

--without-geo (NCBI GEO)

--without-sp (NCBI SP)

--*without-pubmed* (NCBI PubMed)

--*without-orbacus* (ORBacus CORBA)

[and MANY more; ./configure –help | grep –e '—without-' will give a current list for both this and the following heading.]

You can control whether to build the following core packages using the following ***configure*** flags:

--*without-serial* -- do not build C++ ASN.1 serialization library and datatool; see in internal/c++/{ src | include}/serial directories

--*without-ctools* -- do not build projects that use NCBI C Toolkit see in internal/c++/{ src | include}/ctools directories

--*without-gui* -- do not build projects that use wxWindows GUI package see in internal/c++/{ src | include}/gui directories

--*with-objects* -- generate and build libraries to serialize ASN.1 objects; see in internal/c++/{ src | include}/objects directories

--*with-internal* -- build of internal projects is by default disabled on most platforms; see in internal/c++/{ src | include}/internal directories

--*without-exe* -- do not build the executables enlisted in the APP_PROJ.

--*without-execopy* -- do not copy (yet build) the executables enlisted in the APP_PROJ.

--*with-lib-rebuilds* -- when building an application, attempt to rebuild all of the libraries it uses in case they are out of date.

--*with-lib-rebuilds=ask* -- as above, but prompt before any needed rebuilds. (Do not prompt for libraries that are up to date.)

Here's a more detailed explanation of --*with-lib-rebuilds*: There are three modes of operation:

In the default mode (--*without-lib-rebuilds*), starting a build from within a subtree (such as internal) will not attempt to build anything outside of that subtree.

In the unconditional mode (--*with-lib-rebuilds*), building an application will make the system rebuild any libraries it requires that are older than their sources. This can be useful if you have made a change that affects everything under objects but your project only needs a few of those libraries; in that case, you can save time by starting the build in your project's directory rather than at the top level.

The conditional mode (--*with-lib-rebuilds=ask*) is like the unconditional mode, except that when the system discovers that a needed library is out of date, it asks you about it. You can then choose between keeping your current version (because you prefer it or because nothing relevant has changed) and building an updated version.

Quick Reconfiguration

Sometimes, you change or add configurables (*.in files, such as *Makefile.in* meta-makefiles) in the *source tree*.

For the *build tree* to pick up these changes, go to the appropriate build directory and run the script ***reconfigure.sh***. It will automatically use just the same command-line arguments that you used for the original configuration of that *build tree*.

Run ***reconfigure.sh*** with argument:

update - if you did not add or remove any *configurables* in the *source tree* but only modified some of them.

reconf - if you changed, added, and/or removed any *configurables* in the *source tree*.

recheck - if you also suspect that your working environment (compiler features, accessibility of third-party packages, etc.) might have changed since your last (re)configuration of the *build tree* and, therefore, you do not want to use the cached check results obtained during the last (re)configuration.

without arguments - printout of script usage info.

Example:

```
cd /home/foobar/c++/GCC-Debug/build
./reconfigure.sh reconf
```

Naturally, *update* is the fastest of these methods, *reconf* is slower, and *recheck* (which is an exact equivalent of re-running the ***configure*** script with the same command-line arguments as were provided during the original configuration) is the slowest.

Building with UNIX

Following are some examples of how to build specific projects and some additional topics:

- General Principles for Building with UNIX
- Building Only Core Libraries and Applications
- Building GUI Libraries and Applications
- Building the Genome Workbench
- Building the Entire Toolkit

General Principles for Building with UNIX

Use this key for the examples in the "Building with UNIX" sections:

$YOUR_WORK_DIR	your directory corresponding to the top-level c++ directory in the source tree
$YOUR_CONFIG_OPTIONS	any optional configuration options you've chosen
--with-flat-makefile	creates a makefile that can build all or selected projects
--without-internal	excludes NCBI-internal projects from the makefile
--without-gui	excludes FLTK-based projects from the makefile
--with-gbench	ensures that the makefile will contain everything necessary to build the Genome Workbench
GCC401-Debug	will be replaced based on the compiler and configuration options you're using
gui/	selects the GUI libraries target in the flat makefile
gui/app/	selects the sub-tree containing the primary Genome Workbench executable and its helpers

all_r	selects a recursive build of all targets at this and lower levels in the source tree

The import_project script builds a single project in the working directory while referencing the rest of a pre-built Toolkit for all other Toolkit components. For example, to build only the app/id2_fetch application and have the rest of the pre-built Toolkit available, use these commands:

```
mkdir $YOUR_WORK_DIR
cd $YOUR_WORK_DIR
import_project app/id2_fetch
cd trunk/c++/src/app/id2_fetch
make
```

The update_projects script builds a single project and all the components it depends on in the working directory, and does not reference or build any other Toolkit components. For example, to build only the corelib project, use these commands:

```
mkdir $YOUR_WORK_DIR
cd $YOUR_WORK_DIR
update_projects corelib .
```

The update_projects script will automatically retrieve updated source code and then prompt you for configuring, compiling, building tests, and running tests.

To run a test suite after building, use this additional command:

```
make check_r
```

Building Only Core Libraries and Applications with UNIX

```
cd $YOUR_WORK_DIR
./configure –without-gui -without-internal $YOUR_CONFIG_OPTIONS
cd GCC401-Debug/build
make all_r
```

Building GUI Libraries and Applications with UNIX

```
cd $YOUR_WORK_DIR
./configure $YOUR_CONFIG_OPTIONS --with-flat-makefile
cd GCC401-Debug/build
make -f Makefile.flat gui/
```

Building the Genome Workbench with UNIX

```
cd $YOUR_WORK_DIR
./configure $YOUR_CONFIG_OPTIONS --with-flat-makefile --with-gbench
cd GCC401-Debug/build
make -f Makefile.flat gui/app/
(cd gui/app/gbench_install && make)
```

Building the Entire Toolkit with UNIX

```
cd $YOUR_WORK_DIR
./configure $YOUR_CONFIG_OPTIONS
cd GCC401-Debug/build
make all_r
```

Using the Toolkit with UNIX

This section discusses the following examples of how to use the Toolkit with UNIX:

- Modify or Debug an Existing Toolkit Application
- Modify or Debug an Existing Toolkit Library

Modify or Debug an Existing Toolkit Application with UNIX

If you want to modify or debug an application (e.g. gi2taxid) start with these commands:

```
cd $YOUR_WORK_DIR
import_project app/gi2taxid
```

You will be prompted to select a desired stability and configuration and then the script will create the include and src trees necessary to work on the chosen application. It will also create all the necessary makefiles to build the application. The makefiles will be configured to use the latest nightly build of the chosen stability and configuration to resolve all dependencies outside the chosen application.

You can now edit, build, and/or debug the application:

```
cd trunk/c++/src/app/gi2taxid
# if you want to make changes, edit the desired file(s)
make all_r
# if desired, debug using your favorite debugger
```

Modify or Debug an Existing Toolkit Library with UNIX

If you want to modify or debug a library (e.g. corelib) start with these commands:

```
cd $YOUR_WORK_DIR
import_project corelib
```

You will be prompted to select a desired stability and configuration and then the script will create the include and src trees necessary to work on the chosen library. It will also create all the necessary makefiles to build the library. The makefiles will be configured to use the latest nightly build of the chosen stability and configuration to resolve all dependencies outside the chosen library.

You can now edit, build, and/or debug (via some application) the library:

```
cd trunk/c++/src/corelib
# if you want to make changes, edit the desired file(s)
make all_r
# if you want to debug the library, build a dependent application
# then debug using your favorite debugger
```

Special Considerations for Specific UNIX Platforms

Most of the non-GCC compilers require special tools and additional mandatory flags to compile and link C++ code properly. That's why there are special scripts that perform the required non-standard, compiler-specific pre-initialization for the tools and flags used before running ***configure***.

These wrapper scripts are located in the *compilers/* directory, and now we have such wrappers for the SUN WorkShop (5.5 through 5.9), GCC and ICC compilers:

- WorkShop.sh {32|64} [build_dir] [--configure-flags]
- WorkShop55.sh {32|64} [build_dir] [--configure-flags]
- ICC.sh [build_dir] [--configure-flags]

Note that these scripts accept all regular ***configure*** flags and then pass them to the ***configure*** script.

The following topics are discussed in this section:

- Linux / ICC
- Cygwin / GCC

Linux / ICC

To build a project on Linux / ICC, just follow the generic UNIX guidelines but instead of running the ./configure.sh script you will need to run compilers/unix/ICC.sh.

Cygwin / GCC

To build a project on Cygwin / GCC, just follow the generic UNIX guidelines but instead of running the ./configure.sh script you will need to run compilers/cygwin/build.sh.

MS Windows

Note: Please also see the General Information for All Platforms section, as it contains relevant information that is not repeated here.

The following topics are discussed in this section:

- MS Visual C++
  - Choosing a Build Scope
  - Configuring
  - Building
  - Using
- Cygwin / GCC

MS Visual C++

The following topics are discussed in this section:

- Choosing a Build Scope
- Configuring
  - Site-Specific Build Tree Configuration
  - Fine-Tuning with MSVC Project Files
    - Excluding project from the build

Choosing a Build Scope with Visual C++

The Toolkit is very large and you may not want to retrieve and build the entire Toolkit if you don't need to. Therefore, after preparing the development environment and getting the source files, you'll need to choose a build scope. Several mechanisms are provided to enable working with only a portion of the Toolkit.

If you are interested in building only one project, you can limit the source code retrieved from the repository:

- using the shell script import_project; or
- using the shell script update_projects.

You can also limit what will be built by choosing a standard solution. Five standard solutions are provided to enable working only with selected portions of the Toolkit.

compilers\msvc1000_prj\static\build\ncbi_cpp.sln

compilers\msvc1000_prj\dll\build\ncbi_cpp.sln

compilers\msvc1000_prj\static\build\gui\ncbi_gui.sln

compilers\msvc1000_prj\dll\build\gui\ncbi_gui.sln

compilers\msvc1000_prj\dll\build\gbench\ncbi_gbench.sln

The first two solutions build console applications and required libraries only; the last three solutions build GUI applications.

You can also choose between static and shared libraries. Building with static libraries will result in much larger applications and require much more disk space. Using static libraries is not an option for the Genome Workbench.

Configuring with Visual C++

Once you have chosen a build scope, you are ready to configure.

If you used either the import_project script or the update_projects script then you don't need to configure because both of those scripts use existing configurations.

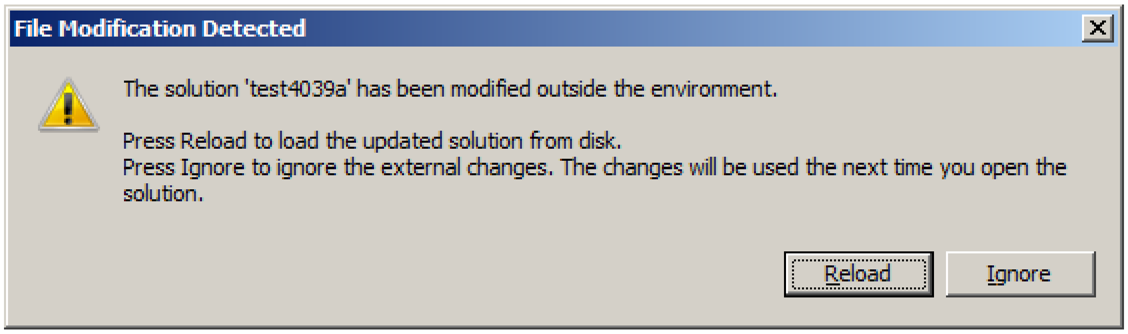
If you chose a standard solution then you will need to configure. Each standard solution contains a special project called **-CONFIGURE-** which is used for generating a Visual Studio project file based on UNIX-style makefile templates src\....\Makefile.*

The Visual Studio specific configuration files are:

- src\build-system\Makefile.mk.in.msvc
- src\build-system\project_tree_builder.ini
- src\....\Makefile.*.msvc

Each of the standard solutions use a predefined list of projects to build, which is taken from scripts\projects\*.lst files.

To configure and generate the project list, open the chosen solution, select the desired configuration, right click on the **-CONFIGURE-** project, and click 'Build'. This will rewrite the project file that Visual C++ is currently using, so you should see one or more dialog boxes similar to this:



The solution 'test4039a' has been modified outside the environment.

Press Reload to load the updated solution from disk.
Press Ignore to ignore the external changes. The changes will be used the next time you open the solution.

Note: At least one such dialog will typically appear *before* the configuration is complete. Therefore, you need to wait until you see the message:

```
******************************************************************************
*
============== It is now safe to reload the solution: ==============
============== Please, close it and open again ==============
******************************************************************************
*
```

in the Output window before reloading. Once this message appears, you can either click "Reload" or click "Ignore" and then manually close and reopen the solution. The reloaded solution will list all configured projects.

A configuration tool with a Java-based GUI is also available and can be launched by building the **-CONFIGURE-DIALOG-** project. For more information on using the configuration GUI, see the general section on configuring.

The following topics discuss configuring with Visual C++ in more detail:

- Site-Specific Build Tree Configuration
- Fine-Tuning with MSVC Project Files

Site-Specific Build Tree Configuration

File project_tree_builder.ini (see Table 4) describes build and source tree configurations, contains information about the location of 3rd-party libraries and applications, and includes information used to resolve macro definitions found in the UNIX -style makefile templates.

Toolkit project makefiles can list (in a pseudo-macro entry called 'REQUIRES') a set of requirements that must be met in order for the project to be built. For example, a project can be built only on UNIX, or only in multi-thread mode, or if a specific external library is available. Depending on which of the requirements are met, the Toolkit configurator may exclude some projects in some (or all) build configurations or define preprocessor and/or makefile macros.

Some of the Toolkit projects can be built differently depending on the availability of non-Toolkit components. For them, there is a list of macros - defined in 'Defines' entry - that define conditional compilation. To establish a link between such a macro and a specific component, the configuration file also has sections with the names of the macro. For each build configuration, project tree builder creates a header file (see 'DefinesPath' entry) and defines these macros there depending on the availability of corresponding components.

Many of the requirements define dependency on components that are 3rd-party packages, such as BerkeleyDB. For each one of these there is a special section (e.g. [BerkeleyDB]) in project_tree_builder.ini that describes the path(s) to the include and library directories of the package, as well as the preprocessor definitions to compile with and the libraries to link against. The Toolkit configurator checks if the package's directories and libraries do exist, and uses this information when generating appropriate MSVS projects.

There are a few indispensable external components that have analogs in the Toolkit. If the external component is not found, the analog in the Toolkit is used. The 'LibChoices' entry identifies such pairs, and 'LibChoiceIncludes' provides additional include paths to the builtin headers.

Note: There are some requirements which, when building for MS Visual Studio, are always or never met. These requirements are listed in 'ProvidedRequests', 'StandardFeatures', or 'NotProvidedRequests' of the 'Configure' section.

Fine-Tuning with MSVC Project Files

While default MSVS project settings are defined in the Makefile.mk.in.msvc file, each project can require additional MSVC-specific fine-tuning, such as compiler or linker options, additional source code, etc. These tune-ups can be specified in Makefile.<project_name>.[lib|app].msvc file located in the project source directory. All entries in such *.msvc file are optional.

Any section name can have one or several optional suffixes, so it can take the following forms:

- SectionName
- SectionName.CompilerVersion
- SectionName.Platform
- SectionName.[static|dll]
- SectionName.[debug|release]
- SectionName.CompilerVersion.[debug|release]
- SectionName.[static|dll].[debug|release]
- SectionName.[debug|release].ConfigurationName
- SectionName.[static|dll].[debug|release].ConfigurationName

CompilerVersion	1000 (i.e. MSVC 2010)
Platform	Win32 or x64
static or dll	type of runtime libraries
debug or release	build configuration type
ConfigurationName	build configuration name (e.g. DebugDLL, or ReleaseMT)

Settings in sections with more detailed names (ones that appear later on this list) override ones in sections with less detailed names (ones that appear earlier).

Note: After changing settings, you will need to reconfigure and reload the solution for the change to take effect.

The following topics discuss further fine-tuning with MSVC project files:

- Excluding a Project From the Build
- Adding Files to a Project
- Excluding Files From a Project
- Adjusting Build Tools Settings
- Specifying Custom Build Rules

To exclude a project from the build, set the 'ExcludeProject' entry in the 'Common' section:

- [Common]
- ExcludeProject=TRUE

To add files to a project, add entries to the 'AddToProject' section. The section can have the following entries:

- [AddToProject]
- HeadersInInclude=
- HeadersInSrc=
- IncludeDirs=
- LIB=
- ResourceFiles=
- SourceFiles=

HeadersInInclude	override default list of headers from include directory
HeadersInSrc	override default list of headers from source directory
IncludeDirs	additional include directories (relative to the source directory)
LIB	additional C++ Toolkit libraries (without extension)
ResourceFiles	MS Windows resource files
SourceFiles	additional (usually MS Windows specific) source files (without extension)

By default, all header files found in the project's include and source directories are added to the MSVS project. If that's not exactly what you need, the list of headers can be overridden using the 'HeadersInInclude' and 'HeadersInSrc' entries. There, file names should be entered with their extension; an exclamation mark means negation; and wildcards are allowed. For example, the entry:

HeadersInInclude = *.h file1.hpp !file2.h

means "add all files with h extension, add file1.hpp, and do not add file2.h".

Note: A single exclamation mark with no file name means "do not add any header files".

All directories given in the 'IncludeDirs' entry should be specified relative to the source directory (absolute paths aren't supported). After reconfiguring, these directories are saved in the AdditionalIncludeDirectories project property - now relative to $(ProjectDir). The following table illustrates this path conversion:

IncludeDirs Path - specified relative to source directory	**AdditionalIncludeDirectories Path - saved relative to $(ProjectDir)**
somedir	..\..\..\..\..\src\$(SolutionName)\somedir
..\\somedir	..\..\..\..\..\src\somedir
..\\..\\somedir	..\..\..\..\..\somedir
..\\..\\..\\somedir	..\..\..\..\..\..\somedir
..\\..\\..\\..\\somedir, etc.	..\..\..\..\..\..\..\somedir, etc.

Although 'IncludeDirs' does not support absolute paths, it is possible to add absolute paths by changing the 'AdditionalOptions' entry in the '[Compiler]' section (see Build Tool Settings).

Here are some example entries for the 'AddToProject' section:

```
[AddToProject]
HeadersInInclude = *.h
HeadersInSrc = task_server.hpp server_core.hpp srv_sync.hpp \
 srv_stat.hpp
IncludeDirs=..\\..\\sra\\sdk\\interfaces
LIB=xser msbuild_dataobj
ResourceFiles=cn3d.rc
SourceFiles = sysalloc
```

To exclude files from a project, set the 'SourceFiles' or 'LIB' entries of the 'ExcludedFromProject' section.

The build tools are 'Compiler', 'Linker', 'Librarian', and 'ResourceCompiler' - that is, the tools used by the MS Visual Studio build system. The names of available entries in any one of these sections can be found in the Makefile.mk.in.msvc file. For the meaning and possible values of these entries, see Microsoft's VCProjectEngine reference, or the specific reference pages for the VCCLCompilerTool, VCLinkerTool, VCLibrarianTool, and VCResourceCompilerTool Interfaces.

Here are some example settings, with some illustrating how section name suffixes can be used:

```
[Compiler]
AdditionalOptions=/I\"\\\\server\\share\\absolute path with spaces\"

[Compiler.release]
Optimization=0
EnableFunctionLevelLinking=FALSE
GlobalOptimizations=FALSE

[Compiler.900]
PreprocessorDefinitions=UCS2;_CRT_SECURE_NO_DEPRECATE=1;
[Compiler.900.release]
PreprocessorDefinitions=UCS2;_SECURE_SCL=0;_CRT_SECURE_NO_DEPRECATE=1;


[Linker]
subSystem = 1
GenerateManifest=true
EmbedManifest=true
AdditionalOptions=test1.lib test2.lib \\\\server\\share\\path_no_spaces\
\test3.lib

[Linker.debug]
OutputFile = $(OutDir)\\python_ncbi_dbapi_d.pyd
[Linker.release]
OutputFile = $(OutDir)\\python_ncbi_dbapi.pyd
```

Relative paths specified in build tool settings are relative to $(ProjectDir).

Note: 'AdditionalOptions' entries are applied when the tool executes - they do not modify other project properties. For example, if you add an include path using 'AdditionalOptions', it will not affect the 'AdditionalIncludeDirectories' property, which is used by the IDE. In this case, Visual C++ will not be able to check syntax, lookup definitions, use IntelliSense, etc. for files in that location while you're editing - but they will compile normally. Therefore, use the 'AddToProject' section (see above) for include directories unless you must use an absolute path.

See the Makefile.mk.in.msvc file for the default MSVS project settings.

To specify custom build rules for selected files in the project (usually non C++ files) use the 'CustomBuild' section. It has a single entry, 'SourceFiles', which lists one or more files to apply the custom build rules to. Then, create a section with the name of the file, and define the following entries there: 'Commandline', 'Description', 'Outputs', and 'AdditionalDependencies' - that is, the same entries as in the Custom Build Step of Microsoft Visual Studio project property pages. This data will then be inserted "as is" into the MSVS project file.

DLL Configuration

The Toolkit UNIX-style makefile templates give a choice of building the library as dynamic or static (or both). However, it is often convenient to assemble a "bigger" DLL made of the sources of several static libraries.

In the Toolkit, such compound DLLs are described using a set of special makefiles in the src/dll subdirectory. Each such file – Makefile.*.dll – contains the following entries:

DLL	name of the compound DLL
HOSTED_LIBS	names of the included static libraries
DEPENDENCIES	dependencies on other static or dynamic libraries
CPPFLAGS	additional compiler flags, specific for this DLL

Fine-Tuning with Environment Variables

It is possible to fine-tune the configuration process by using the following environment variables:

- PREBUILT_PTB_EXE
- PTB_PROJECT

When the PREBUILT_PTB_EXE environment variable defines an existing file (e.g. project_tree_builder.exe), this EXE is used. Otherwise, the configuration process builds project_tree_builder using existing sources, and then uses this EXE. At NCBI, even when PREBUILT_PTB_EXE is not defined, the toolkit still tries to use an external project_tree_builder – to speed up the configuration. Normally, this is the most recent successfully built one. To disable such behavior, this variable should be defined and have the value bootstrap:

PREBUILT_PTB_EXE=bootstrap

The PTB_PROJECT environment variable can be used to redefine the default project list. For example, it can be defined as follows:

PTB_PROJECT=scripts\projects\datatool\project.lst

Building with Visual C++

Once you have chosen a build scope and have configured, you are ready to build. The configure process creates a solution containing all the projects you can build.

To build a library, application, sample, or any other project, simply choose your configuration (e.g. ReleaseDLL), right-click on the desired project, and select "Build". To build all projects in the solution, build the **-BUILD-ALL-** project.

Note: Do not use the 'Build Solution' command because this would include building the **–CONFIGURE-** project, which would result in: (a) reconfiguring (which may not be necessary at the time), and more importantly (b) not building the remaining projects in the solution.

By the way, you can build a desired project by right-clicking on it and selecting build, but debugging applies only to the StartUp Project. To select a project for debugging, right-click the desired project and select "Set as StartUp Project".

Following are some additional build-related topics:

- Building a Custom Solution
- Building External Libraries (Optional)
- The Build Results

Building a Custom Solution

This section deals with building a custom solution within the C++ Toolkit source tree. To build a custom solution outside the source tree, please see the section on using the new_project script.

There is a template solution, compilers\msvc1000_prj\user\build\ncbi_user.sln, that should help you build a customized solution. The project list for this solution is in scripts\projects \ncbi_user.lst

Note: Do not use this solution directly. Instead, make a new solution based on the template:

1. Make copies of the compilers\msvc1000_prj\user\ subtree and the scripts\projects \ncbi_user.lst file (keep the copies in the same folders as the originals).
2. Rename the subtree, solution file, and project list file appropriately, for example to compilers\msvc1000_prj\project_name\, compilers\msvc1000_prj\project_name \build\project_name.sln, and scripts\projects\project_name.lst.
3. In the folder compilers\msvc1000_prj\project_name\build\UtilityProjects\, use a text editor to edit _CONFIGURE_.vcproj, and _CONFIGURE_DIALOG_.vcproj. Change all instances of "ncbi_user" to "project_name".
4. In the same folder, also edit configure._, and configure_dialog._:
   - **a** Change all instances of "ncbi_user" to "project_name".
   - **b** By default, the solution uses static runtime libraries. If you want to use DLL's, also add the '-dll' option to the 'set PTB_FLAGS=' line.
   - **c** By default, the solution uses a project list file. If you don't want to use a project list file (e.g. if you want to use a project tag filter instead), also change the 'set PTB_PROJECT_REQ=' line to the appropriate subtree, e.g. 'set PTB_PROJECT_REQ=src\cgi\'.
   - **d** If you want to use a project tag filter, add the '-projtag' option to the 'set PTB_FLAGS=' line, e.g. 'set PTB_FLAGS=-projtag "core && !test"'. See the section on reducing build scope for more information on using project tags.
5. If your new project will use a project list file, edit scripts\projects\project_name.lst to identify the required project folders.
6. Your custom solution can now be built. Open the solution file compilers \msvc1000_prj\project_name\build\project_name.sln, configure, and build.

Note that the project directory, msvc1000_prj, may be different for your version of Visual C ++.

Building External Libraries (Optional)

Some of the NCBI C++ Toolkit projects make use of the NCBI C Toolkit (not to be confused with the NCBI C++ Toolkit) and/or freely distributed 3rd-party packages (such as BerkeleyDB, LibZ, FLTK, etc.).

At NCBI, these libraries are already installed, and their locations are hard coded in the C++ Toolkit configuration files. If you are outside of NCBI, you may need to build and install these libraries before building the C++ Toolkit.

Alternatively, the source code for the NCBI C Toolkit and the 3rd-party packages can be downloaded from the NCBI FTP site and built - ideally, in all available configurations.

If you do not have the external libraries already installed, you can download, build, and install the NCBI C Toolkit and the freely distributed 3rd-party packages. The source code for the NCBI C Toolkit and the freely distributed 3rd-party packages can be downloaded from the NCBI FTP site and built in all available configurations. Refer to the documentation on the specific packages you wish to install for more information.

The Build Results

The built Toolkit applications and libraries will be put, respectively, to:

compilers\msvc1000_prj\{static|dll}\bin\<config_name>

compilers\msvc1000_prj\{static|dll}\lib\<config_name>

Note that the project directory, msvc1000_prj, may be different for your version of Visual C ++.

Using the Toolkit with Visual C++

This section dissusses the following examples of how to use the Toolkit with Windows:

- Start a New Project That Uses the Toolkit
- Start a New Project in the Toolkit
- Modify or Debug an Existing Project in the Toolkit

Start a New Project That Uses the Toolkit

To use an already built C++ Toolkit (with all its build settings and configured paths), use the new_project script to create a new project:

```
new_project <name> <type> [builddir] [flags]
```

where:

<name>	is the name of the project to create
<type>	is one of the predefined project types
[builddir]	is the location of the C++ Toolkit libraries
[flags]	selects a recursive build of all targets at this and lower levels in the source tree

For example, if the Toolkit is built in the U:\cxx folder, then this command:

```
new_project test app U:\cxx\compilers\msvc1000_prj
```

- creates a new local build tree;
- puts the project source files into the \src\name folder;
- puts the header files into name\include\name;
- puts the Visual Studio project file into name\compilers\msvc1000_prj\static\build \name; and
- puts the solution file into name\compilers\msvc1000_prj\static\build.

To add new source files or libraries to the project, edit name\src\name\Makefile.name.app makefile template, then rebuild the **-CONFIGURE-** project of the solution.

Start a New Project in the Toolkit with Visual C++

Follow the regular UNIX-style guidelines for adding a new project to the Toolkit.

Then, build the **-CONFIGURE-** project and reload the solution.

To start a new project that will become part of the Toolkit, create the makefile template first. For applications it must be named Makefile.< project_name>.app; for libraries - Makefile.<project_name>.lib. If it is a new folder in the source tree, you will also need to create Makefile.in file in the new folder, to specify to the configuration system what should be built in the new folder. Also, the new folder must be listed in the SUB_PROJ section of the parent folder's Makefile.in. Finally, make sure your new project folder is listed in the appropriate project list file in scripts\projects\*.lst. It can be either a subdirectory of an already listed directory, or a new entry in the list.

Modify or Debug an Existing Project in the Toolkit with Visual C++

Within NCBI, the import_project script can be used to work on just a few projects and avoid retrieving and building the whole source tree. For example, to work on the 'corelib' subtree, run:

```
import_project corelib
```

The script will create the build tree, copy (or extract from the repository) relevant files, and create Visual Studio project files and a solution which references pre-built Toolkit libraries installed elsewhere. Then, you can modify and/or debug the project as desired.

Here's an example showing all the steps needed to build and debug the COBALT test application using import_project with Visual C++ (you should be able to apply the approach of this example to your project by changing some names):

1. In the Windows command-line prompt, run:
   import_project algo/cobalt
   This will prepare a Visual Studio solution and open Visual Studio. There, build "cobalt_unit_test.exe". It's all 32-bit by default, even though your Windows is 64-bit.
   (Agree to map "S:" disk if you want to see debug info from the pre-built libraries.)
2. Copy your "data" dir from:
   imported_projects\src\algo\cobalt\unit_test\data
   to:
   imported_projects\compilers\msvc1000_prj\static\build\algo\cobalt\unit_test\data
3. Debug it (right-click on it, and choose Debug).

If this doesn't work (for whatever reasons) on your own PC, you're welcome to use the communal PC servers (via Remote Desktop):

http://intranet.ncbi.nlm.nih.gov/wiki-private/CxxToolkit/index.cgi/Software_Development#Software_Development9

Cygwin / GCC

To build the project with Cygwin / GCC, just follow the generic UNIX guidelines, noting any special considerations.

Mac OS X

Note: Please also see the General Information for All Platforms section, as it contains relevant information that is not repeated here.

This section covers the following topics:

Xcode 3.0, 3.1

Starting with Xcode build system version 3.0, the NCBI C++ Toolkit uses a new approach to configuring and building the toolkit with Mac OS X. The goal is to make the build process match the build process of Microsoft Visual C++ as closely as possible.

The following topics are discussed in this section:

Choosing a Build Scope with Xcode 3.0 or Later

The Toolkit is very large and you may not want to retrieve and build the entire Toolkit if you don't need to. Therefore, after preparing the development environment and getting the source files, you'll need to choose a build scope. Several mechanisms are provided to enable working with only a portion of the Toolkit.

The first thing you can do is to limit the source code retrieved from the repository:

- using the shell script import_project; or
- using the shell script update_projects.

Next, you can limit what will be built by choosing one of the five standard projects:

compilers/xcode30_prj/static/ncbi_cpp.xcodeproj

compilers/xcode30_prj/dll/ncbi_cpp_dll.xcodeproj

compilers/xcode30_prj/static/ncbi_gui.xcodeproj

compilers/xcode30_prj/dll/ncbi_gui_dll.xcodeproj

compilers/xcode30_prj/dll/ncbi_gbench_dll.xcodeproj

The first two projects build console applications and required libraries only; the last three projects build GUI applications:

Note that the project directory, xcode30_prj, may be different for your version of Xcode.

Configuring with Xcode 3.0 or Later

Once you have chosen a build scope, you are ready to configure.

Each standard project contains a single special target called **CONFIGURE**. Building **CONFIGURE** first builds an application called project tree builder (PTB) and then runs that application. PTB overwrites the current standard project file with a new project that contains all the other Xcode build targets. The new build targets are based on UNIX-style makefile templates (src/.../Makefile.*) and are specified by predefined lists of projects in scripts/projects/*.lst files.

When **CONFIGURE** is built, a dialog will pop up stating that the project file has been overwritten by an external process (the external process is the PTB). Reload the project to ensure that it is loaded correctly. Then any or all of the other targets can be built.

A configuration tool with a Java-based GUI is also available and can be launched by building the **CONFIGURE-DIALOG** target. For more information on using the configuration GUI, see the general section on configuring.

You may build any of the five standard projects. The projects in the static directory build libraries and applications using static Toolkit libraries, the other three use dynamic libraries.

To build a specific target, make it an active one and invoke the **Build** command in the Xcode workspace. To build all project targets, build the **BUILD_ALL** target.

Additional configuration files include:

- src/build-system/project_tree_builder.ini

- src/build-system/Makefile.mk.in.xcode
- src/.../Makefile.*.xcode

Modifying project_tree_builder.ini is described below in the section Site-Specific Build Tree Configuration.

Modifying Makefile.mk.in.xcode and Makefile.*.xcode is described below in the section Fine-Tuning Xcode Target Build Settings.

The following topics discuss additional information about configuring with Xcode:

- Site-Specific Build Tree Configuration
- Dynamic Libraries Configuration
- Fine-Tuning Xcode Target Build Settings
- Adding Files to Target
- Specifying a Custom Build Script

Site-Specific Build Tree Configuration

The build tree configuration can be tailored to your site by modifying the file src/build-system/project_tree_builder.ini (see Table 4). For example, you may need to change the location of 3rd-party libraries to match your systems. Or you may need to specify conditions under which a certain project is excluded from the build.

project_tree_builder.ini describes build and source tree configurations; contains information about the location of 3rd-party libraries and applications; and includes information used to resolve macro definitions found in the UNIX-style makefile templates.

Toolkit project makefiles can list a set of requirements that must be met in order for the project to be built. These requirements are specified in the pseudo-macro **REQUIRES**. For example, a project can be built only on UNIX, or only in multi-thread mode, or only if a specific external library is available. Depending on which of the requirements are met, the Toolkit configuration tool may exclude some projects in some (or all) build configurations, preprocessor defines, and/or makefile macros.

Some of the Toolkit projects can be built differently depending on the availability of non-Toolkit components. For those projects, there is a list of macros - defined in the 'Defines' entry - that define conditional compilation. Each of these macros also has its own section in project_tree_builder.ini that links the macro to a specific component. Using the 'Defines' entry and the associated macro sections, a project can be linked to a list of components. For each build configuration, project tree builder creates a header file (see 'DefinesPath' entry) and defines these macros there depending on the availability of the corresponding components.

Many of the requirements define dependencies on 3rd-party packages, such as BerkeleyDB. For each one of these there is a special section (e.g. [BerkeleyDB]) in project_tree_builder.ini that describes the path(s) to the include and library directories of the package, as well as the preprocessor definitions to compile with and the libraries to link against. The Toolkit configurator checks if the package's directories and libraries do exist, and uses this information when generating appropriate projects.

There are a few indispensable external components that have analogs in the Toolkit. If external libraries for these components are not available then the internal analog can be used. The 'LibChoices' entry identifies such pairs, and 'LibChoiceIncludes' provides additional include paths to the built-in headers.

Note: There may be some requirements which are always or never met. These requirements are listed in the 'ProvidedRequests', 'StandardFeatures', or 'NotProvidedRequests' entries of the 'Configure' section.

Dynamic Libraries Configuration

The Toolkit UNIX-style makefile templates give a choice of building the library as dynamic or static (or both). However, it is often convenient to assemble "bigger" dynamic libraries made of the sources of several static libraries.

In the Toolkit, such compound libraries are described using a set of special makefiles in src/dll subdirectory. Each such file – Makefile.*.dll – contains the following entries:

- DLL – the name of the compound dynamic library;
- HOSTED_LIBS – the names of the static libraries to be assembled into the compound dynamic library;
- DEPENDENCIES – dependencies on other static or dynamic libraries; and
- CPPFLAGS – additional compiler flags, specific for this dynamic library.

Fine-Tuning Xcode Target Build Settings

While default build settings are defined in the Makefile.mk.in.xcode file, it is possible to redefine some of them in special tune-up files – Makefile.<project_name>.{lib|app}.xcode – located in the project source directory. All entries in the tune-up files are optional.

Section names in the tune-up files can have one or more optional suffixes and can take any of the following forms:

- SectionName
- SectionName.CompilerVersion
- SectionName.Platform
- SectionName.[static|dll]
- SectionName.[debug|release]
- SectionName.CompilerVersion.[debug|release]
- SectionName.[static|dll].[debug|release]
- SectionName.[debug|release].ConfigurationName
- SectionName.[static|dll].[debug|release].ConfigurationName

Here, 'static' or 'dll' means the type of runtime libraries that a particular build uses; 'debug' or 'release' means the type of the build configuration; and 'ConfigurationName' means the name of the build configuration, for example DebugDLL or ReleaseMT.

Settings in sections with more detailed names (ones that appear later on this list) override ones in sections with less detailed names (ones that appear earlier).

Adding Files to Target

This information should be entered in the 'AddToProject' section. The section can have the following entries:

- [AddToProject]
- SourceFiles=
- IncludeDirs=

- LIB=
- HeadersInInclude=
- HeadersInSrc=

The 'SourceFiles' entry lists additional (usually OSX specific) source files for the project. Source file entries should not include file name extensions. The 'IncludeDirs' entry lists additional include directories, and the 'LIB' entry lists additional libraries for the project.

By default, all header files found in the project's include and source directories are added to the Xcode target. If that's not exactly what you need though, then the default set of headers to be added to the target can be altered using the 'HeadersInInclude' and 'HeadersInSrc' entries. Unlike the 'SourceFiles' entry, file names in these entries should include their extension. Use an exclamation mark to exclude files that would otherwise be included. Wildcards are allowed. For example, the following entry

HeadersInInclude = *.h file1.hpp !file2.h

means "add all files with the .h extension, add file1.hpp, and do not add file2.h".

Note: A single exclamation mark with no file name means "do not add any header files".

Specifying a Custom Build Script

For a particular target, it is possible to specify a custom build script which will run in addition to the standard build operation. This could be used, for example, to copy application resource files once the build is completed. Xcode will automatically incorporate the custom script into the standard build process.

In the appropriate Makefile.*.xcode customization file, define a section called ‘CustomScript’. It has one mandatory entry – Script, and three optional ones:

- Input – a list of input files;
- Output – a list of output files; and
- Shell – which shell to use (the default is ‘/bin/sh’).

Building with Xcode 3.0 or Later

Once you have chosen a build scope and have configured, you are ready to build.

Note: Some projects may require using 3rd-party libraries.

Select the desired project and build it. To build all projects, select the **BUILD-ALL** project.

Following are some examples of how to build specific projects and some additional topics:

- Building 3rd-Party Libraries (Optional)
- Building from a Command-Line
- The Build Results

Build 3rd-Party Libraries (optional)

Some of the NCBI C++ Toolkit projects make use of the NCBI C Toolkit (not to be confused with the NCBI C++ Toolkit) and/or freely distributed 3rd-party packages (such as BerkeleyDB, LibZ, FLTK, etc.).

At NCBI, these libraries are already installed, and their locations are hard coded in the C++ Toolkit configuration files. If you are outside of NCBI, you may need to build and install these libraries before building the C++ Toolkit.

If you do not have the external libraries already installed, you can download, build, and install the NCBI C Toolkit and the freely distributed 3rd-party packages. The source code for the NCBI C Toolkit and the freely distributed 3rd-party packages can be downloaded from the NCBI FTP site and built in all available configurations. Refer to the documentation on the specific packages you wish to install for more information.

Building from a Command-Line with Xcode 3.0 or Later

From the command-line, you can either build exactly as under UNIX, or you can build for Xcode.

To configure for Xcode, first run configure in the Xcode project directory (run configure --help to see available options):

```
cd compilers/xcode30_prj
./configure
```

Once you have configured for Xcode, you can either open and work in the Xcode IDE or build from the command-line.

To build from the command-line, run make all_r. Optionally build the testsuite with make check_r.

```
make all_r
make check_r
```

The Build Results

Applications and libraries produced by the build will be put, respectively, into:

- compilers/xcode30_prj/{static|dll}/bin/<ConfigurationName>
- compilers/xcode30_prj/{static|dll}/lib/<ConfigurationName>

Xcode 1.0, 2.0

For versions of Xcode earlier than 3.0 the handmade scripts have to be used.

The following topics are discussed in this section:

- Build the Toolkit
- The Build Results

Build the Toolkit

Open, build and run a project file in compilers/xCode.

This GUI tool generates a new NCBI C++ Toolkit Xcode project. It allows you to:

- Choose which Toolkit libraries and applications to build.
- Automatically download and install all 3rd-party libraries.
- Specify third-party installation directories.

The Build Results

The above process results in the Toolkit applications and libraries being put into the output directory selected by the user.

Apple Xcode versions 2.0 and above support build configurations. We use the default names Debug and Release, so the built applications will go to, for example:

- <output_dir>/bin/Debug/Genome Workbench.app, or
- <output_dir>/bin/Release/Genome Workbench.app

Apple Xcode versions before 2.0 do not support build configurations, so the build results will always go to:

- <output_dir>/bin/Genome Workbench.app

Most libraries are built as Mach-O dynamically linked and shared (.dylib) and go to:

- <output_dir>/lib

Genome Workbench plugins are built as Mach-O bundles (also with .dylib extension) and get placed inside Genome Workbench application bundle:

- <output_dir>/Genome Workbench.app/Contents/MacOS/plugins

Darwin / GCC

To build the project with Darwin / GCC, just follow the generic UNIX guidelines.

CodeWarrior

For various reasons we have decided to drop support for CodeWarrior. The latest version of the Toolkit that supported CodeWarrior can be found here.

Table 1. Environment variables that affect the build process

Name	**Default**	**Synopsis**
CC	gcc, cc	C compiler
CXX	c++, g++, gcc, CC, cxx, cc++	C++ compiler, also being used as a linker
CPP	$CC -E	C preprocessor
CXXCPP	$CXX -E	C++ preprocessor
AR	ar cru	Librarian
STRIP	strip	To discard symbolic info
CFLAGS	-g or/and/nor -O	C compiler flags
CXXFLAGS	-g or/and/nor -O	C++ compiler flags
CPPFLAGS	-D_DEBUG or/and/nor-DNDEBUG	C/C++ preprocessor flags
LDFLAGS	None	Linker flags
LIBS	None	Libraries to link to every executable
CONFIG_SHELL	/bin/sh	Command interpreter to use in the configuration scripts and makefiles (it must be compatible with sh)

Table 2. User-defined localization variables

Name	Default	Synopsis
THREAD_LIBS	-lpthread	System thread library
NETWORK_LIBS	-lsocket -lnsl	System network libraries
MATH_LIBS	-lm	System math library
KSTAT_LIBS	-lkstat	System kernel statistics library
RPCSVC_LIBS	-lrpcsvc	System RPC services library
CRYPT_LIBS	-lcrypt[_i]	System encrypting library
SYBASE_PATH	/netopt/Sybase/clients/current	Path to Sybase package (but see note below)
FTDS_PATH	/netopt/Sybase/clients-mssql/current	Path to FreeTDS package
FASTCGI_PATH	$NCBI/fcgi-current	Path to the in-house FastCGI client lib
FLTK_PATH	$NCBI/fltk	Path to the FLTK package
WXWIN_PATH	$NCBI/wxwin	Path to the wxWindows package
NCBI_C_PATH	$NCBI	Path to the NCBI C Toolkit
NCBI_SSS_PATH	$NCBI/sss/BUILD	Path to the NCBI SSS package
NCBI_GEO_PATH	$NCBI/geo	Path to the NCBI GEO package
SP_PATH	$NCBI/SP	Path to the SP package
NCBI_PM_PATH	$NCBI/pubmed[64]	Path to the NCBI PubMed package
ORBACUS_PATH	$NCBI/corba/OB-4.0.1	Path to the ORBacus CORBA package

Note: It is also possible to make configure look elsewhere for Sybase by means of --with-sybase-local[=DIR]. If you specify a directory, it will override SYBASE_PATH; otherwise, the default will change to /export/home/sybase/clients/current, but SYBASE_PATH will still take priority. Also, the option --with-sybase-new will change the default version of Sybase from 12.0 to 12.5 and adapt to its layout.

It is also possible to override WXWIN_PATH by --with-wxwin=DIR, FLTK_PATH by --> --with-fltk=DIR, and ORBACUS_PATH by --with-orbacus=DIR.

Table 3. Derived localization variables for makefiles

Name	Value	Used to...
THREAD_LIBS	$THREAD_LIBS	Link with system thread lib.
NETWORK_LIBS	$NETWORK_LIBS	Link with system network libs.
MATH_LIBS	$MATH_LIBS	Link with system math lib.
KSTAT_LIBS	$KSTAT_LIBS	Link with system kernel stat lib.
RPCSVC_LIBS	$RPCSVC_LIBS	Link with system RPC lib.
CRYPT_LIBS	$CRYPT_LIBS	Link with system encrypting lib.
SYBASE_INCLUDE	-I$SYBASE_PATH/include	#include Sybase headers
SYBASE_LIBS	-L$SYBASE_PATH/lib[64] -lblk[_r][64] -lct[_r][64] -lcs[_r][64] -ltcl[_r][64] -lcomn[_r][64] -lintl[_r][64]	Link with Sybase libs.
SYBASE_DLLS	-ltli[_r][64]	Sybase DLL-only libs
SYBASE_DBLIBS	-L$SYBASE_PATH/lib[64] -lsybdb[64]	Link with Sybase DB Lib API.
FTDS_INCLUDE	-I$FTDS_PATH/include	#include FreeTDS headers
FTDS_LIBS	-L$FTDS_PATH/lib -lsybdb -ltds	Link with the FreeTDS API.
FASTCGI_INCLUDE	-I$FASTCGI_PATH/include[64]	#include Fast-CGI headers
FASTCGI_LIBS	-L$FASTCGI_PATH/lib[64] -lfcgi or -L$FASTCGI_PATH/altlib[64] -lfcgi	Link with FastCGI lib.
FLTK_INCLUDE	-I$FLTK_PATH/include	#include FLTK headers
FLTK_LIBS	-L$FLTK_PATH/[GCC-]{Release\|Debug}[MT][64]/lib -lfltk ... -lXext -lX11 ... or -L$FLTK_PATH/lib	Link with FLTK libs.
WXWIN_INCLUDE	-I$WXWIN_PATH/include	#include wxWindows headers
WXWIN_LIBS	-L$WXWIN_PATH/[GCC-]{Release\|Debug}/lib -lwx_gtk[d] -lgtk -lgdk -lgmodule -lglib or -L$WXWIN_PATH/lib	Link with wxWindows libs.
NCBI_C_INCLUDE	-I$NCBI_C_PATH/include[64]	#include NCBI C Toolkit headers
NCBI_C_LIBPATH	-L$NCBI_C_PATH/lib[64] or -L$NCBI_C_PATH/altlib[64]	Path to NCBI C Toolkit libs.
NCBI_C_ncbi	-lncbi	NCBI C Toolkit CoreLib
NCBI_SSS_INCLUDE	-I$NCBI_SSS_PATH/include	#include NCBI SSS headers
NCBI_SSS_LIBPATH	-L$NCBI_SSS_PATH/lib/.... ....{Release|Debug}[GNU][64][mt]	Link with NCBI SSS libs.
NCBI_GEO_INCLUDE	-I$NCBI_GEO_PATH/include	#include NCBI GEO headers
NCBI_GEO_LIBPATH	-L$NCBI_GEO_PATH/lib/.... ...[GCC-|KCC-|ICC-]{Release|Debug}[64]	Link with NCBI GEO libs.
SP_INCLUDE	-I$SP_PATH/include	#include SP headers
SP_LIBS	-L$SP_PATH/{Release|Debug}[MT][64] -lsp	Link with the SP lib.
NCBI_PM_PATH	$NCBI_PM_PATH	Path to the PubMed package.
ORBACUS_INCLUDE	-I$ORBACUS_PATH/include -I$ORBACUS_PATH/{Release|Debug}[MT][64]/inc	#include ORBacus CORBA headers
ORBACUS_LIBPATH	-L$ORBACUS_PATH/{Release|Debug}[MT][64]/lib	Link with ORBacus CORBA libs.

Table 4. Project Tree Builder INI file (Local Site)

Section	Key	Comments
[Configure]	ThirdPartyBasePath, ThirdParty_* ThirdPartyAppsBasePath ThirdParty_C_ncbi	Location of 3rd party libraries and applications
	ProvidedRequests StandardFeatures	List of requirements from UNIX makefiles that are always met
	NotProvidedRequests	List of requirements from UNIX makefiles that are never met. Projects with that require any one of these, will be excluded
	DefinesPath	Path to .h file that will contain HAVE_XXXX definitions. The path is relative from the project tree root.
	Defines	List of HAVE_XXXX preprocessor definitions.
	Macros	List of optional macros. Definition of any such macro depends upon availability of Components
	LibChoices	List of pairs <libID>/<Component>. If the third-party library <Component> is present, then this library will be used instead of the internal library <libID>.
	ThirdPartyLibsBinPathSuffix	Part of the naming convention for third-party DLLs installation makefile.
	ThirdPartyLibsBinSubDir	Part of the third-party DLLs installation target location.
	ThirdPartyLibsToInstall	List of components, which DLLs will be automatically installed in the binary build directory.
[ProjectTree]	MetaData	Makefile.mk.in - in this file the project tree builder will be looking for the UNIX project tree macro definitions.
	include	include "include" branch of project tree
	src	src "src" branch
	dll	Subdirectory with DLL Makefiles
	compilers	compilers "compilers" branch
	projects	scripts/projects "projects" branch
[msvc*]	Configurations	List of buid configurations that use static runtime libraries
		List of build configurations that use dynamic runtime libraries
	msvc_prj	Sub-branch of compilers branch for MSVC projects
	MakefilesExt	Extension of MSVC-specific makefiles
	Projects	"build" sub-branch
	MetaMakefile	Master .msvc makefile - Makefile.mk.in.msvc
[LibChoicesIncludes]	CMPRS_INCLUDE et al.	Definition for the include directories for LibChoices.
[Defines]		Contains definition of macros from UNIX makefiles that cannot be resolved otherwise
[HAVE_XXXX]	Component	List of the components to check. An empty list means that the component is always available. A non-empty list means that the component(s) must be checked on presentation during configure.

[Debug],[DebugDLL],etc...	debug	TRUE means that the debug configuration will be created.
	runtimeLibraryOption	C++ Runtime library to use.
[NCBI_C_LIBS], [FLTK_LIBS_GL]	Component	List of libraries to use.
[<LIBRARY>]	INCLUDE	Include path to the library headers.
	DEFINES	Preprocessor definition for library usage.
	LIBPATH	Path to library.
	LIB	Library files.
	CONFS	List of supported configurations.
[DefaultLibs]	INCLUDE	Default libraries will be added to each project. This section is to negotiate the differences in the default libraries on the UNIX and Win32 platforms. Same as for [<LIBRARY>].
	LIBPATH	Same as for [<LIBRARY>].
	LIB	Same as for [<LIBRARY>].
[Datatool]	datatool	ID of the datatool project. Some projects (with ASN or DTD sources) depend on the datatool.
	Location.App	Location of datatool executable for APP projects.
	Location.Lib	Location of datatool executable for LIB projects.
	CommandLine	Partial command line for datatool.

5: Working with Makefiles

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Building executables and libraries for a large, integrated set of software tools such as the C++ Toolkit, and doing so consistently on different platforms and architectures, is a daunting task. Therefore, the Toolkit developers have expended considerable effort to design a build system based upon the make utility as controlled by makefiles. Although it is, of course, possible to write one's own Toolkit makefile from scratch, it is seldom desirable. To take advantage of the experience, wisdom, and alchemy invested in Toolkit and to help avoid often inscrutable compilation issues:

We strongly advise users to work with the Toolkit's make system.

With minimal manual editing (and after invoking the configure script in your build tree), the build system adapts to your environment, compiler options, defines all relevant makefile macros and targets, allows for recursive builds of the entire Toolkit and targeted builds of single modules, and handles many other details that can confound manual builds.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Major Makefiles

Before describing the make system in detail, we list the major types of makefiles used by the Toolkit:

- meta-makefiles. These files exist for each project and tie the project together in the Toolkit hierarchy; defining those applications and libraries as a project is necessary for (possibly recursively) building.

Generic makefile **Templates** (Makefile*.in). The configure script processes these files from the src hierarchy to substitute for the special tags "@some_name@" and make other specializations required for a given project. Note that meta-makefiles are typically derived from such templates.

- **Customized** makefiles. (Makefile.*.[lib|app]) For each library or application, this file gives specific targets, compiler flags, and other project-specific build instructions. These files appear in the src hierarchy.
- **Configured** makefiles. (Makefile) A makefile generated by configure for each project and sub-project and placed in the appropriate location in the build tree ready for use will be called a “configured makefile”. Note that meta-makefiles in the build tree may be considered “configured”.

Makefile Hierarchy

All Toolkit makefiles reside in either the src directory as templates or customized files, or in the appropriate configured form in each of your <builddir> hierarchies as illustrated in Figure 1

Most of the files listed in Figure 1 are templates from the src directory, with each corresponding configured makefile at the top of the build tree. Of these, <builddir>/Makefile can be considered the master makefile in that it can recursively build the entire Toolkit. The role of each top-level makefile template is summarized as follows:

- Makefile.in - makefile to perform a recursive build in all project subdirectories.
- Makefile.meta.in - included by all makefiles that provide both local and recursive builds.
- Makefile.mk.in - included by all makefiles; sets a lot of configuration variables.
- Makefile.lib.in - included by all makefiles that perform a "standard" library build, when building only static libraries.
- Makefile.dll.in - included by all makefiles that perform a "standard" library build, when building only shared libraries.
- Makefile.both.in - included by all makefiles that perform a "standard" library build, when building both static and shared libraries.
- Makefile.lib.tmpl.in - serves as a template for the project customized makefiles (Makefile.*.lib[.in]) that perform a "standard" library build.
- Makefile.app.in - included by all makefiles that perform a "standard" application build.
- Makefile.app.tmpl.in - serves as a template for the project customized makefiles (Makefile.*.app[.in]) that perform a "standard" application build.
- Makefile.rules.in, Makefile.rules_with_autodep.in -- instructions for building object files; included by most other makefiles.

The project-specific portion of the makefile hierarchy is represented in the figure by the meta-makefile template c++/src/myProj/Makefile.in, the customized makefile c++/src/myProj/Makefile.myProj.[app|lib] (not shown), and the configured makefile c++/myBuild/build/myProj/Makefile. In fact, every project and sub-project in the Toolkit has analogous files specialized to its project; in most circumstances, every new or user project should emulate this file structure to be compatible with the make system.

Meta-Makefiles

A typical meta-makefile template (e.g. Makefile.in in your foo/c++/src/bar_proj/ dir) looks like this:

```
# Supply Makefile.bar_u1, Makefile.bar_u2 ...
#
USR_PROJ = bar_u1 bar_u2 ...

# Supply Makefile.bar_l1.lib, Makefile.bar_l2.lib ...
#
LIB_PROJ = bar_l1 bar_l2 ...

# Supply Makefile.bar_a1.app, Makefile.bar_a2.app ...
#
APP_PROJ = bar_a1 bar_a2 ...

# Subprojects
#
SUB_PROJ = app sub_proj1 sub_proj2

srcdir = @srcdir@
include @builddir@/Makefile.meta
```

This template separately specifies instructions for user, library and application projects, along with a set of three sub-projects that can be made. The mandatory final two lines "srcdir = @srcdir@; include @builddir@/Makefile.meta" define the standard build targets.

Makefile.in Meta Files

The Makefile.in meta-make file in the project's source directory defines a kind of road map that will be used by the configure script to generate a makefile (Makefile) in the corresponding directory of the build tree. Makefile.in does **not** participate in the actual execution of make, but rather, defines what will happen at that time by directing the configure script in the creation of the Makefile that will be executed (see also the description of standard build targets below).

The meta-makefile myProj/Makefile.in should define at least one of the following macros:

- USR_PROJ (optional) - a list of names for user-defined makefiles. This macro is provided for the usage of ordinary stand-alone makefiles which do not utilize the make commands contained in additional makefiles in the top-level build directory. Each p_i listed in USR_PROJ = p_1 ... p_N must have a corresponding Makefile.p_i in the project's source directory. When make is executed, the make directives contained in these files will be executed directly to build the targets as specified.
- LIB_PROJ (optional) - a list of names for library makefiles. For each library l_i listed in LIB_PROJ = l_1 ... l_N, you must have created a corresponding project makefile named Makefile.l_i.lib in the project's source directory. When make is executed, these library project makefiles will be used along with Makefile.lib and Makefile.lib.tmpl (located in the top-level of the build tree) to build the specified libraries.
- ASN_PROJ (optional) is like LIB_PROJ, with one additional feature: Any projects listed there will be interpreted as the names of ASN.1 module specifications to be processed by datatool.

- APP_PROJ (optional) - a list of names for application makefiles. Similarly, each application (p1, p2, ..., pN) listed under APP_PROJ must have a corresponding project makefile named Makefile.p*.app in the project's source directory. When make is executed, these application project makefiles will be used along with Makefile.app and Makefile.app.tmpl to build the specified executables.
- SUB_PROJ (optional) - a list of names for subproject directories (used on recursive makes). The SUB_PROJ macro is used to recursively define make targets; items listed here define the subdirectories rooted in the project's source directory where make should also be executed.

Some additional meta-makefile macros (listed in Table 1) exist to specify various directory paths that make needs to know. The "@"-delimited tokens are substituted during configuration based on your environment and any command-line options passed to configure.

Expendable Projects

By default, failure of any project will cause make to exit immediately. Although this behavior can save a lot of time, it is not always desirable. One way to avoid it is to run make -k rather than make, but then major problems affecting a large portion of the build will still waste a lot of time.

Consequently, the toolkit's build system supports an alternative approach: meta-makefiles can define expendable projects which should be built if possible but are allowed to fail without interrupting the build. The way to do this is to list such projects in EXPENDABLE_*_PROJ rather than *_PROJ.

Project Makefiles

When beginning a new project, the new_project shell script will generate an initial makefile, Makefile.<project_name>_app, that you can modify as needed. In addition, a working sample application can also be checked out to experiment with or as an alternate template.

The import_project script is useful for working on existing Toolkit projects without needing to build the whole Toolkit. In this case things are particularly straightforward as the project will be retrieved complete with its makefile already configured as Makefile.<project_name>_[app|lib]. (Note that there is an underscore in the name, not a period as in the similarly-named customizable makefile from which the configured file is derived.)

If you are working outside of the source tree: In this scenario you are only linking to the Toolkit libraries and will not need to run the configure script, so a Makefile.in template meta-makefile is not required. Some of the typical edits required for the customized makefile are shown in the section on working in a separate directory.

If you are working within the source tree or subtree: Project subdirectories that do not contain any *.in files are ignored by the configure script. Therefore, you will now also need to create a meta-makefile for the newly created project before configuring your build directory to include the new project.

Several examples are detailed on the "Starting New Projects" section.

List of optional packages, features and projects

Table 2 displays the keywords you can list in REQUIRES in a customized application or library makefile, along with the corresponding configure options:

Standard Build Targets

The following topics are discussed in this section:

- Meta-Makefile Targets
- Makefile Targets

Meta-Makefile Targets

The mandatory lines from the meta-makefile example above,

```
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

provide the build rules for the following standard meta-makefile targets:

- all:
  - run "make -f {Makefile.*} all" for the makefiles with the suffixes listed in macro USR_PROJ:
    make -f Makefile.bar_u1 all make -f Makefile.bar_u2 all ......
  - build libraries using attributes defined in the customized makefilesMakefile.*.lib with the suffixes listed in macro LIB_PROJ
  - build application(s) using attributes defined in the customized makefilesMakefile.*.app with the suffixes listed in macro APP_PROJ
- all_r -- first make target all, then run "make all_r" in all subdirectories enlisted in $(SUB_PROJ):
  cd bar_test && make -f Makefile all_r cd bar_sub_proj1 && make -f Makefile all_r ......
- clean, clean_r -- run just the same makefiles but with targets clean and clean_r (rather than all and all_r), respectively
- purge, purge_r -- .....with targets purge and purge_r, respectively

Makefile Targets

The standard build targets for Toolkit makefiles are all, clean and purge. Recall that recursive versions of these targets exist for meta-makefiles.

- all -- compile the object modules specified in the "$(OBJ)" macro, and use them to build the library "$(LIB)" or the application "$(APP)"; then copy the resultant [lib|app] to the [libdir|bindir] directory, respectively
- clean -- remove all object modules and libs/apps that have been built by all
- purge -- do clean, and then remove the copy of the [libs|apps] from the [libdir|bindir] directory.

The customized makefiles do not distinguish between recursive (all_r, clean_r, purge_r) and non-recursive (all, clean, purge) targets -- because the recursion and multiple build is entirely up to the meta-makefiles.

Makefile Macros and Makefile.mk

There is a wide assortment of configured tools, flags, third party packages and paths (see above). They can be specified for the whole build tree with the appropriate entry in Makefile.mk, which is silently included at the very beginning of the customized makefiles used to build libraries and applications.

Many makefile macros are supplied with defaults ORIG_* in Makefile.mk. See the list of ORIG_* macros, and all others currently defined, in the Makefile.mk.in template for details. One should not override these defaults in normal use, but add your own flags to them as needed in the corresponding working macro; e.g., set CXX = $(ORIG_CXX) -DFOO_BAR.

Makefile.mk defines the following makefile macros obtained during the configuration process for flags (see Table 3), system and third-party packages (see Table 4) and development tools (see Table 5).

(*) The values of user-specified environment variables $FAST_CFLAGS, $FAST_CXXFLAGS will substitute the regular optimization flag -O (or -O2, etc.). For example, if in the environment: $FAST_CXXFLAGS=-fast -speedy and $CXXFLAGS=-warn -O3 -std, then in makefile: $(FAST_CXXFLAGS)=-warn -fast -speedy -std.

Example Makefiles

Below are links to examples of typical makefiles, complete with descriptions of their content.

- Inside the Tree
  - An example meta-makefile and its associated project makefiles
  - Library project makefile: Makefile.myProj.lib
  - Application project makefile: Makefile.myProj.app
  - Custom project makefile: Makefile.myProj
- New Projects and Outside the Tree
  - Use Shell Scripts to Create Makefiles
  - Customized makefile to build a library
  - Customized makefile to build an application
  - User-defined makefile to build... whatever

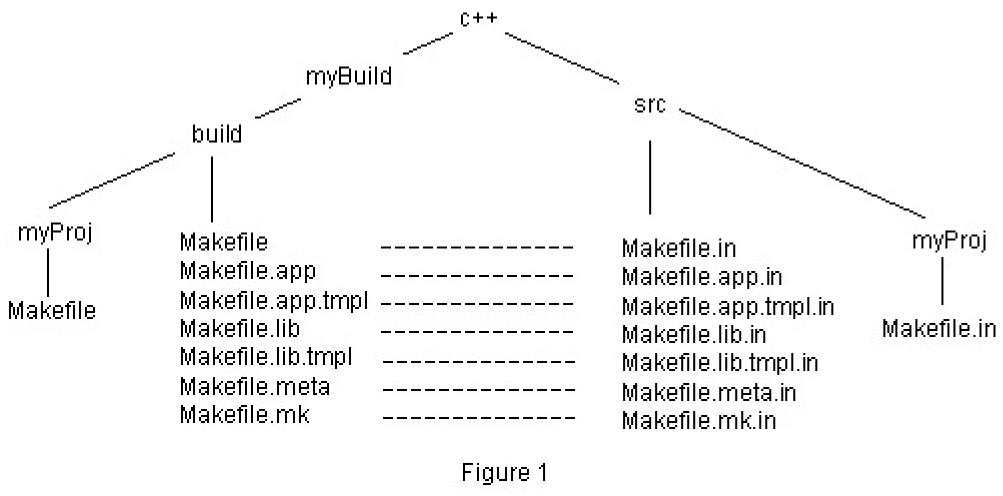


Figure 1. Makefile hierarchy.

Table 1. Path Specification Makefile Macros

Macro	**Source**	**Synopsis**
top_srcdir	@top_srcdir@	Path to the whole NCBI C++ package
srcdir	@srcdir@	Directory in the source tree that corresponds to the directory (./) in the build tree where the build is currently going on
includedir	@includedir@	Top include directory in the source tree
build_root	@build_root@	Path to the whole build tree
builddir	@builddir@	Top build directory inside the build tree
incdir	@incdir@	Top include directory inside the build tree
libdir	@libdir@	Libraries built inside the build tree
bindir	@bindir@	Executables built inside the build tree
status_dir	@status_dir@	Configuration status files

Table 2. Optional Packages, Features, and Projects

Keyword	**Optional...**	**Configure option(s)**
	...package	
Sybase	Sybase libraries	--without-sybase, --with-sybase-local(=DIR), --with-sybase-new
FreeTDS	FreeTDS libraries	--without-ftds, --with-ftds=DIR
Fast-CGI	Fast-CGI library	--without-fastcgi
FLTK	the Fast Light ToolKit	--without-fltk, --with-fltk=DIR
wxWindows	wxWindows	--without-wxwin, --with-wxwin=DIR
C-Toolkit	NCBI C Toolkit	--without-ncbi-c
SSSDB	NCBI SSS DB library	--without-sssdb, --without-sss
SSSUTILS	NCBI SSS UTILS library	--without-sssutils, --without-sss
GEO	NCBI GEO libraries	--without-geo
SP	SP libraries	--without-sp
PubMed	NCBI PubMed libraries	--without-pubmed
ORBacus	ORBacus CORBA	--without-orbacus, --with-orbacus=DIR
	...feature	
MT	multithreading is available	--with-mt
	...project(s)	
serial	ASN.1/XML serialization library and datatool	--without-serial
ctools	projects based on the NCBI C toolkit	--without-ctools
gui	projects that use the wxWindows GUI package	--without-gui
objects	libraries to serialize ASN.1/XML objects	--with-objects
app	standalone applications like ID1_FETCH	--with-app
internal	all internal projects	--with-internal
local_lbsm	IPC with locally running LBSMD	--without-local-lbsm

Table 3. Flags

Macro	**Source**	**Synopsis**
CFLAGS	$CFLAGS	C compiler flags
FAST_CFLAGS	$FAST_CFLAGS	(*) C compiler flags to generate faster code
CXXFLAGS	$CXXFLAGS	C++ compiler flags
FAST_CXXFLAGS	$FAST_CXXFLAGS	(*) C++ compiler flags to generate faster code
CPPFLAGS	$CPPFLAGS	C/C++ preprocessor flags
DEPFLAGS	$DEPFLAGS	Flags for file dependency lists
LDFLAGS	$LDFLAGS	Linker flags
LIB_OR_DLL	@LIB_OR_DLL@	Specify whether to build a library as static or dynamic
STATIC	@STATIC@	Library suffix to force static linkage (see example)

Table 4. System and third-party packages

Macro	**Source**	**Synopsis**
LIBS	$LIBS	Default libraries to link with
PRE_LIBS	$PRE_LIBS	??? Default libraries to link with first
THREAD_LIBS	$THREAD_LIBS	Thread library (system)
NETWORK_LIBS	$NETWORK_LIBS	Network library (system)
MATH_LIBS	$MATH_LIBS	Math library (system)
KSTAT_LIBS	$KSTAT_LIBS	KSTAT library (system)
RPCSVC_LIBS	$RPCSVC_LIBS	RPCSVC library (system)
SYBASE_INCLUDE	$SYBASE_INCLUDE	SYBASE headers
SYBASE_LIBS	$SYBASE_LIBS	SYBASE libraries
FASTCGI_INCLUDE	$FASTCGI_INCLUDE	Fast-CGI headers
FASTCGI_LIBS	$FASTCGI_LIBS	Fast-CGI libraries
NCBI_C_INCLUDE	$NCBI_C_INCLUDE	NCBI C toolkit headers
NCBI_C_LIBPATH	$NCBI_C_LIBPATH	Path to the NCBI C Toolkit libraries
NCBI_C_ncbi	$NCBI_C_ncbi	NCBI C CoreLib
NCBI_SSS_INCLUDE	$NCBI_SSS_INCLUDE	NCBI SSS headers
NCBI_SSS_LIBPATH	$NCBI_SSS_LIBPATH	Path to NCBI SSS libraries
NCBI_PM_PATH	$NCBI_PM_PATH	Path to the PubMed package
ORBACUS_LIBPATH	$ORBACUS_LIBPATH	Path to the ORBacus CORBA libraries
ORBACUS_INCLUDE	$ORBACUS_LIBPATH	Path to the ORBacus CORBA headers

Table 5. Compiler, Linker, and other development Tools

Macro	Source	Synopsis
CC	$CC	C compiler
CXX	$CXX	C++ compiler
LINK	$CXX	Linker (C++-aware)
CPP	$CPP	C preprocessor
CXXCPP	$CXXCPP	C++ preprocessor
AR	$AR	Library archiver
STRIP	$STRIP	Tool to strip symbolic info from binaries
RM	rm -f	Remove file(s)
RMDIR	rm -rf	Remove file(s) and directory(ies) recursively
COPY	cp -p	Copy file (preserving the modification time)
CC_FILTER	@CC_FILTER@	Filters for the C compiler
CXX_FILTER	@CXX_FILTER@	Filters for the C++ compiler
CHECK_ARG	@CHECK_ARG@	
LN_S	@LN_S@	Make a symbolic link if possible; otherwise, hard-link or copy
BINCOPY	@BINCOPY@	Copy a library or an executable -- but only if it was changed

6: Project Creation and Management

Last Update: July 10, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discusses the setup procedures for starting a new project such as the location of makefiles, header files, source files, etc. It also discusses the SVN tree structure and how to use SVN for tracking your code changes, and how to manage the development environment.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Starting New Projects

The following assumes that you have all of the necessary Toolkit components. If you need to obtain part or the Toolkit's entire source tree, consult the FTP instructions or SVN checkout procedures. Please visit the Getting Started page for a broad overview of the NCBI C++ Toolkit and its use.

The following topics are discussed in this section:

New Projects: Location and File Structure

Before creating the new project, you must decide if you need to work within a C++ source tree (or subtree) or merely need to link with the Toolkit libraries and work in a separate directory. The later case is simpler and allows you to work independently in a private directory, but it is not an option if the Toolkit source, headers, or makefiles are to be directly used or altered during the new project's development.

- Work in the Full Toolkit Source Tree
- Work in a Toolkit Subtree
- Work in a Separate Directory

Regardless of where you build your new project, it must adopt and maintain a particular structure. Specifically, each project's source tree relative to $NCBI/c++ should contain:

- include/*.hpp -- project's public headers
- src/*.{cpp, hpp} -- project's source files and private headers
- src/Makefile.in -- a meta-makefile template to specify which local projects (described in Makefile.*.in) and sub-projects (located in the project subdirectories) must be built
- src/Makefile.<project_name>.{lib, app}[.in] -- one or more customized makefiles to build a library or an application
- src/Makefile.*[.in] -- "free style" makefiles (if any)
- sub-project directories (if any)

The following topics are discussed in this section:

- new_project: Starting a New Project outside the C++ Toolkit Tree
- Creating a New Project Inside the C++ Toolkit Tree

new_project: Starting a New Project outside the C++ Toolkit Tree

Script usage:

```
new_project <name> <type>[/<subtype>] [builddir]
```

NOTE: in NCBI, you can (and should) invoke common scripts simply by name - i.e. without path or extension. The proper script located in the pre-built NCBI C++ toolkit directory will be invoked.

This script will create a startup makefile for a new project which uses the NCBI C++ Toolkit (and possibly the C Toolkit as well). Replace <type> with lib for libraries or app for applications.

Sample code will be included in the project directory for new applications. Different samples are available for type=app[/basic] (a command-line argument demo application based on the corelib library), type=app/cgi (for a CGI or Fast-CGI application), type=app/objmgr (for an application using the Object Manager), type=app/objects (for an application using ASN.1 objects), and many others.

You will need to slightly edit the resultant makefile to:

- specify the name of your library (or application)
- specify the list of source files going to it
- modify some preprocessor, compiler, etc. flags, if needed
- modify the set of additional libraries to link to it (if it's an application), if needed

For example:

```
new_project foo app/basic
```

creates a model makefile Makefile.foo_app to build an application using tools and flags hard-coded in $NCBI/c++/Debug/build/Makefile.mk, and headers from $NCBI/c++/include/. The file /tmp/foo/foo.cpp is also created; you can either replace this with your own foo.cpp or modify its sample code as required.

Now, after specifying the application name, list of source files, etc., you can just go to the created working directory foo/ and build your application using:

```
make -f Makefile.foo_app
```

You can easily change the active version of NCBI C++ Toolkit by manually setting variable $(builddir) in the file Makefile.foo_app to the desired Toolkit path, e.g.,

```
builddir = $(NCBI)/c++/GCC-Release/build
```

In many cases, you work on your own project which **is a part** of the NCBI C++ tree, and you do not want to check out, update and rebuild the whole NCBI C++ tree. Instead, you just want to use headers and libraries of the pre-built NCBI C++ Toolkit to build your project. In these cases, use the import_project script instead of new_project.

Note for users inside NCBI: To be able to view debug information in the Toolkit libraries for Windows builds, you will need to have the S: drive mapped to \\snowman\win-coremake\Lib. By default, new_project will make this mapping for you if it's not already done.

Creating a New Project Inside the C++ Toolkit Tree

To create your new project (e.g., "bar_proj") directories in the NCBI C++ Toolkit source tree (assuming that the entire NCBI C++ Toolkit has been checked out to directory foo/c++/):

```
cd foo/c++/include && mkdir bar_proj && svn add bar_proj
cd foo/c++/src && mkdir bar_proj && svn add bar_proj
```

From there, you can now add and edit your project C++ files.

NOTE: remember to add this new project directory to the $(SUB_PROJ) list of the upper level meta-makefile configurable template (e.g., for this particular case, to foo/c++/src/Makefile.in).

Projects and the Toolkit's SVN Tree Structure

(For the overall NCBI C++ SVN tree structure see SVN details.)

Even if you work outside of the C++ tree, it is necessary to understand how the Toolkit uses makefiles, meta-makefiles, and makefile templates, and the SVN tree structure.

The standard SVN location for NCBI C++/STL projects is $SVNROOT/internal/c++/. Public header files (*.hpp, *.inl) of all projects are located below the $SVNROOT/internal/c++/include/ directory. $SVNROOT/internal/c++/src/ directory has just the same hierarchy of subdirectories as .../include/, and its very top level contains:

- Templates of generic makefiles (Makefile.*.in):
  - Makefile.in -- makefile to perform a recursive build in all project subdirectories
  - Makefile.meta.in -- included by all makefiles that provide both local and recursive builds
  - Makefile.lib.in -- included by all makefiles that perform a "standard" library build, when building only static libraries.

    - Makefile.dll.in -- included by all makefiles that perform a "standard" library build, when building only shared libraries.
    - Makefile.both.in -- included by all makefiles that perform a "standard" library build, when building both static and shared libraries.
    - Makefile.lib.tmpl.in -- serves as a template for the project customized makefiles (Makefile.*.lib[.in]) that perform a "standard" library build
    - Makefile.app.in -- included by all makefiles that perform a "standard" application build
    - Makefile.lib.tmpl.in -- serves as a template for the project customized makefiles (Makefile.*.app[.in]) that perform a "standard" application build
    - Makefile.rules.in, Makefile.rules_with_autodep.in -- instructions for building object files; included by most other makefiles
    - Makefile.mk.in -- included by all makefiles; sets a lot of configuration variables
- The contents of each project are detailed above. If your project is to become part of the Toolkit tree, you need to ensure that all makefiles and Makefile*.in templates are available so the master makefiles can properly configure and build it (see "Meta-Makefiles" and "Project Makefiles" below). You will also need to prepare SVN directories to hold the new source and header files.

Creating source and include SVN dirs for a new C++ project

To create your new project (e.g., "bar_proj") directories in the NCBI C++ SVN tree to directory foo/c++/):

```
cd foo/c++/include && mkdir bar_proj && SVN add -m "Project Bar" bar_proj
cd foo/c++/src && mkdir bar_proj && SVN add -m "Project Bar" bar_proj
```

Now you can add and edit your project C++ files in there.

NOTE: remember to add this new project directory to the $(SUB_PROJ) list of the upper level meta-makefile configurable template (e.g., for this particular case, to foo/c++/src/Makefile.in).

Starting New Modules

Projects in the NCBI C++ Toolkit consist of “modules”, which are most often a pair of source (*.cpp) and header (*.hpp) files. To help create new modules, template source and header files may be used, or you may modify the sample code generated by the script new_project. The template source and header files are .../doc/public/framewrk.cpp and .../doc/public/framewrk.hpp. The template files contain a standard startup framework so that you can just cut-and-paste them to start a new module (just don't forget to replace the "framewrk" stubs by your new module name).

- Header file (*.hpp) -- API for the external users. Ideally, this file contains only (well-commented) declarations and inline function implementations for the public interface. No less, and no more.
- Source file (*.cpp) -- Definitions of non-inline functions and internally used things that should not be included by other modules.

On occasion, a second private header file is required for good encapsulation. Such second headers should be placed in the same directory as the module source file.

Each and every source file **must** include the NCBI disclaimer and (preferably) Subversion keywords (e.g. $Id$). Then, the header file must be protected from double-inclusion, and it must define any inlined functions.

Meta-makefiles (to provide multiple and/or recursive builds)

All projects from the NCBI C++ hierarchy are tied together by a set of meta-makefiles which are present in all project source directories and provide a uniform and easy way to perform both local and recursive builds. See more detail on the Working with Makefiles page. A typical meta-makefile template (e.g. Makefile.in in your foo/c++/src/bar_proj/ dir) looks like that:

```
# Makefile.bar_u1, Makefile.bar_u2 ...
USR_PROJ = bar_u1 bar_u2 ...
# Makefile.bar_l1.lib, Makefile.bar_l2.lib ...
LIB_PROJ = bar_l1 bar_l2 ...
# Makefile.bar_a1.app, Makefile.bar_a2.app ...
APP_PROJ = bar_a1 bar_l2 ...
SUB_PROJ = app sub_proj1 sub_proj2
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

This template separately specifies instructions for user, library and application projects, along with a set of three sub-projects that can be made. The mandatory final two lines "srcdir = @srcdir@ ; include @builddir@/Makefile.meta" define the standard build targets.

Project makefiles

Just like the configurable template Makefile.meta.in is used to ease and standardize the writing of meta-makefiles, so there are templates to help in the creation of "regular" project makefiles to build a library or an application. These auxiliary template makefiles are described on the "Working with Makefiles" page and listed above. The configure'd versions of these templates get put at the very top of a build tree.

In addition to the meta-makefile that must be defined for each project, a customized makefile Makefile.<project_name>.[app|lib] must also be provided. The following three sections give examples of customized makefiles for a library and an application, along with a case where a user-defined makefile is required.

You have great latitude in specifying optional packages, features and projects in makefiles. The macro REQUIRES in the examples is one way to allows you access them. See the "Working with Makefiles" page for a complete list; the configuration page gives the corresponding configure options.

The following examples are discussed in this section:

- Example 1: Customized makefile to build a library
- Example 2: Customized makefile to build an application
- Example 3: User-defined makefile to build... whatever

Example 1: Customized makefile to build a library

Here is an example of a customized makefile to build library libxmylib.a from two source files xmy_src1.cpp and xmy_src2.c, and one pre-compiled object file some_obj1.o. To make the example even more realistic, we assume that the said source files include headers from the NCBI C Toolkit.

```
LIB = xmylib
SRC = xmy_src1 xmy_src2
OBJ = some_obj1
REQUIRES = xrequirement
CFLAGS = $(ORIG_CFLAGS) -abc -DFOOBAR_NOT_CPLUSPLUS
CXXFLAGS = $(FAST_CXXFLAGS) -xyz
CPPFLAGS = $(ORIG_CPPFLAGS) -UFOO -DP1_PROJECT -I$(NCBI_C_INCLUDE)
```

- Skip building this library if xrequirement (an optional package or project) is disabled or unavailable.
- Compile xmy_src1.cpp using the C++ compiler $(CXX) with the flags $(FAST_CXXFLAGS) -xyz $(CPPFLAGS), which are the C++ flags for faster code, some additional flags specified by the user, and the original preprocessor flags.
- Compile xmy_src2.c using the C compiler $(CC) with the flags $(ORIG_CFLAGS) -abc -DFOOBAR_NOT_CPLUSPLUS $(CPPFLAGS), which are the original C flags, some additional flags specified by the user, and the original preprocessor flags.
- Using $(AR) and $(RANLIB) [$(LINK_DLL) if building a shared library], compose the library libxmylib.a [libxmylib.so] from the resultant object files, plus the pre-compiled object file some_obj1.o.
- Copy libxmylib.* to the top-level lib/ directory of the build tree (for the later use by other projects).

This customized makefile should be referred to as xmylib in the LIB_PROJ macro of the relevant meta-makefile. As usual, Makefile.mk will be implicitly included.

This customized makefile can be used to build both static and dynamic (DLL) versions of the library. To encourage its build as a DLL on the capable platforms, you can explicitly specify:

```
LIB_OR_DLL = dll
```

or

```
LIB_OR_DLL = both
```

Conversely, if you want the library be always built as static, specify:

```
LIB_OR_DLL = lib
```

Example 2: Customized makefile to build an application

Here is an example of a customized makefile to build the application my_exe from three source files, my_main.cpp, my_src1.cpp, and my_src2.c. To make the example even more realistic, we assume that the said source files include headers from the NCBI SSS DB packages, and the target executable uses the NCBI C++ libraries libxmylib.* and libxncbi.*, plus NCBI SSS DB, SYBASE, and system network libraries. We assume further that the user would prefer to link statically against libxmylib if building the toolkit as both shared and static libraries (configure --with-dll --with-static ...), but is fine with a shared libxncbi.

```
APP = my_exe
SRC = my_main my_src1 my_src2
OBJ = some_obj
LIB = xmylib$(STATIC) xncbi
```

```
REQUIRES = xrequirement
CPPFLAGS = $(ORIG_CPPFLAGS) $(NCBI_SSSDB_INCLUDE)
LIBS = $(NCBI_SSSDB_LIBS) $(SYBASE_LIBS) $(NETWORK_LIBS) $(ORIG_LIBS)
```

- Skip building this library if xrequirement (an optional package or project) is disabled or unavailable.
- Compile my_main.cpp and my_src1.cpp using the C++ compiler $(CXX) with the flags $(CXXFLAGS) (see Note below).
- Compile my_src2.c using the C compiler $(CC) with the flags $(CFLAGS) (see Note below).
- Using $(CXX) as a linker, build an executable my_exe from the object files my_main.o, my_src1.o, my_src2.o, the precompiled object file some_obj.o, NCBI C ++ Toolkit libraries libxmylib.a and libxncbi.*, and NCBI SSS DB, SYBASE, and system network libraries (see Note below).
- Copy the application to the top-level bin/ directory of the build tree (for later use by other projects).

Note: Since we did not redefine CFLAGS, CXXFLAGS, or LDFLAGS, their default values ORIG_*FLAGS (obtained during the build tree configuration) will be used.

This customized makefile should be referred to as my_exe in the APP_PROJ macro of the relevant meta-makefile. Note also, that the Makefile.mk will be implicitly included.

Example 3: User-defined makefile to build... whatever

In some cases, we may need more functionality than the customized makefiles (designed to build libraries and applications) can provide.

So, if you have a "regular" non-customized user makefile, and you want to make from it, then you must enlist this user makefile in the USR_PROJ macro of the project's meta-makefile.

Now, during the project build (and before any customized makefiles are processed), your makefile will be called with one of the standard make targets from the project's build directory. Additionally, the builddir and srcdir macros will be passed to your makefile (via the make command line).

In most cases, it is necessary to know your "working environment"; i.e., tools, flags and paths (those that you use in your customized makefiles). This can be easily done by including Makefile.mk in your makefile.

Shown below is a real-life example of a user makefile:

- build an auxiliary application using the customized makefile Makefile.hc_gen_obj.app (this part is a tricky one...)
- use the resultant application $(bindir)/hc_gen_obj to generate the source and header files humchrom_dat.[ch] from the data file humchrom.dat
- use the script $(top_srcdir)/scripts/if_diff.sh to replace the previous copies (if any) of humchrom_dat.[ch] with the newly generated versions if and only if the new versions are different (or there were no old versions).

And, of course, it provides build rules for all the standard make targets.

```
File $(top_srcdir)/src/internal/humchrom/Makefile.hc_gen_obj:
# Build a code generator for hard-coding the chrom data into
# an obj file
# Generate header and source "humchrom_dat.[ch]" from data
# file "humchrom.dat"
# Deploy the header to the compiler-specific include dir
# Compile source code
#################################
include $(builddir)/Makefile.mk
BUILD__HC_GEN_OBJ = $(MAKE) -f "$(builddir)/Makefile.app.tmpl" \
srcdir="$(srcdir)" TMPL="hc_gen_obj" $(MFLAGS)
all_r: all
all: build_hc_gen_obj humchrom_dat.dep
purge_r: purge
purge: x_clean
 $(BUILD__HC_GEN_OBJ) purge
clean_r: clean
clean: x_clean
 $(BUILD__HC_GEN_OBJ) clean
x_clean:
 -rm -f humchrom_dat.h
 -rm -f humchrom_dat.c
build_hc_gen_obj:
 $(BUILD__HC_GEN_OBJ) all
humchrom_dat.dep: $(srcdir)/data/humchrom.dat $(bindir)/hc_gen_obj
 -cp -p humchrom_dat.c humchrom_dat.save.c
 $(bindir)/hc_gen_obj -d $(srcdir)/data/humchrom.dat
 -f humchrom_dat
 $(top_srcdir)/scripts/if_diff.sh "mv" humchrom_dat.h
 $(incdir)/humchrom_dat.h
 -rm humchrom_dat.h
 $(top_srcdir)/scripts/if_diff.sh "mv" humchrom_dat.c
 humchrom_dat.save.c
 mv humchrom_dat.save.c humchrom_dat.c
 touch humchrom_dat.dep
```

An example of the NCBI C++ makefile hierarchy ("corelib/")

See also the source and build hierarchy charts.

c++/src/Makefile.in:

```
SUB_PROJ = corelib cgi html @serial@ @internal@
include @builddir@/Makefile.meta
```

c++/src/corelib/Makefile.in:

```
LIB_PROJ = corelib
SUB_PROJ = test
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

c++/src/corelib/Makefile.corelib.lib:

```
SRC = ncbidiag ncbiexpt ncbistre ncbiapp ncbireg ncbienv ncbistd
LIB = xncbi
```

c++/src/corelib/test/Makefile.in:

```
APP_PROJ = coretest
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

Managing the Work Environment

The following topics are discussed in this section:

Obtaining the Very Latest Builds

Each new nightly build is available in the $NCBI/c++.by-date/{date} subdirectory. This is done regardless of whether the build succeeds or not.

There are defined symlinks into this directory tree. They include:

- $NCBI/c++ - Symbolic link to $NCBI/c++.production.

- $NCBI/c++.potluck - The most recent nightly build. It contains whatever libraries and executables have managed to build, and it can miss some of the libraries and/or executables. Use it if you desperately need yesterday's bug fix and do not care of the libraries which are missing.
- $NCBI/c++.metastable - The most recent nightly build for which the compilation (but not necessarily the test suite) succeeded in all configurations on the given platform. Please note that some projects, including the entire "gui" tree, are considered expendable due to their relative instability and therefore not guaranteed to be present.
- $NCBI/c++.current - Symbolic link to $NCBI/c++.metastable.
- $NCBI/c++.stable - The most recent nightly build for which the nightly build (INCLUDING the gui projects) succeeded AND the test suite passed all critical tests on the given platform. This would be the preferred build most of the time for the developers whose projects make use of the actively developed C++ Toolkit libraries. It is usually relatively recent (usually no more than 1 or 2 weeks behind), and at the same time quite stable.
- $NCBI/c++.frozen - A "production candidate" build made out of the production codebase. There are usually two such builds made for each version of production codebase -- one is for the original production build, and another (usually made in about 2 months after the original production build) is the follow-up bugfix build.
- $NCBI/c++.production - The most recent production snapshot. This is determined based on general stability of the Toolkit and it is usually derived off the codebase of one of the prior "c++.stable" builds. Its codebase is the same for all platforms and configurations. It is installed only on the major NCBI development platforms (Linux, MS-Windows, and MacOS). It is the safest bet for long-term development. It changes rarely, once in 1 to 3 months. Also, unlike all other builds mentioned here it is guaranteed to be accessible for at least a year (or more), and its DLLs are installed on all (including production) Linux hosts.
- $NCBI/c++.prod-head - This build is for NCBI developers to quickly check their planned stable component commits using import_project. It is based on the repository path toolkit/production/candidates/production.HEAD – which is the HEAD SVN revision of the C++ Stable Components on which the latest c++.production build was based. It is available on 64-bit Linux.
- $NCBI/c++.trial - This build is for NCBI developers to quickly check their planned stable component commits using import_project. It is based on the repository path toolkit/production/candidates/trial – which is usually a codebase for the upcoming production build. It is available on 64-bit Linux.

Working in a separate directory

The following topics are discussed in this section:

- Setting up Directory Location
- The Project's Makefile
- Testing your setup

Setting up Directory Location

There are two topics relevant to writing an application using the NCBI C++ Toolkit:

- Where to place the source and header files for the project
- How to create a makefile which can link to the correct C++ libraries

What you put in your makefile will depend on where you define your working directory. In this discussion, we assume you will be working **outside** the NCBI C++ tree, say in a directory called newproj. This is where you will write both your source and header files. The first step then, is to create the new working directory and use the new_project script to install a makefile there:

```
mkdir newproj
new_project newproj app $NCBI/c++/GCC-Debug/build
 Created a model makefile "/home/user/newproj/Makefile.newproj_app".
```

The syntax of the script command is:

```
new_project <project_name> <app | lib> [builddir]
```

where: - project_name is the name of the directory you will be working in - app (lib) is used to indicate whether you will be building an application or a library - builddir (optional) specifies what version of the pre-built NCBI C++ Toolkit libraries to link to

Several build environments have been pre-configured and are available for developing on various platforms using different compilers, in either **debug** or **release** mode. These environments include custom-made configuration files, makefile templates, and links to the appropriate pre-built C++ Toolkit libraries. To see a list of the available environments for the platform you are working on, use: ls -d $NCBI/c++/*/build. For example, on Solaris, the build directories currently available are shown in Table 1.

In the example above, we specified the GNU compiler debug environment: $NCBI/c++/GCC-Debug/build. For a list of currently supported compilers, see the release notes. Running the new_project script will generate a ready-to-use makefile in the directory you just created. For a more detailed description of this and other scripts to assist you in the set-up of your working environment, see Starting a new C++ project.

The Project's Makefile

The file you just created with the above script will be called Makefile.newproj_app. In addition to other things, you will see definitions for: - $(builddir) - a path to the build directory specified in the last argument to the above script - $(srcdir) - the path to your current working directory (".") - $(APP) - the application name - $(OBJ) - the names of the object modules to build and link to the application - $(LIB) - specific libraries to link to in the NCBI C++ Toolkit - $(LIBS) - all other libraries to link to (outside the C++ Toolkit)

$(builddir)/lib specifies the library path (-L), which in this case points to the GNU debug versions of the NCBI C++ Toolkit libraries. $(LIB) lists the individual libraries in this path that you will be linking to. Minimally, this should include xncbi - the library which implements the foundational classes for the C++ tools. Additional library names (e.g. xhtml, xcgi, etc.) can be added here.

Since the shell script assumes you will be building a single executable with the same name as your working directory, the application is defined simply as newproj. Additional targets to build can be added in the area indicated towards the end of the file. The list of objects (OBJ) should include the names (without extensions) of all source files for the application (APP). Again, the script makes the simplest assumption, i.e. that there is a single source file named newproj.cpp. Additional source names can be added here.

Testing your setup

For a very simple application, this makefile is ready to be run. Try it out now, by creating the file newproj.cpp:

```
// File name: newproj.cpp
#include <iostream>
using namespace std;
int main() {
cout << "Hello again, world" << endl;
}
```

and running:

```
make -f Makefile.newproj_app
```

Of course, it wasn't necessary to set up the directories and makefiles to accomplish this much, as this example does not use any of the C++ classes or resources defined in the NCBI C++ Toolkit. But having accomplished this, you are now prepared to write an actual application, such as described in Writing a simple application project

Most real applications will at a minimum, require that you #include ncbistd.hpp in your header file. In addition to defining some basic NCBI C++ Toolkit objects and templates, this header file in turn includes other header files that define the C Toolkit data types, NCBI namespaces, debugging macros, and exception classes. A set of template files are also provided for your use in developing new applications.

Working Independently In a C++ Subtree

An alternative to developing a new project from scratch is to work within a subtree of the main NCBI C++ source tree so as to utilize the header, source, and make files defined for that subtree. One way to do this would be to check out the entire source tree and then do all your work within the selected subtree(s) only. A better solution is to create a new working directory and check out only the relevant subtrees into that directory. This is somewhat complicated by the distributed organization of the C++ SVN tree: header files are (recursively) contained in an include subtree, while source files are (recursively) contained in a src subtree. Thus, multiple checkouts may be required to set things up properly, and the customized makefiles (Makefile.*.app) will need to be modified. The shell script import_project will do all of this for you. The syntax is:

```
import_project subtree_name [builddir]
```

where:

- subtree_name is the path to a selected directory inside [internal/]c++/src/
- builddir (optional) specifies what version of the pre-built NCBI C++ Toolkit libraries to link to.

As a result of executing this shell script, you will have a new directory created with the pathname ./[internal/]c++/ whose structure contains "slices" of the original SVN tree. Specifically, you will find:

```
./[internal/]c++/include/subtree_name
./[internal/]c++/src/subtree_name
```

The src and include directories will contain all of the requested subtree's source and header files along with any hierarchically defined subdirectories. In addition, the script will create new makefiles with the suffix *_*app*. These makefiles are generated from the original customized makefiles (Makefile.*.app) located in the original src subtrees. The customized makefiles were designed to work only in conjunction with the build directories in the larger NCBI C++ tree; the newly created makefiles can be used directly in your new working directories.

You can re-run import_project to add multiple projects to your tree.

Note: If you'd like to import both internal and public projects into a single tree, you'll need to use the -topdir option, which will locate the public project within the internal tree, for example:

```
import_project internal/demo/misc/xmlwrapp
import_project -topdir trunk/internal/c++ misc/xmlwrapp
pushd trunk/internal/c++/src/misc/xmlwrapp
make
popd
pushd trunk/internal/c++/src/internal/demo/misc/xmlwrapp
make
```

In this case, your public projects will be located in the internal tree. You must build in each imported subtree, in order from most-dependent to least-dependent so that the imported libraries will be linked to rather than the pre-built libraries.

The NCBI C++ Toolkit project directories, along with the libraries they implement and the logical modules they entail, are summarized in the Library Reference.

Two project directories, internal and objects, may have some subdirectories for which the import_project script does not work normally, if at all. The internal subdirectories are used for in-house development, and the author of a given project may customize the project for their own needs in a way that is incompatible with import_project. The objects subdirectories are used as the original repositories for ASN.1 specifications (which are available for use in your application as described in the section Processing ASN.1 Data), and subsequently, for writing the object definitions and implementations created by the datatool program. Again, these projects can be altered in special ways and some may not be compatible with import_project. Generally, however, import_project should work well with most of these projects.

Working within the C++ source tree

The following topics are discussed in this section:

- The configure scripts

Most users will find that working in a checked-out subtree or a private directory is preferable to working directly in the C++ source tree. There are two good reasons to avoid doing so:

- Building your own versions of the extensive libraries can be very time-consuming.
- There is no guarantee that the library utilities your private code links to have not become obsolete.

This section is provided for those developers who must work within the source tree. The Library Reference provides more complete and technical discussion of the topics reviewed here, and many links to the relevant sections are provided. This page is provided as an overview of material presented in the Library Reference and on the Working with Makefiles pages.

Checkout (*) the source tree and configure a build directory

To checkout full Toolkit tree:

```
svn co https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/internal/c++ c++
```

or, if you don't need internal projects:

```
svn co https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/c++ c++
```

Once you have done so, you will need to run one of the configure scripts in the Toolkit's root directory. For example, to configure your environment to work with the gcc compiler (on any platform), just run: ./configure.

Users working under Windows should consult the MS Visual C++ section in the chapter on Configuring and Building the Toolkit.

The configure script is a multi-platform configuration shell script (generated from configure.in using autoconf). Here are some pointers to sections that will help you configure the build environment:

- Wrapper scripts supporting various platforms
- Optional configuration flags

The configure script concludes with a message describing how to build the C++ Toolkit libraries. If your application will be working with ASN.1 data, use the --with-objects flag in running the configure script, so as to populate the include/objects and src/objects subdirectories and build the objects libraries. The objects directories and libraries can also be updated separately from the rest of the compilation, by executing make inside the build/objects directory. Prior to doing so however, you should always verify that your build/bin directory contains the latest version of datatool.

The project's directories and makefiles

To start a new project ("myProj"), you should begin by creating both a src and an include subtree for that project inside the C++ tree. In general, all header files that will be accessed by multiple source modules outside the project directory should be placed in the include directory. Header files that will be used solely inside the project's src directory should be placed in the src directory, along with the implementation files.

In addition to the C++ source files, the src subtrees contain meta-makefiles named Makefile.in, which are used by the configure script to generate the corresponding makefiles in the build subtrees. Figure 1 shows slices of the directory structure reflecting the correspondences between the meta-makefiles in the src subtrees and makefiles in the build subtrees. Figure 2 is a sketch of the entire C++ tree in which these directories are defined.

During the configuration process, each of the meta-makefiles in the top-level of the src tree is translated into a corresponding makefile in the top-level of the build tree. Then, for each project directory containing a Makefile.in, the configure script will: (1) create a corresponding subdirectory of the same name in the build tree if it does not already exist, and (2) generate a corresponding makefile in the project's build subdirectory. The contents of the project's Makefile.in in the src subdirectory determine what is written to the project's makefile in the build subdirectory. Project subdirectories that do not contain a Makefile.in file are ignored by the configure script.

Thus, you will also need to create a meta-makefile in the newly created src/myProj directory before configuring your build directory to include the new project. The configure script will then create the corresponding subtree in the build directory, along with a new makefile generated from the Makefile.in you created. See Makefile Hierarchy (Chapter 4, Figure 1) and Figure 1.

Makefile.in meta files

The meta-makefile myProj/Makefile.in should define at least one of the following macros:

- USR_PROJ (optional) - a list of names for user-defined makefiles.
  This macro is provided for the usage of ordinary stand-alone makefiles which do not utilize the make commands contained in additional makefiles in the top-level build directory. Each p_i listed in USR_PROJ = p_1 ... p_N must have a corresponding Makefile.p_i in the project's source directory. When make is executed, the make directives contained in these files will be executed directly to build the targets as specified.
- LIB_PROJ (optional) - a list of names for library makefiles.
  For each library l_i listed in LIB_PROJ = l_1 ... l_N, you must have created a corresponding project makefile named Makefile.l_i.lib in the project's source directory. When make is executed, these library project makefiles will be used along with Makefile.lib and Makefile.lib.tmpl (located in the top-level of the build tree) to build the specified libraries.
- APP_PROJ (optional) - a list of names for application makefiles.
  Similarly, each application (p1, p2, ..., pN) listed under APP_PROJ must have a corresponding project makefile named Makefile.p*.app in the project's source directory. When make is executed, these application project makefiles will be used along with Makefile.app and Makefile.app.tmpl to build the specified executables.
- SUB_PROJ (optional) - a list of names for subproject directories (used on recursive makes).
  The SUB_PROJ macro is used to recursively define make targets; items listed here define the subdirectories rooted in the project's source directory where make should also be executed.

The Makefile.in meta file in the project's source directory defines a kind of road map that will be used by the configure script to generate a makefile (Makefile) in the corresponding directory of the build tree. Makefile.in does *not* participate in the actual execution of make, but rather, defines what will happen at that time by directing the configure script in the creation of the Makefile that **will** be executed (see also the description of Makefile targets).

An example meta-makefile and its associated project makefiles

A simple example should help to make this more concrete. Assuming that myProj is used to develop an application named myProj, myProj/Makefile.in should contain the following:

```
####### Example: src/myProj/Makefile.in
APP_PROJ = myProj
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

The last two lines in Makefile.in should always be exactly as shown here. These two lines specify make variable templates using the @var_name@ syntax. When generating the corresponding makefile in the build directory, the configure script will substitute each identifier name bearing that notation with full path definitions.

The corresponding makefile in build/myProj generated by the configure script for this example will then contain:

```
####### Example: myBuild/build/myProj/Makefile
# Generated automatically from Makefile.in by configure.
APP_PROJ = myProj
srcdir = /home/zimmerma/internal/c++/src/myProj
include /home/zimmerma/internal/c++/myBuild/build/Makefile.meta
```

As demonstrated in this example, the @srcdir@ and @builddir@ aliases in the makefile template have been replaced with absolute paths in the generated makefile, while the definition of APP_PROJ is copied verbatim.

The only build target in this example is myProj. myProj is specified as an application - not a library - because it is listed under APP_PROJ rather than under LIB_PROJ. Accordingly, there must also be a file named Makefile.myProj.app in the src/myProj directory. A project's application makefile specifies:

- APP - the name to be used for the resulting executable
- OBJ - a list of object files to use in the compilation
- LIB - a list of NCBI C++ Toolkit libraries to use in the linking
- LIBS - a list of other libraries to use in the linking

There may be any number of application or library makefiles for the project, Each application should be listed under APP_PROJ and each library should be listed under LIB_PROJ in Makefile.in. A suitable application makefile for this simple example might contain just the following text:

```
####### Example: src/myProj/Makefile.myProj.app
APP = myProj
OBJ = myProj
LIB = xncbi
```

In this simple example, the APP_PROJ definition in Makefile.in is identical to the definitions of both APP and OBJ in Makefile.myProj.app. This is not always the case, however, as the APP_PROJ macro is used to define which makefiles in the src directory should be used during compilation, while APP defines the name of the resulting executable and OBJ specifies the names of object files. (Project makefiles for applications are described in more detail below.)

Executing make

Given these makefile definitions, executing make all_r in the build project subdirectory indirectly causes build/Makefile.meta to be executed, which sets the following chain of events in motion:

1. For each proj_name listed in USR_PROJ, Makefile.meta first tests to see if Makefile.proj_name is available in the current build directory, and if so, executes:

   ```
   make -f Makefile.proj_name builddir="$(builddir)"
   srcdir="$(srcdir)" $(MFLAGS)
   ```

   Otherwise, Makefile.meta assumes the required makefile is in the project's source directory, and executes:

   ```
   make -f $(srcdir)/Makefile.proj_name builddir="$(builddir)" srcdir="$(srcdir)" $
   (MFLAGS)
   ```

   In either case, the important thing to note here is that the commands contained in the project's makefiles are executed directly and are **not** combined with additional makefiles in the top-level build directory. The aliased srcdir, builddir, and MFLAGS are still available and can be referred to inside Makefile.proj_name. By default, the resulting libraries and executables are written to the build directory only.
2. For each lib_name listed in LIB_PROJ,

   ```
   make -f $(builddir)/Makefile.lib.tmpl
   ```

   is executed. This in turn specifies that $(builddir)/Makefile.mk, $(srcdir)/ Makefile.lib_name.lib, and $(builddir)/Makefile.lib should be included in the generated makefile commands that actually get executed. The resulting libraries are written to the build subdirectory and copied to the lib subtree.
3. For each app_name listed in APP_PROJ,

   ```
   make -f $(builddir)/Makefile.app.tmpl
   ```

   is executed. This in turn specifies that $(builddir)/Makefile.mk, $(srcdir)/ Makefile.app_name.app, and $(builddir)/Makefile.app should be included in the generated makefile commands that actually get executed. The resulting executables are written to the build subdirectory and copied to the bin subtree.
4. For each dir_name listed in SUB_PROJ (on make all_r),

   ```
   cd dir_name
   make all_r
   ```

   is executed. Steps (1) - (3) are then repeated in the project subdirectory.

More generally, for each subdirectory listed in SUB_PROJ, the configure script will create a relative subdirectory inside the new build project directory, and generate the new subdirectory's Makefile from the corresponding meta-makefile in the src subtree. Note that each subproject directory must also contain its own Makefile.in along with the corresponding project makefiles. The recursive make commands, make all_r, make clean_r, and make purge_r all refer to this definition of the subprojects to define what targets should be recursively built or removed.

Custom project makefile: Makefile.myProj (*)

As described, regular makefiles contained in the project's src directory will be invoked from the build directory if their suffixes are specified in the USR_PROJ macro. This macro is originally defined in the project's src directory in the Makefile.in meta file, and is propagated to the corresponding Makefile in the build directory by the configure script.

For example, if USR_PROJ = myProj in the build directory's Makefile, executing make will cause Makefile.myProj (the project makefile) to be executed. This project makefile may be located in either the current build directory **or** the corresponding src directory. In either case, although the makefile is executed directly, references to the source or object files (contained in the project makefile) must give complete paths to those files. In the first case, make is invoked as: make -f Makefile.myProj, so the makefile is located in the current working (build) directory but the source files are not. In the second case, make is invoked as:

make -f $(srcdir)/Makefile.myProj,

so both the project makefile **and** the source files are non-local. For example:

```
####### Makefile.myProj
include $(NCBI)/ncbi.mk
# use the NCBI default compiler for this platform
CC = $(NCBI_CC)
# along with the default include
INCPATH = $(NCBI_INCDIR)
# and library paths
LIBPATH = $(NCBI_LIBDIR)
all: $(srcdir)/myProj.c
 $(CC) -o myProj $(srcdir)/myProj.c $(NCBI_CFLAGS) -I($INCPATH) \
 -L($LIBPATH) -lncbi
 cp -p myProj $(builddir)/bin
clean:
 -rm myProj myProj.o
purge: clean
 -rm $(builddir)/bin/myProj
```

will cause the C program myProj to be built directly from Makefile.myProj using the default C compiler, library paths, include paths, and compilation flags defined in ncbi.mk. The executables and libraries generated from the targets specified in USR_PROJ are by default written to the current build directory only. In this example however, they are also explicitly copied to the bin directory, and accordingly, the purge directives also remove the copied executable.

Library project makefile: Makefile.myProj.lib (*)

Makefile.lib_name.lib should contain the following macro definitions:

- $(SRC) - the names of all source files to compile and include in the library
- $(OBJ) - the names of any pre-compiled object files to include in the library
- $(LIB) - the name of the library being built

In addition, any of the make variables defined in build/Makefile.mk, such as $CPPFLAGS, $LINK, etc., can be referred to and/or redefined in the project makefile, e.g.:

```
CFLAGS = $(ORIG_CFLAGS) -abc -DFOOBAR_NOT_CPLUSPLUS
CXXFLAGS = $(ORIG_CXXFLAGS) -xyz
CPPFLAGS = $(ORIG_CPPFLAGS) -UFOO -DP1_PROJECT -I$(NCBI_C_INCLUDE)
LINK = purify $(ORIG_LINK)
```

For an example from the Toolkit, see Makefile.corelib.lib, and for a documented example, see example 1 above. This customized makefile can be used to build both static and dynamic (DLL) versions of the library. To build as a DLL on the appropriate platforms, you can explicitly specify:

```
LIB_OR_DLL = dll
```

Conversely, if you want the library to always be built as static, specify:

```
LIB_OR_DLL = lib
```

Application project makefile: Makefile.myProj.app (*)

Makefile.app_name.app should contain the following macro definitions:

- $(SRC) - the names of the object modules to build and link to the application
- $(OBJ) - the names of any pre-compiled object files to include in the linking
- $(LIB) - specific libraries in the NCBI C++ Toolkit to include in the linking
- $(LIBS) - all other libraries to link to (outside the C++ Toolkit)
- $(APP) - the name of the application being built

For example, if C Toolkit libraries should also be included in the linking, use:

```
LIBS = $(NCBI_C_LIBPATH) -lncbi $(ORIG_LIBS)
```

The project's application makefile can also redefine the compiler and linker, along with other flags and tools affecting the build process, as described above for Makefile.*.lib files. For an example from the Toolkit, see Makefile.coretest.app, and for a documented example, see example 2 above.

Defining and running tests

The definition and execution of unit tests is controlled by the CHECK_CMD macro in the test application's makefile, Makefile.app_name.app. If this macro is not defined (or commented out), then no test will be executed. If CHECK_CMD is defined, then the test it specifies will be included in the automated test suite and can also be invoked independently by running "make check".

To include an application into the test suite it is necessary to add just one line into its makefile Makefile.app_name.app:

```
CHECK_CMD =
```

or

```
CHECK_CMD = command line to run application test
```

For the first form, where no command line is specified by the CHECK_CMD macro, the program specified by the makefile variable APP will be executed (without any parameters).

For the second form: If your application is executed by a script specified in a CHECK_CMD command line, and it doesn't read from STDIN, then the script should invoke it like this:

```
$CHECK_EXEC app_name arg1 arg2 ...
```

If your application *does* read from STDIN, then CHECK_CMD scripts should invoke it like this:

```
$CHECK_EXEC_STDIN app_name arg1 arg2 ...
```

Note: Applications / scripts in the CHECK_CMD definition should **not** use ".", for example:

```
$CHECK_EXEC ./app_name arg1 arg2 ... # Do not prefix app_name with ./
```

Scripts invoked via CHECK_CMD should pass an exit code to the testing framework via the exitcode variable, for example:

```
exitcode=$?
```

If your test program needs additional files (for example, a configuration file, data files, or helper scripts referenced in CHECK_CMD), then set CHECK_COPY to point to them:

```
CHECK_COPY = file1 file2 dir1 dir2
```

Before the tests are run, all specified files and directories will be copied to the build or special check directory (which is platform-dependent). Note that all paths to copied files and directories must be relative to the application source directory.

By default, the application's execution time is limited to 200 seconds. You can set a new limit using:

```
CHECK_TIMEOUT = <time in seconds>
```

If application continues execution after specified time, it will be terminated and test marked as FAILED.

If you'd like to get nightly test results automatically emailed to you, add your email address to the WATCHERS macro in the makefile. Note that the WATCHERS macro has replaced the CHECK_AUTHORS macro which had a similar purpose.

For information about using Boost for unit testing, see the "Boost Unit Test Framework" chapter.

The configure scripts

A number of compiler-specific wrappers for different platforms are described in the chapter on configuring and building. Each of these wrappers performs some pre-initialization for the tools and flags used in the configure script before running it. The compiler-specific wrappers are in the c++/compilers directory. The configure script serves two very different types of function: (1) it tests the selected compiler and environment for a multitude of features and generates #include and #define statements accordingly, and (2) it reads the Makefile.in files in the src directories and creates the corresponding build subtrees and makefiles accordingly.

Frequently during development it is necessary to make minor adjustments to the Makefile.in files, such as adding new projects or subprojects to the list of targets. In these contexts however, the compiler, environment, and source directory structures remain unchanged, and configure is actually doing much more work than is necessary. In fact, there is even some risk of failing to re-create the same configuration environment if the user does not exactly duplicate the same set of configure flags when re-running configure. In these situations, it is preferable to run an auxiliary script named config.status, located at the top level of the build directory in a subdirectory named status.

In contrast, changes to the src directory structure, or the addition/deletion of Makefile.in files, all require re-running the configure script, as these actions require the creation/deletion of subdirectories in the build tree and/or the creation/deletion of the associated Makefile in those directories.

Working with the serializable object classes

The following topics are discussed in this section:

Serializable Objects

All of the ASN.1 data types defined in the C Toolkit have been re-implemented in the C++ Toolkit as serializable objects. Header files for these classes can be found in the include/ objects directories, and their implementations are located in the src/objects directories. and

The implementation of these classes as serializable objects has a number of implications. It must be possible to use expressions like: instream >> myObject and outstream << myObject, where specializations are entailed for the serial format of the iostreams (ASN.1, XML, etc.), as well as for the internal structure of the object. The C++ Toolkit deploys several object stream classes that specialize in various formats, and which know how to access and apply the type information that is associated with the serializable object.

The type information for each class is defined in a separate static CTypeInfo object, which can be accessed by all instances of that class. This is a very powerful device, which allows for the implementation of many features generally found only in languages which have built-in class reflection. Using the Toolkit's serializable objects will require some familiarity with the usage of this type information, and several sections of this manual cover these topics (see Runtime Object Type Information for a general discussion).

Locating and browsing serializable objects in the C++ Toolkit

The top level of the include/objects subtree is a set of subdirectories, where each subdirectory includes the public header files for a separately compiled library. Similarly, the src/objects subtree includes a set of subtrees containing the source files for these libraries. Finally, your build/objects directory will contain a corresponding set of build subtrees where these libraries are actually built.

If you checked out the entire C++ SVN tree, you may be surprised to find that initially, the include/objects subtrees are empty, and the subdirectories in the src/objects subtree contain only ASN.1 modules. This is because both the header files and source files are auto-generated from the ASN.1 specifications by the datatool program. As described in Working within the C++ source tree, you can build everything by running make all_r in the build directory.

Note: If you would like to have the objects libraries built locally, you **must** use the --with-objects flag when running the configure script.

You can also access the pre-generated serializable objects in the public area, using the source browsers to locate the objects you are particularly interested in. For example, if you are seeking the new class definition for the Bioseq struct defined in the C Toolkit, you can search for the CBioseq class, using either the LXR identifier search tool, or the Doxygen class hierarchy browser. Starting with the name of the data object as it appears in the ASN.1 module, two simple rules apply in deriving the new C++ class name:

- The one letter 'C' (for class) prefix should precede the ASN.1 name
- All hyphens ('-') should be replaced by underscores ('_')

For example, Seq-descr becomes CSeq_descr.

Base classes and user classes

The classes whose names are derived in this manner are called the user classes, and each also has a corresponding base class implementation. The name of the base class is arrived at by appending "_Base" to the user class name. Most of the user classes are empty wrapper classes that do not bring any new functionality or data members to the inherited base class; they are simply provided as a platform for development. In contrast, the base classes are **not** intended for public use (other than browsing), and should never be modified.

More generally, the base classes should *never* be instantiated or accessed directly in an application. The relation between the two source files and the classes they define reflects a general design used in developing the object libraries: the base class files are auto-generated by datatool according to the ASN.1 specifications in the src/objects directories; the inherited class files (the so-called user classes) are intended for developers who can extend these classes to support features above and beyond the ASN.1 specifications.

Many applications will involve a "tangled hierarchy" of these objects, reflecting the complexity of the real world data that they represent. For example, a CBioseq_set contains a list of CSeq_entry objects, where each CSeq_entry is, in turn, a choice between a CBioseq and a CBioseq_set.

Given the potential for this complexity of interactions, a critical design issue becomes how one can ensure that methods which may have been defined only in the user class will be available for all instances of that class. In particular, these instances may occur as contained elements of another object which is compiled in a different library. These inter-object dependencies are the motivation for the user classes. As shown in Figure 2, all references to external objects which occur inside the base classes, access external user classes, so as to include any methods which may be defined only in the user classes:

In most cases, adding non-virtual methods to a user class will **not** require re-compiling any libraries except the one which defines the modified object. Note however, that adding non-static data members and/or virtual methods to the user classes **will change** the class layouts, and in these cases only, will entail recompiling any external library objects which access these classes.

Adding methods to the user classes

Note: This section describes the steps currently required to add new methods to the user classes. It is subject to change, and there is no guarantee the material here is up-to-date. In general, it is not recommended practice to add methods to the user classes, unless your purpose is to extend these classes across all applications as part of a development effort.

The following topics are discussed in this section:

- Checking out source code, configuring the working environment, building the libraries.
- Adding methods

Checking out source code, configuring the working environment, building the libraries

- Create a working directory (e.g. Work) and check out the C++ tree to that directory:, using either SVN checkout or the shell script, svn_core.
- Configure the environment to work inside this tree using one of the configure scripts, according to the platform you will be working on. Be sure to include the --with-objects flag in invoking the configure script.
- Build the xncbi, xser and xser libraries, and run datatool to create the objects header and source files, and build all of the object module libraries:

```
# Build the core library
cd path_to_compile_dir/build/corelib
make
# Build the util library
cd path_to_compile_dir/build/util
make
# might as well build datatool and avoid possible version skew cd
path_to_compile_dir/build/serial make all_r
# needed for a few projects
cd path_to_compile_dir/build/connect
make
cd path_to_compile_dir/build/objects
make all_r
```

Here path_to_compile_dir is set to the compile work directory which depends on the compiler settings (e.g: ~/Work/internal/GCC-Debug). In addition to creating the header and source files, using make all_r (instead of just make) will build all the libraries. All libraries that are built are also copied to the lib dir, e.g.:~/Work/internal/c++/GCC-Debug/lib. Similarly, all executables (such as asn2asn) are copied to the bin dir, e.g.: ~/Work/internal/c++/GCC-Debug/bin.

You are now ready to edit the user class files and add methods.

Adding methods

As an example, suppose that we would like to add a method to the CSeq_inst class to calculate sequence length, e.g.:CSeq_inst::CalculateLength(). We begin by adding a declaration of this method to the public section of the user class definition in Seq_inst.hpp:

```
class CSeq_inst : public CSeq_inst_Base
{
public:
 CSeq_inst(void);
 ~CSeq_inst(void);
 static CSeq_inst* New(void)
 {
 return new CSeq_inst(eCanDelete);
 }
 int CalculateLength() const;
protected:
 CSeq_inst(ECanDelete);
};
```

and in the source file, Seq_inst.cpp, we implement

```
int CSeq_inst::CalculateLength() const
{
 // implementation goes here
}
```

These files are in the include/objects/seq and src/objects/seq subdirectories, respectively. Once you have made the modifications to the files, you need to recompile the seq library, libseq.a, i.e.:

```
cd path_to_compile_dir/GCC-Debug/build/objects/seq
make
```

Here path_to_compile_dir is set to the compile work directory which depends on the compiler settings (e.g: ~/Work/internal/GCC-Debug). The new method can now be invoked from within a CBioseq object as: myBioseq.GetInst().CalculateLength().

The key issue that determines whether or not you will need to rebuild any external libraries that use the modified user class involves the class layout in memory. All of the external libraries which reference the object refer to the class layout that existed prior to the changes you have made. Thus, if your modifications do **not** affect the class layout, you do not have to rebuild any external libraries. Changes that *do* affect memory mapping include:

- The addition of new, non-static data members
- The addition of virtual methods

If you have added either of the above to the user class, then you will need to identify all external objects which use your object, and recompile the libraries in which these objects are defined.

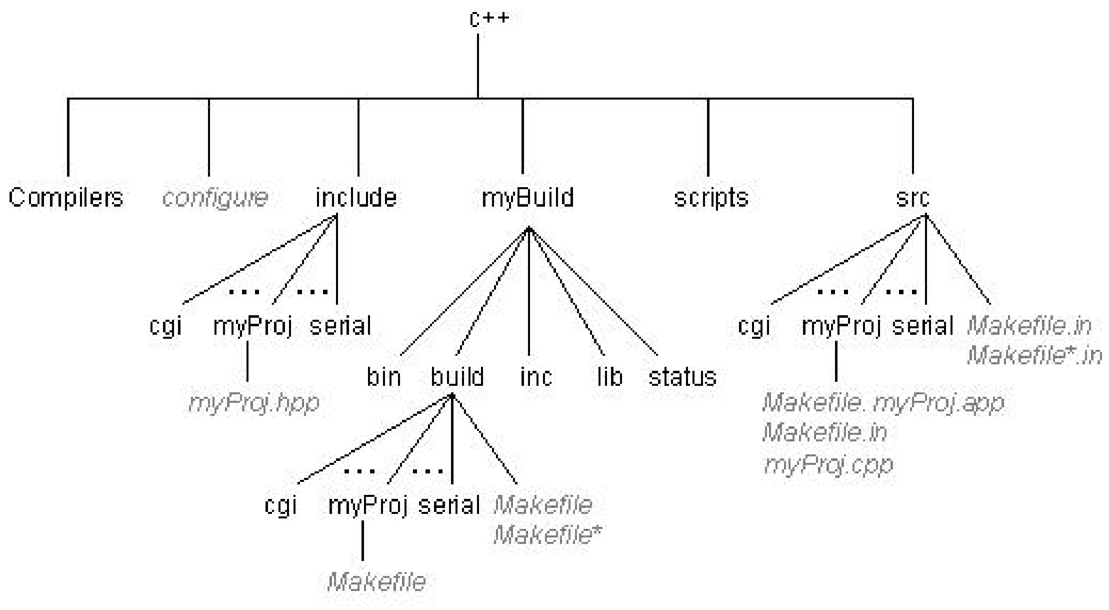


Figure 2

Figure 1. Meta makefiles and the makefiles they generate

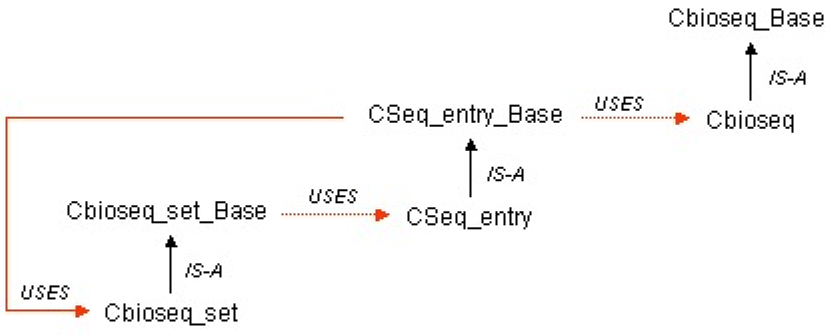


Figure 2. Example of complex relationships between base classes and user classes

Table 1. Build Directories

Directory	**Compiler**	**Version**
/netopt/ncbi_tools/c++/Debug/build	Sun Workshop	Debug
/netopt/ncbi_tools/c++/Debug64/build	Sun Workshop	Debug (64 bit)
/netopt/ncbi_tools/c++/DebugMT/build	Sun Workshop	Debug (Multi-thread safe)
/netopt/ncbi_tools/c++/Release/build	Sun Workshop	Release
/netopt/ncbi_tools/c++/ReleaseMT/build	Sun Workshop	Release (Multi-thread safe)
/netopt/ncbi_tools/c++/GCC-Debug/build	GCC	Debug
/netopt/ncbi_tools/c++/GCC-Release/build	GCC	Release

7: Programming Policies and Guidelines

Last Update: July 8, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discusses policies and guidelines for the development of NCBI software.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Choice of Language

C++ is typically the language of choice for C++ Toolkit libraries and applications. The policy for language choice in other areas within NCBI is:

- **C/C++** -- for high-performance standalone backend servers and CGIs, computationally intensive algorithms and large data flow processing tools used in production.
- **sh** or **bash** -- for primitive scripting.
- **Python** -- for advanced scripting. See its usage policy here.
- **Perl** -- for advanced scripting. The Python usage policy can be applied to Perl as well.
- **Java** -- for Eclipse programming and in-house QA and testing tools.

See the "Recommended programming and scripting languages" Wiki page for more information and updates to this policy. Send proposals for corrections, additions and extensions of the policy on language choice to the languages mailing list, languages@ncbi.nlm.nih.gov.

Source Code Conventions

This section contains C++ style guidelines, although many of these guidelines could also apply, at least in principle, to other languages. Adherence to these guidelines will promote uniform coding, better documentation, easy to read code, and therefore more maintainable code.

The following topics are discussed in this section:

- Public Domain Notice
- Naming Conventions
  - Type Names
  - Preprocessor Define/Macro
  - Function Arguments and Local Variables
  - Constants
  - Class and Structure Data Members (Fields)
  - Class Member Functions (Methods)
  - Module Static Functions and Data
  - Global ("extern") Functions and Data
- Name Prefixing and/or the Use of Namespaces
- Use of the NCBI Name Scope

Public Domain Notice

All NCBI-authored C/C++ source files **must** begin with a comment containing NCBI's public domain notice, shown below. Ideally (subject to the developer's discretion), so should any other publicly released source code and data (including scripting languages and data specifications).

```
/* $Id$
 *
===========================================================================
 *
 * PUBLIC DOMAIN NOTICE
 * National Center for Biotechnology Information
 *
 * This software/database is a "United States Government Work" under the
 * terms of the United States Copyright Act. It was written as part of
 * the author's official duties as a United States Government employee and
 * thus cannot be copyrighted. This software/database is freely available
 * to the public for use. The National Library of Medicine and the U.S.
 * Government have not placed any restriction on its use or reproduction.
 *
 * Although all reasonable efforts have been taken to ensure the accuracy
 * and reliability of the software and data, the NLM and the U.S.
 * Government do not and cannot warrant the performance or results that
 * may be obtained by using this software or data. The NLM and the U.S.
 * Government disclaim all warranties, express or implied, including
 * warranties of performance, merchantability or fitness for any particular
 * purpose.
 *
 * Please cite the author in any work or product based on this material.
 *
 *
===========================================================================
 */
```

If you have questions, please email to cpp-core@ncbi.nlm.nih.gov.

Naming Conventions

Table 1. Naming Conventions

SYNOPSIS	EXAMPLE
Type Names	

***C**ClassTypeName*	class CMyClass { ..... };
***I**InterfaceName*	class IMyInterface { ..... };
***S**StructTypeName*	struct SMyStruct { ..... };
***U**UnionTypeName*	union UMyUnion { ..... };
***E**EnumTypeName*	enum EMyEnum { ..... };
***F**FunctionTypeName*	typedef int (*FMyFunc)(void);
***P**PredicateName*	struct PMyPred { bool operator() (.... , ...); };
***T**AuxiliaryTypedef* (*)	typedef map<int,string> TMyMapIntStr;
T**Iterator_**I	typedef list<int>::iterator TMyList_I;
T**ConstIterator_**CI	typedef set<string>::const_iterator TMySet_CI;
***N**Namespace* (see also)	namespace NMyNamespace { ..... }
Preprocessor Define/Macro	
MACRO_NAME	#define MY_DEFINE 12345
macro_arg_name	#define MY_MACRO(x, y) (((x) + 1) < (y))
Function Arguments and Local Variables	

func_local_var_name	void MyFunc(int foo, const CMyClass& a_class)
	{
	   size_t foo_size;
	   int bar;

Constants	
***k**ConstantName*	const int kMyConst = 123;

***e**EnumValueName*	enum EMyEnum {
	   eMyEnum_1 = 11,
	   eMyEnum_2 = 22,
	   eMyEnum_3 = 33
	};

***f**FlagValueName*	enum EMyFlags {
	   fMyFlag_1 = (1<<0), ///< = 0x1 (describe)
	   fMyFlag_2 = (1<<1), ///< = 0x2 (describe)
	   fMyFlag_3 = (1<<2) ///< = 0x4 (describe)
	};
	typedef int TMyFlags; ///< holds bitwise OR of "EMyFlags"

Class and Structure Data Members (Fields)	
***m**_ClassMemberName*	class C { short int m_MyClassData; };
struct_field_name	struct S { int my_struct_field; };
***sm**_ClassStaticMemberName*	class C { static double sm_MyClassStaticData; };
Class Member Functions (Methods)	
ClassMethod	bool MyClassMethod(void);
***x**_ClassPrivateMethod*	int x_MyClassPrivateMethod(char c);
Module Static Functions and Data	
***s**_StaticFunc*	static char s_MyStaticFunc(void);

***s**_StaticVar*	static int s_MyStaticVar;
Global (*"extern"*) Functions and Data	
***g**_GlobalFunc*	double g_MyGlobalFunc(void);
***g**_GlobalVar*	short g_MyGlobalVar;

(*) The auxiliary typedefs (like ***T**AuxiliaryTypedef*) are usually used for an ad-hoc type mappings (especially when using templates) and not when a real type definition takes place.

Name Prefixing and/or the Use of Namespaces

In addition to the above naming conventions that highlight the nature and/or the scope of things, one should also use prefixes to:

- avoid name conflicts
- indicate the package that the entity belongs to

For example, if you are creating a new class called "Bar" in package "Foo" then it is good practice to name it "CFooBar" rather than just "CBar". Similarly, you should name new constants like "kFooSomeconst", new types like "TFooSometype", etc.

Use of the NCBI Name Scope

<ncbistl.hpp>

All NCBI-made "core" API code must be put into the "ncbi::" namespace. For this purpose, there are two preprocessor macros, BEGIN_NCBI_SCOPE and END_NCBI_SCOPE, that must enclose **all** NCBI C++ API code -- both declarations and definitions (see examples). Inside these "brackets", all "std::" and "ncbi::" scope prefixes can (and must!) be omitted.

For code that does not define a new API but merely **uses** the NCBI C++ API, there is a macro USING_NCBI_SCOPE; (semicolon-terminated) that brings all types and prototypes from the "std::" and "ncbi::" namespaces into the current scope, eliminating the need for the "std::" and "ncbi::" prefixes.

Use macro NCBI_USING_NAMESPACE_STD; (semicolon-terminated) if you want to bring all types and prototypes from the "std::" namespace into the current scope, without bringing in anything from the "ncbi::" namespace.

Use of Include Directives

If a header file is in the local directory or not on the INCLUDE path, use quotes in the include directive (e.g. #include "foo.hpp"). In all other cases use angle brackets (e.g. #include <bar/foo.hpp>).

In general, if a header file is commonly used, it must be on the INCLUDE path and therefore requires the bracketed form.

Code Indentation and Bracing

4-space indentation only! Tabulation symbol **must not** be used for indentation.

Try not to cross the "standard page boundary" of **80** symbols.

In if, for, while, do, switch, case, etc. and type definition statements:

```
if (...) {
 .....;
} else if (...) {
 .....;
} else {
 .....;
}

if (...) {
 .....;
}
else if (...) {
 .....;
}
else {
 .....;
}

for (...; ...; ...) {
 .....;
}

while (...) {
 .....;
}

do {
 .....;
}
while (...);

switch (...) {
case ...: {
 .....;
 break;
}
} // switch

struct|union|enum <[S|U|E]TypeName> {
 .....;
};

class | struct | union <[C|I|P|S|U]TypeName>
{
 .....;
};

try {
 .....;
}
catch (exception& e) {
 .....;
}
```

Class Declaration

Class declarations should be rich in Doxygen-style comments. This will increase the value of the Doxygen-based API documentation.

```
/// @file FileName
/// Description of file -- note that this is _required_ if you want
/// to document global objects such as typedefs, enums, etc.

///////////////////////////////////////////////////////////////////////
///
/// CFooClass
///
/// Brief description of class (or class template, struct, union) --
/// it must be followed by an empty comment line.
///
/// A detailed description of the class -- it follows after an empty
/// line from the above brief description. Note that comments can
/// span several lines and that the three /// are required.

class CFooClass
{
public:
 // Constructors and Destructor:

 /// A brief description of the constructor.
 ///
 /// A detailed description of the constructor.
 CFooClass(const char* init_str = NULL); ///< describe parameter here

 /// A brief description for another constructor.
 CFooClass(int init_int); ///< describe parameter here

 ~CFooClass(void); // Usually needs no Doxygen-style comment.

 // Members and Methods:

 /// A brief description of TestMe.
 ///
 /// A detailed description of TestMe. Use the following when
 /// parameter descriptions are going to be long, and you are
 /// describing a complex method:
 /// @param foo
 /// An int value meaning something.
 /// @param bar
 /// A constant character pointer meaning something.
 /// @return
 /// The TestMe() results.
 /// @sa CFooClass(), ~CFooClass() and TestMeToo() - see also.
 float TestMe(int foo, const char* bar);

 /// A brief description of TestMeToo.
 ///
 /// Details for TestMeToo. Use this style if the parameter
 /// descriptions are going to be on one line each:
 /// @sa TestMe()
 virtual void TestMeToo
 (char par1, ///< short description for par1
 unsigned int par2 ///< short description for par2
 ) = 0;

 /// Brief description of a function pointer type
 /// (note that global objects like this will not be documented
 /// unless the file itself is documented with the @file command).
 ///
 /// Detailed description of the function pointer type.
 typedef char* (*FHandler)
 (int start, ///< argument description 1 -- what start means
 int stop ///< argument description 2 -- what stop means
 );

 // (NOTE: The use of public data members is
 // strictly discouraged!
 // If used they should be well documented!)
 /// Describe public member here, explain why it’s public.
 int m_PublicData;

protected:
 /// Brief description of a data member -- notice no details are
 /// given here since a brief description is adequate.
 double m_FooBar;

 /// Brief function description here.
 /// Detailed description here. More description.
 /// @return Return value description here.
 static int ProtectedFunc(char ch); ///< describe parameter here

private:
 /// Brief member description here.
 /// Detailed description here. More description.
 int m_PrivateData;

 /// Brief static member description here.
 static int sm_PrivateStaticData;

 /// Brief function description here.
 /// Detailed description here. More description.
 /// @return Return value description here.
 double x_PrivateFunc(int some_int = 1); ///< describe parameter here
 // Friends
 friend bool SomeFriendFunc(void);
 friend class CSomeFriendClass;

 // Prohibit default initialization and assignment
 // -- e.g. when the member-by-member copying is dangerous.

 /// This method is declared as private but is not
 /// implemented to prevent member-wise copying.
 CFooClass(const CFooClass&);

 /// This method is declared as private but is not
 /// implemented to prevent member-wise copying.
 CFooClass& operator= (const CFooClass&);
};
```

Function Declaration

Doxygen-style comments for functions should describe what the function does, its parameters, and what it returns.

For global function declarations, put all Doxygen-style comments in the header file. Prefix global functions with g_.

```
/// A brief description of MyFunc2.
///
/// Explain here what MyFunc2() does.
/// @return explain here what MyFunc2() returns.
bool g_MyFunc2
(double arg1, ///< short description of "arg1"
 string* arg2, ///< short description of "arg2"
 long arg3 = 12 ///< short description of "arg3"
 );
```

Function Definition

Doxygen-style comments are not needed for member function definitions or global function definitions because their comments are put with their declarations in the header file.

For static functions, put all Doxygen-style comments immediately before the function definition. Prefix static functions with s_.

```
bool g_MyFunc2
(double arg1,
 string* arg2,
 long arg3
 )
{
 .......
 .......
 return true;
}
```

```
/// A brief description of s_MyFunc3.
///
/// Explain here what s_MyFunc3() does.
/// @return explain here what s_MyFunc3() returns.
static long s_MyFunc3(void)
{
 .......
 .......
}
```

Use of Whitespace

As the above examples do not make all of our policies on whitespace clear, here are some explicit guidelines:

- When reasonably possible, use spaces to align corresponding elements vertically. (This overrides most of the rules below.)
- Leave one space on either side of most binary operators, and two spaces on either side of boolean && and ||.
- Put one space between the names of flow-control keywords and macros and their arguments, but no space after the names of functions except when necessary for alignment.
- Leave two spaces after the semicolons in for (...; ...; ...).
- Leave whitespace around negated conditions so that the ! stands out better.
- Leave two blank lines between function definitions.

Standard Header Template

A standard header template file, header_template.hpp, has been provided in the include/common directory that can be used as a template for creating header files. This header file adheres to the standards outlined in the previous sections and uses a documentation style for files, classes, methods, macros etc. that allows for automatic generation of documentation from the source code. It is strongly suggested that you obtain a copy of this file and model your documentation using the examples in that file.

Doxygen Comments

Doxygen is an automated API documentation tool. It relies on special comments placed at appropriate places in the source code. Because the comments are in the source code near what they document, the documentation is more likely to be kept up-to-date when the code changes. A configuration and parsing system scans the code and creates the desired output (e.g. HTML).

Doxygen documentation is a valuable tool for software developers, as it automatically creates comprehensive cross-referencing of modules, namespaces, classes, and files. It creates inheritance diagrams, collaboration diagrams, header dependency graphs, and documents each class, struct, union, interface, define, typedef, enum, function, and variable (see the NCBI C++ Toolkit Doxygen browser). However, developers must write meaningful comments to get the most out of it.

Doxygen-style comments are essentially extensions of C/C++ comments, e.g. the use of a triple-slash instead of a double-slash. Doxygen-style comments refer to the entity following them by default, but can be made to refer to the entity preceding them by appending the '<' symbol to the comment token (e.g. '///<').

Doxygen commands are keywords within Doxygen comments that are used during the document generation process. Common commands are @param, @return, and @sa (i.e. 'see also').

Please do not use superfluous comments, such as '/// Destructor'. Especially do not use the same superfluous comment multiple times, such as using the same '/// Constructor' comment for different constructors!

Please see the Doxygen manual for complete usage information. More information can also be found in the chapter on Toolkit browsers.

C++ Guidelines

This section discusses the following topics:

Introduction to Some C++ and STL Features and Techniques

C++ Implementation Guide

Use of STL (Standard Template Library)

Use the Standard Template Library (STL), which is part of ANSI/ISO C++. It'll make programming easier, as well as make it easier for others to understand and maintain your code.

Use of C++ Exceptions

- Exceptions are useful. However, since exceptions unwind the stack, you must be careful to destroy all resources (such as memory on the heap and file handles) in every intermediate step in the stack unwinding. That means you must always catch exceptions, even those you don't handle, and delete everything you are using locally. In most cases it's very convenient and safe to use the auto_ptr template to ensure the freeing of temporary allocated dynamic memory for the case of exception.
- Avoid using exception specifications in function declarations, such as:

```
void foo(void) throw ();

void bar(void) throw (std::exception);
```

Design

- Use abstract base classes. This increases the reusability of code. Whether a base class should be abstract or not depends on the potential for reuse.
- Don't expose class member variables, rather expose member functions that manipulate the member variables. This increases reusability and flexibility. For example, this frees you from having the string in-process -- it could be in another process or even on another machine.
- Don't use multiple inheritance (i.e. class A: public B, public C {}) unless creating interface instead of implementation. Otherwise, you'll run into all sorts of problems with conflicting members, especially if someone else owns a base class. The best time to use multiple inheritance is when a subclass multiply inherits from abstract base classes with only pure virtual functions.

NOTE: Some people prefer the Unified Modelling Language to describe the relationships between objects.

Make Your Code Readable

Use NULL instead of 0 when passing a null pointer. For example:

```
MyFunc(0,0); // Just looking at this call, you can’t tell which
 // parameter might be an int and which might be
 // a pointer.

MyFunc(0,NULL); // When looking at this call, it’s pretty clear
 // that the first parameter is an int and
 // the second is a pointer.
```

Avoid using bool as a type for function arguments. For example, this might be hard to understand:

```
// Just looking at this call, you can’t tell what
// the third parameter means:
CompareStrings(s1, s2, true);
```

Instead, create a meaningful enumerated type that captures the meaning of the parameter. For example, try something like this:

```
///////////////////////////////////////////////////////////////////////
///
/// ECaseSensitivity --
///
/// Control case-sensitivity of string comparisons.
///
enum ECaseSensitivity {
 eCaseSensitive, ///< Consider case when comparing.
 eIgnoreCase ///< Don’t consider case when comparing.
};
```

```
.....

/// Brief description of function here.
/// @return
/// describe return value here.
int CompareStrings
(const string& s1, ///< First string.
 const string& s2, ///< Second string.
 ECaseSensitivity comp_case); ///< Controls case-sensitivity
 ///< of comparisons.

.....

// This call is more understandable because the third parameter
// is an enum constant rather than a bool constant.
CompareStrings(s1, s2, eIgnoreCase);
```

As an added benefit, using an enumerated type for parameters instead of bool gives you the ability to expand the enumerated type to include more variants in the future if necessary - without changing the parameter type.

C++ Tips and Tricks

- Writing something like map<int, int, less<int>> will give you weird errors; instead write map<int, int, less<int> >. This is because >> is reserved word.
- Do use pass-by-reference. It'll cut down on the number of pointer related errors.
- Use const (or enum) instead of #define when you can. This is much easier to debug.
- Header files should contain what they contain in C along with classes, const's, and in-line functions.

See the C++ FAQ

Standard Template Library (STL)

The STL is a library included in ANSI/ISO C++ for stream, string, and container (linked lists, etc.) manipulation.

STL Tips and Tricks

end() does not return an iterator to the last element of a container, rather it returns a iterator just beyond the last element of the container. This is so you can do constructs like

```
for (iter = container.begin(); iter != container.end(); iter++)
```

If you want to access the last element, use "--container.end()". Note: If you use this construct to find the last element, you must first ensure that the container is not empty, otherwise you could get corrupt data or a crash.

The C++ Toolkit includes macros that simplify iterating. For example, the above code simplifies to:

```
ITERATE(Type, iter, container)
```

For more info on ITERATE (and related macros), see the ITERATE macros section.

Iterator misuse causes the same problems as pointer misuse. There are versions of the STL that flag incorrect use of iterators.

Iterators are guaranteed to remain valid after insertion and deletion from list containers, but not vector containers. Check to see if the container you are using preserves iterators.

If you create a container of pointers to objects, the objects are not destroyed when the container is destroyed, only the pointers are. Other than maintaining the objects yourself, there are several strategies for handling this situation detailed in the literature.

If you pass a container to a function, don't add a local object to the container. The local variable will be destroyed when you leave the function.

C++/STL Pitfalls and Discouraged/Prohibited Features

STL and Standard C++ Library's Bad Guys

Non-Standard STL Classes

- Don't use the rope class from some versions of the STL. This is a non-standard addition. If you have questions about what is/isn't in the standard library, consult the C++ standards.
- The NCBI C++ Toolkit includes hash_map, hash_multimap, hash_set, and hash_multiset classes (from headers <corelib/hash_map.hpp> and <corelib/hash_set.hpp>). These classes are more portable than, and should be used instead of, the STL's respective hash_* classes.

C++ Bad Guys

Operator Overload

Do not use operator overloading for the objects where they have unnatural or ambiguous meaning. For example, the defining of operator==() for your class "CFoo" so that there exist { CFoo a,b,c; } such that (a == b) and (b == c) are true while (a == c) is false would be a very bad idea. It turns out that otherwise, especially in large projects, people have different ideas of what an overloaded operator means, leading to all sorts of bugs.

Assignment and Copy Constructor Overload

Be advised that the default initialization {CFoo foo = bar;} and assignment {CFoo foo; ...; foo = bar;} do a member-by-member copying. This is not suitable and can be dangerous sometimes. And if you decide to overwrite this default behavior by your own code like:

```
class CFoo {
 // a copy constructor for initialization
 CFoo(const CFoo& bar) { ... }
 // an overloaded assignment(=) operator
 CFoo& operator=(const CFoo& bar) { if (&bar != this) ... }
};
```

it is **extremely important** that:

- **both** copy constructor and overloaded assignment be defined
- they have **just the same** meaning; that is {CFoo foo = bar;} is equivalent to {CFoo foo; foo = bar;}
- there is a check to prevent self-assignment in your overloaded assignment operator

In many cases when you don't want to have the assignment and copy constructor at all, just add to your class something like:

```
class CFoo {
 .............................
private:
 // Prohibit default initialization and assignment
 CFooClass(const CFooClass&);
 CFooClass& operator=(const CFooClass&);
};
```

Omitting "void" in a No-Argument Function Prototype

Do not omit "void" in the prototype of a function without arguments (e.g. always write "int f (void)" rather than just "int f()").

Do Not Mix malloc and new

On some platforms, malloc and new may use completely different memory managers, so never "free()" what you created using "new" and never "delete" what you created using "malloc()". Also, when calling C code from C++ **always** allocate any structs or other items using "malloc ()". The C routine may use "realloc()" or "free()" on the items, which can cause memory corruption if you allocated using "new."

Source Code Repositories

The following Subversion repositories have been set up for general use within NCBI:

Repository	**Purpose**
toolkit	C++ Toolkit (core and internal) development
gbench	GUI / GBENCH
staff	individuals' projects (not parts of any official projects)
misc_projects	projects not falling into any of the other categories

Note for NCBI developers: Using these repositories has the additional advantages that they are:

- backed up;
- partially included in automated builds and tests (along with reporting via email and on the intranet) on multiple platforms and compiler configurations; and

- integrated with JIRA and FishEye.

Testing

Unit testing using the Boost Unit Test Framework is strongly encouraged for libraries. Within NCBI, unit tests can be incorporated into the nightly automated testsuite by using the CHECK_CMD macro in the makefile. All testsuite results are available on the testsuite web page. Users can also be automatically emailed with build and/or test results by using the WATCHERS macro. Please see the chapter on Using the Boost Unit Test Framework for more information.

Applications should also be tested, and shell scripts are often convenient for this purpose.

Data files used for testing purposes should be checked into SVN with the source code unless they are very large.

Part 3: C++ Toolkit Library Reference

Part 3 discusses the the core library and the different specialized libraries such as the connection, database API, CGI, HTML, Serial, Util, GUI etc. The following is a list of chapters in this part:

The **NCBI C++ Toolkit**

8: Portability, Core Functionality and Application Framework

Last Update: July 9, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

- **CORELIB library** xncbi:include | src

The CORELIB provides a portable low-level API and many useful application framework classes for argument processing, diagnostics, environment interface, object and reference classes, portability definitions, portable exceptions, stream wrappers, string manipulation, threads, etc.

This chapter provides reference material for many of CORELIB's facilities. For an overview of CORELIB, please refer to the CORELIB section in the introductory chapter on the C++ Toolkit.

Note: The CORELIB must be linked to every executable that uses the NCBI C++ Toolkit!

- **UTIL library** xutil:include | src

The UTIL module is a collection of useful classes which can be used in more then one application. This chapter provides reference material for many of UTIL's facilities. For an overview of the UTIL module please refer to the UTIL section in the introductory chapter on the C++ Toolkit.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Demo Cases [src/sample/app/basic]

Writing a Simple Application

This section discusses how to write a simple application using the CNcbiApplication and related class. A conceptual understanding of the uses of the CNcbiApplication and related classes is presented in the introductory chapter on the C++ Toolkit.

This section discusses the following topics:

- Basic Classes of the NCBI C++ Toolkit
- Creating a Simple Application

- Inside the NCBI Application Class

Note: The C++ Toolkit can also be used from a third party application framework.

NCBI C++ Toolkit Application Framework Classes

The following five fundamental classes form the foundation of the C++ Toolkit Application Framework:

- CNcbiApplication
- CNcbiArguments (see also CArgDescriptions, CArgs, ...)
- CNcbiEnvironment
- CNcbiRegistry
- CNcbiDiag

Each of these classes is discussed in the following sections:

CNcbiApplication

CNcbiApplication is an abstract class used to define the basic functionality and behavior of an NCBI application. Because this application class effectively supersedes the C-style main() function, minimally, it must provide the same functionality, i.e.:

- a mechanism to execute the actual application
- a data structure for holding program command-line arguments ("argv")
- a data structure for holding environment variables

In addition, the application class provides the same features previously implemented in the C Toolkit, namely:

- mechanisms for specifying where, when, and how errors should be reported
- methods for reading, accessing, modifying, and writing information in the application's registry (configuration) file
- methods to describe, and then automatically parse, validate, and access program command-line arguments and to generate the USAGE message

The mechanism to execute the application is provided by CNcbiApplication's member function Run(), for which you must write your own implementation. The Run() function will be automatically invoked by CNcbiApplication::AppMain(), after it has initialized its CNcbiArguments, CNcbiEnvironment, CNcbiRegistry, and CNcbiDiag data members.

CNcbiArguments

The CNcbiArguments class provides a data structure for holding the application's command-line arguments, along with methods for accessing and modifying these. Access to the argument values is implemented using the built-in [ ] operator. For example, the first argument in argv (following the program name) can be retrieved using the CNcbiApplication::GetArguments() method:

```
string arg1_value = GetArguments()[1];
```

Here, GetArguments() returns the CNcbiArguments object, whose argument values can then be retrieved using the [ ] operator. Four additional CNcbiArguments member functions support retrieval and modification of the program name (initially argv[0]). A helper class, described in Processing Command-Line Arguments, supports the generation of USAGE messages and the imposition of constraints on the values of the input arguments.

In addition to the CNcbiArguments class, there are other related classes used for argument processing. The CArgDescriptions and CArgDesc classes are used for describing unparsed arguments; CArgs and CArgValue for parsed argument values; CArgException and CArgHelpException for argument exceptions; and CArgAllow, CArgAllow_{Strings, ..., Integers, Doubles} for argument constraints. These classes are discussed in the section on Processing Command-Line Arguments.

When using the C++ Toolkit on the Mac OS, you can specify command-line arguments in a separate file with the name of your executable and ".args" extension. Each argument should be on a separate line (see Table 1).

CNcbiEnvironment

The CNcbiEnvironment class provides a data structure for storing, accessing, and modifying the environment variables accessed by the C library routine getenv().

The following describes the public interface to the CNcbiEnvironment:

```
class CNcbiEnvironment
{
public:
 /// Constructor.
 CNcbiEnvironment(void);
 /// Constructor with the envp parameter.
 CNcbiEnvironment(const char* const* envp);
 /// Destructor.
 virtual ~CNcbiEnvironment(void);
 /// Reset environment.
 ///
 /// Delete all cached entries, load new ones from "envp" (if not NULL).
 void Reset(const char* const* envp = 0);
 /// Get environment value by name.
 ///
 /// If environmnent value is not cached then call "Load(name)" to load
 /// the environmnent value. The loaded name/value pair will then be
 /// cached, too, after the call to "Get()".
 const string& Get(const string& name) const;
};
```

For example, to retrieve the value of environment variable PATH:

```
string arg1_value = GetEnvironment().Get("PATH");
```

In this example, the GetEnvironment() is defined in the CNcbiApplication class and returns the CNcbiEnvironment object for which the Get() method is called with the environment variable PATH.

To delete all of the cached entries and reload new ones from the environment pointer (envp), use the CNcbiEnvironment::Reset() method.

CNcbiRegistry

Complete details for the CNcbiRegistry can be found in the section on The CNcbiRegistry Class.

CNcbiDiag

The CNcbiDiag class implements much of the functionality of the NCBI C++ Toolkit error-processing mechanisms; however, it is not intended to be used directly. Instead, use the `{ERR|LOG}_POST*` and `_TRACE` macros. See the sections on Diagnostic Streams and Message Posting for related information.

Creating a Simple Application

This section discusses the following topics:

- Unix-like Systems
- MS Windows
- Discussion of the Sample Application

Unix-like Systems

Using the new_project shell script, create a new project example:

```
new_project example app
```

This will create:

1. the project folder -- example
2. the source file -- example.cpp
3. the makefiles -- Makefile, Makefile.builddir, Makefile.in, Makefile.example.app, Makefile.example_app, Makefile.out

Then build the project and run the application:

```
cd example; make; ./example
```

MS Windows

Using the new_project shell script, create a new project example:

```
new_project example app
```

This will create:

1. the project folder -- example
2. the source file -- example\src\example\basic_sample.cpp (the source file name is always basic_sample.cpp, regardless of the project name)
3. the MSVC project file -- example\compilers\msvc1000_prj\static\build\example \example.exe.vcproj
4. the MSVC solution file -- example\compilers\msvc1000_prj\static\build\example.sln
5. a project makefile -- example\src\example\Makefile.example.app
6. other folders and files needed for building under Windows

Note: If you prefer to have your source file name match your project name, you can achieve that by making the following edits before opening Visual Studio (for basic application projects, that is - other project types might require more edits):

1. Rename the source file from example\src\example\basic_sample.cpp to example.cpp.
2. Edit the MSVC project file example\compilers\msvc1000_prj\static\build\example \example.exe.vcproj and replace "basic_sample" with "example".

3. Edit the project makefile example\src\example\Makefile.example.app and replace "basic_sample" with "example".

Then open the solution file example\compilers\msvc1000_prj\static\build\example.sln with MSVS and:

1. Build the -CONFIGURE- project (reloading the project when prompted).
2. Build the project and run the application.

Discussion of the Sample Application

In the sample application above:

1. There is an application class derived from CNcbiApplication, which overrides the purely virtual function Run() as well as the initialization (Init()) and cleanup (Exit()) functions:

```
class CSampleBasicApplication : public CNcbiApplication
{
private:
 virtual void Init(void);
 virtual int Run(void);
 virtual void Exit(void);
};
```

2. The program's main function creates an object of the application class and calls its AppMain () function:

```
int main(int argc, const char* argv[])
{
 // Execute main application function
 return CSampleBasicApplication().AppMain(argc, argv);
}
```

3. The application's initialization function creates an argument descriptions object, which describes the expected command-line arguments and the usage context:

```
void CSampleBasicApplication::Init(void)
{
 // Create command-line argument descriptions
 auto_ptr<CArgDescriptions> arg_desc(new CArgDescriptions);

 // Specify USAGE context
 arg_desc->SetUsageContext(GetArguments().GetProgramBasename(),
 "CArgDescriptions demo program");
 ...
 // Setup arg.descriptions for this application
 SetupArgDescriptions(arg_desc.release());
}
```

4. The application's Run() function prints those arguments into the standard output stream or in a file.

More realistic examples of applications that use the NCBI C++ Toolkit are available.

Inside the NCBI Application Class

Here is a somewhat simplified view of the application's class definition:

```
class CNcbiApplication
{
public:
 /// Main function (entry point) for the NCBI application.
 ///
 /// You can specify where to write the diagnostics
 /// to (EAppDiagStream), and where to get
 /// the configuration file (LoadConfig()) to load
 /// to the application registry (accessible via GetConfig()).
 ///
 /// Throw exception if:
 /// - not-only instance
 /// - cannot load explicitly specified config.file
 /// - SetupDiag() throws an exception
 ///
 /// If the application name is not specified, a default of "ncbi" is used.
 /// Certain flags such as -logfile, -conffile, and -version are
 /// special, so AppMain() processes them separately.
 /// @return
 /// Exit code from Run(). Can also return a non-zero value if
 /// the application threw an exception.
 /// @sa
 /// Init(), Run(), Exit()
 int AppMain(int argc, const char **argv, const char **envp,
 EAppDiagStream diag, const char* config, const string& name);

 /// Initialize the application.
 ///
 /// The default behavior of this is "do nothing". If you have
 /// special initialization logic that needs to be performed,
 /// then you must override this method with your own logic.
 virtual void Init(void);

 /// Run the application.
 ///
 /// It is defined as a pure virtual method -- so you must(!)
 /// supply theRun() method to implement the
 /// application-specific logic.
 /// @return
 /// Exit code.
 virtual int Run(void) = 0;

 /// Cleanup on application exit.
 ///
 /// Perform cleanup before exiting. The default behavior of this
 /// is "do nothing". If you have special cleanup logic that needs
 /// to be performed, then you must override this method with
 /// your own logic.
 virtual void Exit(void);

 /// Get the application's cached unprocessed command-line
 /// arguments.
 const CNcbiArguments& GetArguments(void) const;

 /// Get parsed command-line arguments.
 ///
 /// Get command-line arguments parsed according to the arg
 /// descriptions set by SetArgDescriptions(). Throw exception
 /// if no descriptions have been set.
 /// @return
 /// The CArgs object containing parsed cmd.-line arguments.
 /// @sa
 /// SetArgDescriptions().
 const CArgs& GetArgs(void) const;

 /// Get the application's cached environment.
 const CNcbiEnvironment& GetEnvironment(void) const;

 /// Get the application's cached configuration parameters.
 const CNcbiRegistry& GetConfig(void) const;

 /// Flush the in-memory diagnostic stream (for "eDS_ToMemory"
 /// case only).
 ///
 /// In case of "eDS_ToMemory", the diagnostics is stored in
 /// the internal application memory buffer ("m_DiagStream").
 /// Call this function to dump all the diagnostics to stream "os" and
 /// purge the buffer.
 /// @param os
 /// Output stream to dump diagnostics to. If it is NULL, then
 /// nothing will be written to it (but the buffer will still be
 /// purged).
 /// @param close_diag
 /// If "close_diag" is TRUE, then also destroy "m_DiagStream".
 /// @return
 /// Total number of bytes actually written to "os".
 SIZE_TYPE FlushDiag(CNcbiOstream* os, bool close_diag = false);

 /// Get the application's "display" name.
 ///
 /// Get name of this application, suitable for displaying
 /// or for using as the base name for other files.
 /// Will be the 'name' argument of AppMain if given.
 /// Otherwise will be taken from the actual name of the
 /// application file or argv[0].
 string GetProgramDisplayName(void) const;

protected:
 /// Setup application specific diagnostic stream.
```

```
    ///
    /// Called from SetupDiag when it is passed the eDS_AppSpecific
    /// parameter. Currently, this calls SetupDiag(eDS_ToStderr) to setup
    /// diagonistic stream to the std error channel.
    /// @return
    /// TRUE if successful, FALSE otherwise.
    virtual bool SetupDiag_AppSpecific(void);

    /// Load configuration settings from the configuration file to
    /// the registry.
    ///
    /// Load (add) registry settings from the configuration file
    /// specified as the "conf" arg passed to AppMain(). The
    /// "conf" argument has the following special meanings:
    /// - NULL -- don't even try to load the registry from any
    /// file at all;
    /// - non-empty -- if "conf" contains a path, then try to load
    /// from theconf.file of name "conf" (only!). Else -
    /// see NOTE.
    /// TIP: if the path is not fully qualified then:
    /// if it starts from "../" or "./" -- look
    /// starting from the current working dir.
    /// - empty -- compose conf.file name from the application
    /// name plus ".ini". If it does not match an existing
    /// file, then try to strip file extensions, e.g., for
    /// "my_app.cgi.exe" -- try subsequently:
    /// "my_app.cgi.exe.ini", "my_app.cgi.ini",
    /// "my_app.ini".
    ///
    /// NOTE:
    /// If "conf" arg is empty or non-empty, but without path, then
    /// config file will be sought for in the following order:
    /// - in the current work directory;
    /// - in the dir defined by environment variable "NCBI";
    /// - in the user home directory;
    /// - in the program dir.
    ///
    /// Throw an exception if "conf" is non-empty, and cannot open
    /// file.
    /// Throw an exception if file exists, but contains invalid entries.
    /// @param reg
    /// The loaded registry is returned via the reg parameter.
    /// @param conf
    /// The configuration file to loaded the registry entries from.
    /// @return
    /// TRUE only if the file was non-NULL, found and successfully
    /// read.
    virtual bool LoadConfig(CNcbiRegistry& reg, const string* conf);
    .............
};
```

The AppMain() function is also inherited from the parent class. Although this function accepts up to six arguments, this example passes only the first two, with missing values supplied by defaults. The remaining four arguments specify:

- (#3) a NULL-terminated array of '\0'-terminated character strings from which the environment variables can be read
- (#4) how to setup a diagnostic stream for message posting
- (#5) the name of a .ini configuration file (see above for its default location)
- (#6) a program name (to be used in lieu of argv[0])

AppMain() begins by resetting the internal data members with the actual values provided by the arguments of main(). Once these internal data structures have been loaded, AppMain() calls the virtual functions Init(), Run(), and Exit() in succession to execute the application.

The Init() and Exit() virtual functions are provided as places for developers to add their own methods for specific applications. If your application does not require additional initialization/ termination, these two functions can be left empty or simply not implemented. The Run() method carries out the main work of the application.

The FlushDiag() method is useful if the diagnostic stream has been set to eDS_toMemory, which means that diagnostic messages are stored in an internal application memory buffer. You can then call FlushDiag() to output the stored messages on the specified output stream. The method will also return the number of bytes written to the output stream. If you specify NULL for the output stream, the memory buffers containing the diagnostic messages will be purged but not deallocated, and nothing will be written to the output. If the close_diag parameter to FlushDiag() is set to true, then the memory buffers will be deallocated (and purged, of course).

The GetProgramDisplayName() method simply returns the name of the running application, suitable for displaying in reports or for using as the base name for building other related file names.

The protected virtual function SetupDiag_AppSpecific() can be redefined to set up error posting specific for your application. SetupDiag_AppSpecific() will be called inside AppMain () by default if the error posting has not been set up already. Also, if you pass diag = eDS_AppSpecific to AppMain(), then SetupDiag_AppSpecific() will be called for sure, regardless of the error posting setup that was active before the AppMain() call.

The protected virtual function LoadConfig() reads the program's .ini configuration file to load the application's parameters into the registry. The default implementation of LoadConfig() expects to find a configuration file named <program_name>.ini and, if the DIAG_POST_LEVEL environment variable is set to "Info", it will generate a diagnostics message if no such file is found.

The NCBI application (built by deriving from CNcbiApplication) throws the exception CAppException when any of the following conditions are true:

- Command-line argument description cannot be found and argument descriptions have not been disabled (via call to protected method DisableArgDescription().
- Application diagnostic stream setup has failed.
- Registry data failed to load from a specified configuration file.
- An attempt is made to create a second instance of the CNcbiApplication (at any time, only one instance can be running).

- The specified configuration file cannot be opened.

As shown above, source files that utilize the CNcbiApplication class must #include the header file where that class is defined, corelib/ncbiapp.hpp, in the include/ directory. This header file in turn includes corelib/ncbistd.hpp, which should **always** be #include'd.

Processing Command-Line Arguments

This section discusses the classes that are used to process command-line arguments. A conceptual overview of these classes is covered in an introductory section. This section discusses these classes in detail and gives sample programs that use these classes.

This section discusses the following topics:

Capabilities of the Command-Line API

The set of classes for argument processing implement automated command line parsing. Specifically, these classes allow the developer to:

- Specify attributes of expected arguments, such as name, synopsis, comment, data type, etc.
- validate values of the arguments passed to the program against these specifications
- validate the number of positional arguments in the command line
- generate a USAGE message based on the argument descriptions

NOTE: -h flag to print the USAGE is defined by default.

- access the input argument values specifically typecast according to their descriptions

Normally, a CArgDescriptions object that contains the argument description is required and should be created in the application's Init() function before any other initialization. Otherwise, CNcbiApplication creates a default one, which allows any program that uses the NCBI C++ Toolkit to provide some standard command -line options, namely:

- to obtain a general description of the program as well as description of all available command-line parameters (-h flag)
- to redirect the program's diagnostic messages into a specified file (-logfile key)
- to read the program's configuration data from a specified file (-conffile key)

See Table 3 for the standard command-line options for the default instance of CArgDescriptions.

To avoid creation of a default CArgDescriptions object that may not be needed, for instance if the standard flags described in Table 3 are not used, one should call the CNcbiApplication::DisableArgDescriptions() function from an application object constructor.

It is also possible to use the CNcbiApplication::HideStdArgs(THideStdArgs hide_mask) method to hide description of the standard arguments (-h, -logfile, -conffile) in the USAGE message. Please note: This only hides the description of these flags; it is still possible to use them.

The Relationships between the CArgDescriptions, CArgs, and CArgValue Classes

The CArgDescriptions class provides an interface to describe the data type and attributes of command-line arguments via a set of AddXxx() methods. Additional constraints on the argument values can be imposed using the SetConstraint() method. The CreateArgs() method is passed the values of all command-line arguments at runtime. This method verifies their overall syntactic structure and matches their values against the stored descriptions. If the arguments are parsed successfully, a new CArgs object is returned by CreateArgs().

The resulting CArgs object will contain parsed, verified, and ready-to-use argument values, which are stored as CArgValue. The value of a particular argument can be accessed using the argument's name (as specified in the CArgDescriptions object), and the returned CArgValue object can then be safely type-cast to a correct C++ type (int, string, stream, etc.) because the argument types have been verified. These class relations and methods can be summarized schematically as shown in Figure 1.

The last statement in this example implicitly references a CArgValue object, in the value returned when the [ ] operator is applied to myArgs. The method CArgValue::AsDouble() is then applied to this object to retrieve a double.

Command-Line Syntax

Note: The C++ Toolkit supports two types of command line: "command-based" and "command-less". A "command-based" command line begins with a "command" (a case-sensitive keyword), typically followed by other arguments. A "command-less" command line doesn't contain such "commands".

This section deals primarily with command-less command lines, while the Supporting Command-Based Command Lines section covers command-based command lines.

Command-less command-line arguments fit the following profile:

```
progname {arg_key, arg_key_opt, arg_key_dflt, arg_flag} [--]
 {arg_pos} {arg_pos_opt, arg_pos_dflt}
 {arg_extra} {arg_extra_opt}
```

where:

arg_key	-<key> <value> -- (mandatory)
arg_key_opt	[-<key> <value>] -- (optional, without default value)
arg_key_dflt	[-<key> <value>] -- (optional, with default value)
arg_flag	-<flag> -- (always optional)
--	optional delimiter to indicate the beginning of pos. args
arg_pos	<value> -- (mandatory)
arg_pos_opt	[<value>] -- (optional, without default value)

arg_pos_dflt	[<value>] -- (optional, with default value)
arg_extra	<value> -- (dep. on the constraint policy)
arg_extra_opt	[<value>] -- (dep. on the constraint policy)

and: <key> must be followed by <value>. In all cases '-<key> <value>' is equivalent to '-<key>=<value>'. If '=' is used as separator, the value can be empty ('-<key>='). For arguments with a single-char name fOptionalSeparator flag can be set. In this case the value can be specified without any separator: -<k><value>

NOTE: No other argument's name can start with the same character to avoid conflicts. <flag> and <key> are case-sensitive, and they can contain only alphanumeric characters and dash ('-'). Only one leading dash is allowed. The leading dash can be used to create arguments which look like --<key> in the command line. <value> is an arbitrary string (additional constraints can be applied in the argument description, see "EType"). {arg_pos***} and {arg_extra***} are position-dependent arguments, with no tag preceding them. {arg_pos***} arguments have individual names and descriptions (see methods AddPositional***). {arg_extra***} arguments have one description for all (see method AddExtra). User can apply constraints on the number of mandatory and optional {arg_extra***} arguments.

Examples of command-less command lines:

```
MyProgram1 -reverse -depth 5 -name Lisa -log foo.log 1.c 2.c 3.c
MyProgram2 -i foo.txt -o foo.html -color red
MyProgram3 -a -quiet -pattern 'Error:' bar.txt
MyProgram4 -int-value=5 -str-value= -kValue
```

The Supporting Command-Based Command Lines section addresses how to support command-based command lines, such as:

```
svn diff myapp.cpp
svn checkin -m "message" myapp.cpp
```

The CArgDescriptions (*) class

CArgDescriptions contains a description of unparsed arguments, that is, user-specified descriptions that are then used to parse the arguments. CArgDescriptions is used as a container to store the command-line argument descriptions. The argument descriptions are used for parsing and verifying actual command-line arguments.

The following is a list of topics discussed in this section:

- The CArgDescriptions Constructor
- Describing Argument Attributes
- Argument Types
- Restricting the Input Argument Values
- Implementing User-defined Restrictions Using the CArgAllow Class
- Using CArgDescriptions in Applications
- Generating a USAGE Message

The CArgDescriptions Constructor

The constructor for CArgDescriptions accepts a Boolean argument, auto_help, set to TRUE by default.

CArgDescriptions(bool auto_help = true);

If "auto_help" is passed TRUE, then a special flag "-h" will be added to the list of accepted arguments, and passing "-h" in the command line will print out USAGE and ignore all other passed arguments.

Describing Argument Attributes

CNcbiArguments contains many methods, called AddXxx(). The "Xxx" refers to the types of arguments, such as mandatory key (named) arguments, optional key arguments, positional arguments, flag arguments, etc. For example, the AddKey() method refers to adding a description for a mandatory key argument.

The methods for AddXxx() are passed the following argument attributes:

- *name*, the string that will be used to identify the variable, as in: CArgs[name]. For all tagged variables in a command line, *name* is also the key (or flag) to be used there, as in: "-name value" (or "-name").
- *synopsis*, for key_*** arguments only. The automatically generated USAGE message includes an argument description in the format: *-name [synopsis] <type, constraint>* comment.
- *comment*, to be displayed in the USAGE message, as described above.
- *value type*, one of the scalar values defined in the EType enumeration, which defines the type of the argument.
- *default,* for key_dflt and pos_dflt arguments only. A default value to be used if the argument is not included in the command line (only available for optional program arguments).
- *flags*, the flags argument, to provide additional control of the arguments' behavior.

Argument Types

The CArgDescriptions class enables registration of command-line arguments that fit one of the following pattern types:

Mandatory named arguments:-<key> <value> (example: -age 31) Position-independent arguments that **must** be present in the command line. AddKey (key, synopsis, comment, value_type, flags)

Optional named arguments:[-<key> <value>] (example: -name Lisa) Position-independent arguments that are **optional**. AddOptionalKey (key, synopsis, comment, value_type, flags) A default value can be specified in the argument's description to cover those cases where the argument does not occur in the command line. AddDefaultKey (key, synopsis, comment, value_type, default_value, flags)

Optional named flags:[-<flag>] (example: -reverse) Position-independent boolean (without value) arguments. These arguments are **always** optional. AddFlag (flag, comment, set_value)

Mandatory named positional arguments:<value> (example: 12 Feb) These are position-dependent arguments (of any type), which are read using a value only. They do, however, have names stored with their descriptions, which they are associated with in an order-dependent fashion. Specifically, the order in which untagged argument descriptions are added to the CArgDescriptions object using AddPositional() defines the order in which these arguments should appear in the command line. AddPositional (key, comment, value_type, flags)

Optional named positional arguments:[value] (example: foo.txt bar) Position-dependent arguments that are optional. They always go after the mandatory positional arguments. The order in which untagged argument descriptions are added to the CArgDescriptions object using Add[Optional|Default]Positional() defines the order in which these arguments should appear in the command line. AddOptionalPositional (key, comment, value_type, flags) AddDefaultPositional (key, comment, value_type, default_value, flags)

Unnamed positional arguments (all of the same type: <value1> | [valueN] (example: foo.c bar.c xxx.c). These are also position-dependent arguments that are read using a value only. They are expected to appear at the very end of the command line, after all named arguments. Unlike the previous argument type, however, these arguments do not have individual, named descriptions but share a single "unnamed" description. You can specify how many mandatory and how many optional arguments to expect using n_mandatory and n_optional parameters: AddExtra (n_mandatory, n_optional, comment, type, flags)

Aliases can be created for any arguments. They allow using an alternative argument name in the command line. However, only the original argument name can be used to access its value in the C++ code.

Any of the registered descriptions can be tested for existence and/or deleted using the following CArgDescriptions methods:

```
bool Exist(const string& name) const;
void Delete(const string& name);
```

These methods can also be applied to the unnamed positional arguments (as a group), using: Exist(kEmptyStr) and Delete(kEmptyStr).

Restricting the Input Argument Values

Although each argument's input value is initially loaded as a simple character string, the argument's specified type implies a restricted set of possible values. For example, if the type is eInteger, then any integer value is acceptable, but floating point and non-numerical values are not. The EType enumeration quantifies the allowed types and is defined as:

```
/// Available argument types.
enum EType {
 eString = 0, ///< An arbitrary string
 eBoolean, ///< {'true', 't', 'false', 'f'}, case-insensitive
 eInteger, ///< Convertible into an integer number (int)
 eDouble, ///< Convertible into a floating point number (double)
 eInputFile, ///< Name of file (must exist and be readable)
 eOutputFile, ///< Name of file (must be writeable)
 k_EType_Size ///< For internal use only
};
```

Implementing User-defined Restrictions Using the CArgAllow Class

It may be necessary to specify a restricted range for argument values. For example, an integer argument that has a range between 5 and 10. Further restrictions on the allowed values can be specified using the CArgDescriptions::SetConstraint() method with the CArgAllow class. For example:

```
auto_ptr<CArgDescriptions> args(new CArgDescriptions);
// add descriptions for "firstint" and "nextint" using AddXxx( ...)
...
CArgAllow* constraint = new CArgAllow_Integers(5,10);
args->SetConstraint("firstInt", constraint);
args->SetConstraint("nextInt", constraint);
```

This specifies that the arguments named "firstInt" and "nextInt" must both be in the range [5, 10].

The CArgAllow_Integers class is derived from the **abstract**CArgAllow class. The constructor takes the two integer arguments as lower and upper bounds for allowed values. Similarly, the CArgAllow_Doubles class can be used to specify a range of allowed floating point values. For both classes, the order of the numeric arguments does not matter, because the constructors will use min/max comparisons to generate a valid range.

A third class derived from the CArgAllow class is the CArgAllow_Strings class. In this case, the set of allowed values cannot be specified by a range, but the following construct can be used to enumerate all eligible string values:

```
CArgAllow* constraint = (new CArgAllow_Strings())->
 Allow("this)->Allow("that")->Allow("etc");
args.SetConstraint("someString", constraint);
```

Here, the constructor takes no arguments, and the Allow() method returns this. Thus, a list of allowed strings can be specified by daisy-chaining a set of calls to Allow(). A bit unusual yet terser notation can also be used by engaging the comma operator, as in:

```
args.SetConstraint("someString",
 &(*new CArgAllow_Strings, "this", "that", "etc"));
```

There are two other pre-defined constraint classes: CArgAllow_Symbols and CArgAllow_String. If the value provided on the command line is not in the allowed set of values specified for that argument, then an exception will be generated. This exception can be caught and handled in the usual manner, as described in the discussion of Generating a USAGE message.

Using CArgDescriptions in Applications

The description of program arguments should be provided in the application's Init() function before any other initialization. A good idea is also to specify the description of the program here:

```
auto_ptr<CArgDescriptions> arg_desc(new CArgDescriptions);
arg_desc->SetUsageContext(GetArguments().GetProgramBasename(),
 "program's description here");
// Define arguments, if any
...
SetupArgDescriptions(arg_desc.release());
```

The SetUsageContext() method is used to define the name of the program and its description, which is to be displayed in the USAGE message. As long as the initialization of the application is completed and there is still no argument description, CNcbiApplication class provides a "default" one. This behavior can be overridden by calling the DisableArgDescriptions() method of CNcbiAppliation.

Generating a USAGE Message

One of the functions of the CArgDescriptions object is to generate a USAGE message automatically (this gives yet another reason to define one). Once such object is defined, there is nothing else to worry about; CNcbiApplication will do the job for you. The SetupArgDescriptions() method includes parsing the command line and matching arguments against their descriptions. Should an error occur, e.g., a mandatory argument is missing, the program prints a message explaining what was wrong and terminates. The output in this case might look like this:

```
USAGE
 myApp -h -k MandatoryKey [optarg]
DESCRIPTION
 myApp test program
REQUIRED ARGUMENTS
 -k <String>
 This is a mandatory alpha-num key argument
OPTIONAL ARGUMENTS
 -h
 Print this USAGE message; ignore other arguments
 optarg <File_Out>
 This is an optional named positional argument without default
 value
```

The message shows a description of the program and a summary of each argument. In this example, the description of the input file argument was defined as:

```
arg_desc->AddKey( "k", "MandatoryKey",
 "This is a mandatory alpha-num key argument",
 CArgDescriptions::eString);
```

The information generated for each argument is displayed in the format:

me [synopsis] <type [, constraint] > comment [default = ...]

The arguments in the USAGE message can be arranged into groups by using SetCurrentGroup () method of the CArgDescriptions object.

The CArgs (*) Class: A Container Class for CArgValue (*) Objects

The CArgs class provides a data structure where the values of the parsed arguments can be stored and includes access routines in its public interface. Argument values are obtained from the unprocessed command-line arguments via the CNcbiArguments class and then verified and processed according to the argument descriptions defined by the user in CArgDescriptions. The following describes the public interface methods in CArgs:

```
class CArgs
{
public:
 /// Constructor.
 CArgs(void);
 /// Destructor.
 ~CArgs(void);
 /// Check existence of argument description.
 ///
 /// Return TRUE if arg 'name' was described in the parent CArgDescriptions.
 bool Exist(const string& name) const;
 /// Get value of argument by name.
 ///
 /// Throw an exception if such argument does not exist.
 /// @sa
 /// Exist() above.
 const CArgValue& operator[] (const string& name) const;
 /// Get the number of unnamed positional (a.k.a. extra) args.
 size_t GetNExtra(void) const { return m_nExtra; }
 /// Return N-th extra arg value, N = 1 to GetNExtra().
 const CArgValue& operator[] (size_t idx) const;
 /// Print (append) all arguments to the string 'str' and return 'str'.
 string& Print(string& str) const;
 /// Add new argument name and value.
 ///
 /// Throw an exception if the 'name' is not an empty string, and if
 /// there is an argument with this name already.
 ///
 /// HINT: Use empty 'name' to add extra (unnamed) args, and they will be
 /// automatically assigned with the virtual names: '#1', '#2', '#3', etc.
 void Add(CArgValue* arg);
 /// Check if there are no arguments in this container.
 bool IsEmpty(void) const;
};
```

The CArgs object is created by executing the CArgDescriptions::CreateArgs() method. What happens when the CArgDescriptions::CreateArgs() method is executed is that the arguments of the command line are validated against the registered descriptions, and a CArgs object is created. Each argument value is internally represented as a CArgValue object and is added to a container managed by the CArgs object.

All named arguments can be accessed using the [ ] operator, as in: myCArgs["f"], where "f" is the name registered for that argument. There are two ways to access the **N**-th unnamed positional argument: myCArgs["#N"] and myCArgs[N], where 1 <= **N** <= GetNExtra().

CArgValue (*) Class: The Internal Representation of Argument Values

The internal representation of an argument value, as it is stored and retrieved from its CArgs container, is an instance of a CArgValue. The primary purpose of this class is to provide type-validated loading through a set of AsXxx() methods where "Xxx" is the argument type such as "Integer", "Boolean", "Double", etc. The following describes the public interface methods in CArgValue:

```
class CArgValue : public CObject
{
public:
 /// Get argument name.
 const string& GetName(void) const { return m_Name; }
 /// Check if argument holds a value.
 ///
 /// Argument does not hold value if it was described as optional argument
 /// without default value, and if it was not passed a value in the command
 /// line. On attempt to retrieve the value from such "no-value" argument,
 /// exception will be thrown.
 virtual bool HasValue(void) const = 0;
 operator bool (void) const { return HasValue(); }
 bool operator!(void) const { return !HasValue(); }
 /// Get the argument's string value.
 ///
 /// If it is a value of a flag argument, then return either "true"
 /// or "false".
 /// @sa
 /// AsInteger(), AsDouble(), AsBoolean()
 virtual const string& AsString(void) const = 0;
 /// Get the argument's integer value.
 ///
 /// If you request a wrong value type, such as a call to "AsInteger()"
 /// for a "boolean" argument, an exception is thrown.
 /// @sa
 /// AsString(), AsDouble, AsBoolean()
 virtual int AsInteger(void) const = 0;
 /// Get the argument's double value.
 ///
 /// If you request a wrong value type, such as a call to "AsDouble()"
 /// for a "boolean" argument, an exception is thrown.
 /// @sa
 /// AsString(), AsInteger, AsBoolean()
 virtual double AsDouble (void) const = 0;
 /// Get the argument's boolean value.
 ///
 /// If you request a wrong value type, such as a call to "AsBoolean()"
 /// for a "integer" argument, an exception is thrown.
 /// @sa
 /// AsString(), AsInteger, AsDouble()
 virtual bool AsBoolean(void) const = 0;
 /// Get the argument as an input file stream.
 virtual CNcbiIstream& AsInputFile (void) const = 0;
 /// Get the argument as an output file stream.
 virtual CNcbiOstream& AsOutputFile(void) const = 0;
 /// Close the file.
 virtual void CloseFile (void) const = 0;
};
```

Each of these AsXxx() methods will access the string storing the value of the requested argument and attempt to convert that string to the specified type, using for example, functions such as atoi() or atof(). Thus, the following construct can be used to obtain the value of a floating point argument named "f":

```
float f = args["f"].AsDouble();
```

An exception will be generated with an appropriate error message, if:

- the conversion fails, or
- "f" was described as an optional key or positional argument without default value (i.e., using the AddOptional***() method), and it was not defined in the command line. Note that you can check for this case using the CArgValue::HasValue() method.

Supporting Command-Based Command Lines

For some applications, multiple command-based command line forms are needed, with different arguments depending on the command. For example:

```
myapp list
myapp create <queue>
myapp post <queue> [-imp importance] <message>
myapp query [queue]
```

Commands are case-sensitive keywords and are typically followed by other arguments. Programs that support command-based command lines can support any number of commands (each with its own set of supported arguments), and may optionally support a command-less command line in addition.

Command-based command lines have a requirement that command-less command lines don't - the ability to have optional arguments between mandatory arguments. Opening arguments address this requirement. Opening arguments are essentially identical to mandatory positional arguments except that opening arguments must precede optional arguments whereas mandatory positional arguments must follow them. Thus, opening arguments allow usage forms such as the "post" command in the above example, which has an optional argument between mandatory arguments.

At a high level, setting up a program to support a command-less command-line requires creating a CArgDescriptions object, adding argument descriptions to it, and passing it to SetupArgDescriptions().

Setting up a program to support command-based command lines is similar, but requires a CCommandArgDescriptions object instead. The CCommandArgDescriptions class is derived from CArgDescriptions, so all the same functionality is available; however, the AddCommand () method of CCommandArgDescriptions allows you to create multiple CArgDescriptions objects (one for each command) in addition to the overall program description. Other command-specific features are also provided, such as command grouping. Note: The ECommandPresence parameter of the CCommandArgDescriptions constructor controls whether or not the user must enter a command-based command line. Use eCommandOptional only when you are setting up both command-less and command-based command lines.

Programs that support command-based command lines must execute these steps:

1. Create a command descriptions object (class CCommandArgDescriptions) for the overall program description.
2. Create argument descriptions objects (class CArgDescriptions) for each command.
3. Add the actual argument descriptions to the argument descriptions objects using methods such as AddOpening(), AddPositional(), etc.
4. Add each argument descriptions object to the overall command descriptions object.

5. Determine which command was specified on the command line.
6. Process the appropriate arguments for the given command.

For a sample program that demonstrates argument processing for command-based command lines, see multi_command.cpp.

For more information on standard command lines and general information applicable to all command line processing, see the Command-Line Syntax and CArgDescriptions sections.

Code Examples

A simple application program, test_ncbiargs_sample.cpp demonstrates the usage of these classes for argument processing. See also test_ncbiargs.cpp (especially main(), s_InitTest0() and s_RunTest0() there), and asn2asn.cpp for more examples.

Namespace, Name Concatenation, and Compiler-specific Macros

The file ncbistl.hpp provides a number of macros on namespace usage, name concatenation, and macros for handling compiler-specific behavior.

These topics are discussed in greater detail in the following subsections:

- NCBI Namespace
- Other Name Space Macros
- Name Concatenation
- Compiler Specific Macros

NCBI Namespace

All new NCBI classes must be in the ncbi:: namespace to avoid naming conflicts with other libraries or code. Rather than enclose all newly defined code in the following, it is, from a stylistic point of view, better to use specially defined macros such as BEGIN_NCBI_SCOPE, END_NCBI_SCOPE, USING_NCBI_SCOPE:

```
namespace ncbi {
 // Indented code etc.
}
```

The use of BEGIN_NCBI_SCOPE, END_NCBI_SCOPE, and USING_NCBI_SCOPE is discussed in use of the NCBI name scope.

Other Namespace Macros

The BEGIN_NCBI_SCOPE, END_NCBI_SCOPE, and USING_NCBI_SCOPE macros in turn use the more general purpose BEGIN_SCOPE(ns), END_SCOPE(ns), and USING_SCOPE(ns) macros, where the macro parameter ns is the namespace being defined. All NCBI-related code should be in the ncbi:: namespace so the BEGIN_NCBI_SCOPE, END_NCBI_SCOPE, and USING_NCBI_SCOPE should be adequate for new NCBI code. However, in those rare circumstances, if you need to define a new name scope, you can directly use the BEGIN_SCOPE(ns), END_SCOPE(ns), and USING_SCOPE(ns) macros.

Name Concatenation

The macros NCBI_NAME2 and NCBI_NAME3 define concatenation of two and three names, respectively. These are used to build names for program-generated class, struct, or method names.

Compiler-specific Macros

To cater to the idiosyncrasies of compilers that have non-standard behavior, certain macros are defined to normalize their behavior.

The BREAK(it) macro advances the iterator to the end of the loop and then breaks out of the loop for the Sun WorkShop compiler with versions less than 5.3. This is done because this compiler fails to call destructors for objects created in for-loop initializers. This macro prevents trouble with iterators that contain CRefs by advancing them to the end using a while-loop, thus avoiding the "deletion of referenced CObject" errors. For other compilers, BREAK(it) is defined as the keyword break.

The ICC compiler may fail to generate code preceded by template<>. In this case, use the macro EMPTY_TEMPLATE instead, which expands to an empty string for the ICC compiler and to template<> for all other compilers.

For MSVC v6.0, the for keyword is defined as a macro to overcome a problem with for-loops in the compiler. The local variables in a for-loop initalization are visible outside the loop:

```
for (int i; i < 10; ++i) {
// scope of i
}
// i should not be visible, but is visible in MSVC 6.0
```

Another macro called NCBI_EAT_SEMICOLON is used in creating new names that can allow a trailing semicolon without producing a compiler warning in some compilers.

Configuration Parameters

The CParam class is the preferred method for defining configuration parameters. This class enables storing parameters with per-object values, thread-wide defaults, and application-wide defaults. Global default values may be set through the application registry or the environment.

The following topics discuss using the CParam class.

- General Usage Information
- Macros for Creating Parameters
- Methods for Using Parameters
- Supporting Classes

General Usage Information

A CParam instance gets its initial value from one of three sources. If the application registry specifies a value, then that value will be used. Otherwise if the environment specifies a value, then that value will be used. Otherwise the default value supplied in the definition will be used. Later, the value can be changed using various methods.

N.B. statically defined instances of configuration parameters will be assigned their default values even if the environment and / or application registry specify (possibly different) values for them. This is because they are constructed (using their default value) at program startup and at that time the application framework for reading from the environment and application registry hasn't been set up yet. Therefore it is important to call the Reset() method for these parameters prior to reading their value. Alternatively, the GetState() method will indicate whether or not all possible sources were checked when a value was assigned to a configuration parameter - if they were, it will have either the value eState_Config or eState_User.

For more information on the application framework, the environment, and the application registry, see the sections on CNcbiApplication, CNcbiEnvironment, and CNcbiRegistry.

Be sure to include the header file in your source files:

```
#include <corelib/ncbi_param.hpp>
```

and include the NCBI core library in your makefile:

```
LIB = xncbi
```

Macros for Creating Parameters

The CParam class is not designed to be used directly for creating configuration parameter variables. Instead, it supplies macros which your code should use. These macros have parameters for types, sections, names, default values, flags, and environment.

The type macro parameter must:

- be a POD type;
- be initializable by the pre-processor from a literal;
- be readable from and writable to streams.

Typically, the type is a simple type such as string, bool, int, or enum, as these are most convenient for specifying parameter values.

The section macro parameter indicates which section of a configuration file the parameter should be located in.

The name macro parameter uniquely identifies the parameter within the section.

The default_value macro parameter provides the default value for the parameter - i.e. the value the parameter has from the time it is created until it is overwritten by a value from the environment, configuration file, or user code - and the value it is assigned by the Reset() method.

The flags macro parameter (a bitwise OR of enum values) can be used to control certain behavior options for the parameter. Currently, these enum values are:

Enum Value	**Purpose**
eParam_Default	Default flags
eParam_NoLoad	Do not load from registry or environment
eParam_NoThread	Do not use per-thread values

See the enum definition for an up-to-date list.

The env macro parameter can be used to specify the environment variable to be searched. If the env macro parameter is not used, the environment will be searched for a variable having the form NCBI_CONFIG__<section>__<name> (note: the first underscore is single; the others are double).

CParam instances must be declared and defined before use. A typedef may also be created.

To *declare* simple parameters, use the NCBI_PARAM_DECL macro:

```
NCBI_PARAM_DECL(type, section, name);
```

For example, declaring a host name parameter for a server might look like:

```
NCBI_PARAM_DECL(string, XyzSrv, Host);
```

To declare an enum:

```
NCBI_PARAM_ENUM_DECL(type, section, name);
```

Additional macros for parameter declarations include:

- NCBI_PARAM_DECL_EXPORT and NCBI_PARAM_ENUM_DECL_EXPORT to include the EXPORT specifier (i.e. NCBI_XNCBI_EXPORT). Note: this form must be used if the parameter is defined in a header file and compiled into a library. Otherwise the linker may create several instances of the parameter which could contain different values.

To *define* simple parameters, use the NCBI_PARAM_DEF or NCBI_PARAM_DEF_EX macro:

```
NCBI_PARAM_DEF(type, section, name, default_value); // OR
NCBI_PARAM_DEF_EX(type, section, name, default_value, flags, env);
```

For example, an extended definition of a host name parameter for a server could look like:

```
NCBI_PARAM_DEF_EX(string, Xyz, Host, "xyz.nih.gov", eParam_NoThread,
XYZ_HOST);
```

To define an enum:

```
NCBI_PARAM_ENUM_ARRAY(type, section, name); // USE THIS AND EITHER:
NCBI_PARAM_ENUM_DEF(type, section, name, default_value); // OR:
NCBI_PARAM_ENUM_DEF_EX(type, section, name, default_value, flags, env);
```

For example, an enum definition could look like:

```
NCBI_PARAM_ENUM_ARRAY(EMyEnum, MySection, MyEnumParam)
{
 {"My_A", eMyEnum_A},
 {"My_B", eMyEnum_B},
 {"My_C", eMyEnum_C},
};
NCBI_PARAM_ENUM_DEF(EMyEnum, MySection, MyEnumParam, eMyEnum_B);
```

An additional macro for parameter definitions is:

- NCBI_PARAM_DEF_IN_SCOPE to define the parameter within a scope.

Another way to conveniently use a configuration parameter is to use the NCBI_PARAM_TYPE macro to create an instance of a type. The following example illustrates the declaration, definition, typedef, and use of a configuration parameter:

```
NCBI_PARAM_DECL(bool, NCBI, ABORT_ON_COBJECT_THROW);
NCBI_PARAM_DEF_EX(bool, NCBI, ABORT_ON_COBJECT_THROW, false,
 eParam_NoThread, NCBI_ABORT_ON_COBJECT_THROW);
typedef NCBI_PARAM_TYPE(NCBI, ABORT_ON_COBJECT_THROW) TAbortOnCObectThrow;

void CObjectException::x_InitErrCode(CException::EErrCode err_code)
{
 CCoreException::x_InitErrCode(err_code);
 static TAbortOnCObectThrow sx_abort_on_throw;
 if ( sx_abort_on_throw.Get() ) {
 Abort();
 }
}
```

Methods for Using Parameters

Important methods of the CParam class are:

Method	Static	Purpose
GetState()	Yes	Get the current state of the parameter. The state indicates the last source checked when assigning its value. N.B. it specifically does *not* indicate the origin of the current value. See the EParamState enum for specific values.
Get()	No	Get the current parameter value.
Set()	No	Set a new parameter value (this instance only).
Reset()	No	Reset the value as if it has not been initialized yet.
GetDefault()	Yes	Get the global default value.
SetDefault()	Yes	Set a new global default value.
ResetDefault()	Yes	Reload the global default value from the environment/registry or reset it to the initial value specified in NCBI_PARAM_DEF.
GetThreadDefault()	Yes	Get the thread-local default value if set, otherwise the global default value.
SetThreadDefault()	Yes	Set a new thread-local default value.
ResetThreadDefault()	Yes	Reset the thread default value as if it has not been set.

Typical uses involve getting the current or default values:

```
// get a parameter's default value
string bots = NCBI_PARAM_TYPE(CGI,Bots)::GetDefault();

// get a parameter's current value
typedef NCBI_PARAM_TYPE(READ_FASTA, USE_NEW_IMPLEMENTATION) TParam_NewImpl;
TParam_NewImpl new_impl;
if (new_impl.Get()) {
 // do something
}
```

Supporting Classes

The CParam class is packaged with two supporting classes: CParamException and CParamParser.

CParamException will be thrown by the parameter parser if invalid parameter values are specified in the environment, configuration file, or code.

CParamParser is a templatized helper class that parses parameter literals into parameter values, using its StringToValue() method. [Note: the "String" in this method name refers to the string of characters in the literal being parsed (regardless of the type it represents), not to the std::string type.] A ValueToString() method is also provided for completeness.

CParamParser templates have been pre-defined for string, bool, int, and enum types. If you need to create a configuration parameter that is more complex than these types, then you will need to either instantiate CParamParser for your type or define appropriate operator<<() and operator>>() methods. This will:

- enable parsing of the default value specified in the definition of your complex configuration parameter;
- enable that type to be read from the application registry or environment; and
- enable that type to be assigned values via the Set*() methods.

Note: Defining the appropriate operator<<() and operator>>() methods is preferrable to instantiating CParamParser for your type because:

- instantiating CParamParser for your type would make it more difficult to change the CParamParser template, if that should become necessary; and
- operator<<() and operator>>() can be useful in other contexts.

Using the CNcbiRegistry Class

If for some reason the CParam class cannot be used to define configuration parameters, the CNcbiRegistry class may be used instead.

This section provides reference information on the use of the CNcbiRegistry class. For an overview of this class, refer to the introductory chapter. This class is also discussed in the library configuration chapter.

The following topics are discussed in this section:

- Working with the Registry class: CNcbiRegistry
- Syntax of the Registry Configuration File
- Search Order for Initialization (*.ini) Files
- Fine-Tuning Registry Parameters Using IRegistry::EFlags
- Main Methods of CNcbiRegistry
- Additional Registry Methods

Working with the Registry Class: CNcbiRegistry

The CNcbiRegistry class is used to load, access, modify, and store runtime information read from configuration files. Previously, these files were by convention named .*rc files on Unix-like systems. The convention for all platforms now is to name such files *.ini (where * is by default the application name). An exception to this rule is the system-wide registry, which is named .ncbirc on Unix-like systems and ncbi.ini on Windows systems. The CNcbiRegistry class can read and parse configuration files, search and edit retrieved information, and write back to the file.

The following resources are checked when loading a registry:

- the environment
- the overrides registry
- the application registry
- the system registry
- inherited registries

In addition, registries can be loaded from files programmatically.

An environment registry is created from configuration parameters specified in the environment. Often, such variables have the form NCBI_CONFIG__<section>__<entry> (note the double underscores) and can have corresponding entries in initialization files, but see the library configuration chapter for details on specific parameters. Entries in the environment registry have the highest precedence.

If the special environment variable NCBI_CONFIG_OVERRIDES is defined, the configuration file it names will be loaded as the overrides registry. This registry will have the next highest precedence after the environment.

For the application registry, the name of the configuration file can be explicitly set with the -conffile command-line argument, set (or disabled) with the conf argument of CNcbiApplication::AppMain(), or implicitly set (or disabled) according to search order rules. If the -conffile command-line argument is supplied, that path will be used. If the conf argument to AppMain() is supplied, the file will be determined according to Table 2. Otherwise, the file will be determined according to search order rules. The application registry follows the overrides registry in precedence.

When the application registry is successfully loaded, you can access it using the method CNcbiApplication::GetConfig(). The application will throw an exception if the config file is found, is not empty, and either cannot be opened or contains invalid entries. If the conf argument to CNcbiApplication::AppMain() is not NULL and the config file cannot be found, then a warning will be posted to the application diagnostic stream.

System-wide configuration parameters can be defined in the system registry. The system registry will not be loaded if it contains the DONT_USE_NCBIRC entry in the NCBI section or if the environment variable NCBI_DONT_USE_NCBIRC is defined. See the search order section below for details. The system registry follows the application registry in precedence.

Configuration files may "inherit" entries from other configuration files using the .Inherits entry in the [NCBI] section. The .Inherits entry is a space- and/or comma- delimited list of file names. Files having a .ini extension may be listed in the .Inherits entry without the .ini extension. Note that extensionless file names are not supported in the .Inherits entry. Inherited registries have the same precedence as the registry that inherited them.

Registries can be programmatically loaded from files by calling CNcbiRegistry::Read(). CNcbiApplication::LoadConfig() can also be called to "manually" load the application registry - for example, if special flags are required. The precedence for programmatically loaded registries depends on the flags they are loaded with. By default (or if loaded with the IRegistry::fOverride flag) they will have greater precedence that previously loaded registries, but if loaded with the IRegistry::fNoOverride flag, they will not override existing parameters.

Although registry objects can be instantiated and manipulated independently, they are typically used by the CNcbiApplication class. Specifically, CNcbiApplication::AppMain() attempts to load a registry with entries from all of the above sources (except programmatically loaded registries). AppMain() will look for the system and application registries in multiple locations, and possibly with a modified name, as described in the search order section below.

See the Registry and Environment sections of the library configuration chapter for more information on controlling the registry via the environment.

Syntax of the Registry Configuration File

The configuration file is composed of section headers and "name=value" strings, which occur within the named sections. It is also possible to include comments in the file, which are indicated by a new line with a leading semicolon. An example configuration file is shown below.

```
# Registry file comment (begin of file)
# MyProgram.ini
; parameters for section1
[section1]
name1 = value1 and value1.2
n-2.3 = " this value has two spaces at its very beginning and at the end "
name3 = this is a multi\
line value
name4 = this is a single line ended by back slash\\
name5 = all backslashes and \
new lines must be \\escaped\\...
[ section2.9-bis ]
; This is a comment...
name2 = value2
```

All comments and empty lines are ignored by the registry file parser. Line continuations, as usual, are indicated with a backslash escape. More generally, backslashes are processed as:

- [backslash] + [backslash] -- converted into a single [backslash]
- [backslash] + [space(s)] + [EndOfLine] -- converted to an [EndOfLine]
- [backslash] + ["] -- converted into a ["]

Character strings with embedded spaces do not need to be quoted, and an unescaped double quote at the very beginning or end of a value is ignored. All other combinations with [backslash] and ["] are invalid.

The following restrictions apply to the section and name identifiers occurring in a registry file:

- the string must contain only: [a-z], [A-Z], [0-9], [_.-/] characters
- the interpretation of the string is **not** case sensitive, e.g., PATH == path == PaTh
- all leading and trailing spaces will be truncated

A special syntax is provided for "including" the content of one section into another section:

```
.Include = section_name
```

For example, this:

```
[section-a]
;section-a specific entries...
```

```
a1 = a one
.Include = common

[section-b]
;section-b specific entries...
b1 = b one
.Include = common

[common]
;common entries
c1 = c one
c2 = c two
```

is equivalent to:

```
[section-a]
;section-a specific entries...
a1 = a one
;common entries
c1 = c one
c2 = c two

[section-b]
;section-b specific entries...
b1 = b one
;common entries
c1 = c one
c2 = c two
```

Another special syntax is provided for "including" other configuration files:

```
[NCBI]
.Inherits = subregistry_list
```

Here, subregistry_list is a comma- or space- separated list of one or more subregistry files. Subregistry file names are not required to have a ".ini" extension. However if they do, the ".ini" can be omitted from the subregistry list. For example, the specification:

```
[NCBI]
.Inherits = a
```

will select "a.ini". Subregistries can also define their own subregistries, thus permitting an application to read a tree of configuration files.

Given a specification of:

```
[NCBI]
.Inherits = a b
```

an entry in "a.ini" or any of its subregistries will take priority over an identically named entry in "b.ini" or any of its subregistries. This could be used, for example, to retain a default configuration while working with a test configuration, such as in:

```
[NCBI]
.Inherits = mytest.ini myapp.ini
```

Entries in the main configuration file take priority over entries in subregistries.

Entries defined in a subregistry can be "undefined" by explicitly defining the entry as empty in a higher priority registry file.

Finally, the environment variable NCBI_CONFIG_OVERRIDES can be used to name a configuration file whose entries override any corresponding entries in all the processed registry files.

Search Order for Initialization (*.ini) Files

Note: This section discusses the search order for initialization files, which is only applicable to the application and system initialization files. Please see the Working with the Registry Class section for a discussion about the other sources of configuration information and the relative precedence of all registry sources.

Note: See Table 2 for rules about how the conf argument to AppMain() affects the search rules for the application initialization file. Also, if the -conffile command-line argument is used, then only that application initialization file is tried.

Note: Several means are available to control loading of the system initialization file. It can be enabled by the IRegistry::fWithNcbirc flag. It can be disabled if (1) it contains the DONT_USE_NCBIRC entry in the NCBI section, (2) it contains syntax errors or no entries, or (3) if the environment variable NCBI_DONT_USE_NCBIRC is defined.

With the exceptions noted above, the following rules determine the search order for application and system initialization files. Although application and system initialization files are not typically found in the same place, the same search order rules apply to both (with the above exceptions).

1. If the environment variable NCBI_CONFIG_PATH is set, that will be the only path searched for initialization files.
2. Otherwise, the search order includes the following directories in order:
    - **a** If the environment variable NCBI_DONT_USE_LOCAL_CONFIG is *not* defined then:
        - **i** The current working directory (".").
        - **ii** The user's home directory (if it can be established).
    - **b** The path in the environment variable NCBI (if it is defined).
    - **c** The standard system directory ("/etc" on Unix-like systems, and given by the environment variable SYSTEMROOT on Windows).
    - **d** The directory containing the application, if known (this requires use of CNcbiApplication).

Note: The search ends with the first file found.

The above rules determine the search order for directories, but there are also rules for initialization file names:

For the application registry: When the initialization file name is not explicitly specified (e.g. on the command line) then the implicit name will be formed by appending ".ini" to the application name. When the application name contains extensions, multiple names may be tried by sequentially stripping extensions off the application name. For example, if an application name is a.b.c then the sequence of initialization file names tried is: a.b.c.ini, a.b.ini, and finally a.ini.

On Unix-like systems, if an application dir1/app1 is a symlink to dir2/app2, the directory/name search order will be:

1. ./app1.ini
2. $NCBI/app1.ini
3. ~/app1.ini
4. dir1/app1.ini
5. dir2/app1.ini
6. ./app2.ini
7. $NCBI/app2.ini
8. ~/app2.ini
9. dir1/app2.ini
10. dir2/app2.ini

For the system registry: The name .ncbirc is tried on Unix-like systems and ncbi.ini is tried on Windows. Note: NCBI in-house Linux systems have "/etc/.ncbirc" symlinked to "/opt/ncbi/config/.ncbirc" so that applications running on production systems (or with NCBI unset) still pick up standard configuration settings.

Fine-Tuning Registry Parameters Using IRegistry::EFlags

Note: This section deals with concepts not typically needed by most C++ Toolkit users. The functionality of CNcbiRegistry is automatically and transparently provided when you use CNcbiApplication. You probably won't need to read this section unless you're working with an application that edits registry files or explicitly sets registry entry values.

Each CNcbiRegistry entry has a set of flags that control how it is handled, defined by this enum:

```
enum EFlags {
 fTransient = 0x1, ///< Transient -- not saved by default
 fPersistent = 0x100, ///< Persistent -- saved when file is written
 fOverride = 0x2, ///< Existing value can be overriden
 fNoOverride = 0x200, ///< Cannot change existing value
 fTruncate = 0x4, ///< Leading, trailing blanks can be truncated
 fNoTruncate = 0x400, ///< Cannot truncate parameter value
 fJustCore = 0x8, ///< Ignore auxiliary subregistries
 fNotJustCore = 0x800, ///< Include auxiliary subregistries
 fIgnoreErrors = 0x10, ///< Continue reading after parse errors
 fInternalSpaces = 0x20, ///< Allow internal whitespace in names
 fWithNcbirc = 0x40, ///< Include .ncbirc (used only by CNcbiRegistry)
 fCountCleared = 0x80, ///< Let explicitly cleared entries stand
 fSectionCase = 0x1000,///< Create with case-sensitive section names
 fEntryCase = 0x2000,///< Create with case-sensitive entry names
 fCoreLayers = fTransient | fPersistent | fJustCore,
 fAllLayers = fTransient | fPersistent | fNotJustCore,
 fCaseFlags = fSectionCase | fEntryCase
};
typedef int TFlags; ///< Binary OR of "EFlags"
```

Some pairs of these flags are mutually exclusive and have a default if neither flag is given:

Flag Pair	Default
fTransient / fPersistent	fPersistent
fOverride / fNoOverride	fOverride
fJustCore / fNotJustCore	fJustCore

It is not necessary to use the fNoTruncate flag because it represents the default behavior - no values are truncated unless fTruncate is used.

The flag fWithNcbirc can be passed to the CNcbiRegistry constructor, the CNcbiRegistry::IncludeNcbircIfAllowed() method, or the IRWRegistry::IncludeNcbircIfAllowed() method. If it is set then the system-wide registry is used - see the search order section for details on the system-wide registry.

For example, the following code demonstrates that the bit-wise OR of fTruncate and fNoOverride strips all leading and trailing blanks and does not override an existing value:

```
CNcbiRegistry reg;
CNcbiRegistry::TFlags flags = CNcbiRegistry::fNoOverride |
 CNcbiRegistry::fTruncate;
reg.Set("MySection", "MyName", " Not Overridden ", flags);
reg.Set("MySection", "MyName", " Not Saved ", flags);
cout << "[MySection]MyName=" << reg.Get("MySection", "MyName") << ".\n" <<
endl;

// outputs "[MySection]MyName=Not Overridden."
```

Main Methods of CNcbiRegistry

The CNcbiRegistry class constructor takes two arguments - an input stream to read the registry from (usually a file), and an optional TFlags argument, where the latter can be used to specify that all of the values should be stored as transient rather than in the default mode, which is persistent:

```
CNcbiRegistry(CNcbiIstream& is, TFlags flags = 0);
```

Once the registry has been initialized by its constructor, it is also possible to load additional parameters from other file(s) using the Read() method:

```
void Read(CNcbiIstream& is, TFlags flags = 0);
```

Valid flags for the Read() method include eTransient and eNoOverride. The default is for all values to be read in as persistent, with the capability of overriding any previously loaded value associated with the same name. Either or both of these defaults can be modified by specifying eTransient, eNoOverride, or (eTransient | eNoOverride) as the flags argument in the above expression.

The Write() method takes as its sole argument, a destination stream to which only the persistent configuration parameters will be written.

```
bool Write(CNcbiOstream& os) const;
```

The configuration parameter values can also be set directly inside your application, using:

```
bool Set(const string& section, const string& name,
 const string& value, TFlags flags = 0);
```

Here, valid flag values include ePersistent, eNoOverride, eTruncate, or any logical combination of these. If eNoOverride is set and there is a previously defined value for this parameter, then the value is not reset, and the method returns false.

The Get() method first searches the set of transient parameters for a parameter named name, in section section, and if this fails, continues by searching the set of persistent parameters. However, if the ePersistent flag is used, then only the set of persistent parameters will be searched. On success, Get() returns the stored value. On failure, the empty string is returned.

```
const string& Get(const string& section, const string& name,
 TFlags flags = 0) const;
```

Additional Registry Methods

Four additional note-worthy methods defined in the CNcbiRegistry interface are:

```
bool Empty(void) const;
void Clear(void);
void EnumerateSections(list<string>*sections) const;
void EnumerateEntries(const string& section, list<string>* entries) const;
```

Empty() returns true if the registry is empty. Clear() empties out the registry, discarding all stored parameters. EnumerateSections() writes all registry section names to the list of strings parameter named "sections". EnumerateEntries() writes the list of parameter names in section to the list of strings parameter named "entries".

Portable Stream Wrappers

Because of differences in the C++ standard stream implementations between different compilers and platforms, the file ncbistre.hpp contains portable aliases for the standard classes. To provide portability between the supported platforms, it is recommended the definitions in ncbistre.hpp be used.

The ncbistre.hpp defines wrappers for many of the standard stream classes and contains conditional compilation statements triggered by macros to include portable definitions. For example, not all compilers support the newer '#include <iostream>' form. In this case, the older '#include <iostream.h>' is used based on whether the macro NCBI_USE_OLD_IOSTREAM is defined.

Instead of using the iostream, istream or ostream, you should use the portable CNcbiIostream, CNcbiIstream and CNcbiOstream. Similarly, instead of using the standard cin, cout, cerr you can use the more portable NcbiCin, NcbiCout, and NcbiCerr.

The ncbistre.hpp also defines functions that handle platform-specific end of line reads. For example, Endl() represents platform specific end of line, and NcbiGetline() reads from a specified input stream to a string, and NcbiGetlineEOL() reads from a specified input stream to a string taking into account platform specific end of line.

Working with Diagnostic Streams (*)

This section provides reference information on the use of the diagnostic stream classes. For an overview of the diagnostic stream concepts refer to the introductory chapter.

The CNcbiDiag class implements the functionality of an output stream enhanced with error posting mechanisms similar to those found in the NCBI C Toolkit. A CNcbiDiag object has the look and feel of an output stream; its member functions and friends include output operators and format manipulators. A CNcbiDiag object is not itself a stream, but serves as an interface to a stream which allows multiple threads to write to the same output. Each instance of CNcbiDiag includes the following private data members:

- a buffer to store (a single) message text
- a severity level
- a set of post flags

Limiting each instance of CNcbiDiag to the storage and handling of a single message ensures that multiple threads writing to the same stream will not have interleaving message texts.

The following topics are discussed in this section:

Where Diagnostic Messages Go

The following decision tree describes how the destination for diagnostics messages is determined.

1. Before the application is constructed (before AppMain() is called), everything goes to:
    1. (Unix-like systems only) /log/fallback/UNKNOWN.{log|err|trace} -- if available
    2. STDERR -- otherwise
2. When the application is ready, and its name is known, but before the configuration file is loaded:
    1. If AppMain() is passed flags eDS_Default or eDS_ToStdlog, then the diagnostics goes:
        1. (Unix-like systems only) if /log is present:
            1. if the application is described in /etc/toolkitrc -- to /log/<token>/appname.{log|err|trace}
            2. else if environment variable $SERVER_PORT is set -- to /log/$SERVER_PORT/appname.{log|err|trace}
            3. else (or if failed to switch to one of the above two locations) -- to /log/srv/appname.{log|err|trace}
            4. or, if failed to switch to that -- to /log/fallback/appname.{ log|err|trace}
        2. else (or if failed to switch to any of the /log location):
            1. eDS_ToStdlog -- to <current_working_dir>/appname.{ log|err|trace} (and, if cannot, then continues to go to STDERR)
            2. eDS_Default -- continues to go to STDERR
    2. If AppMain() is passed flags other than eDS_Default or eDS_ToStdlog, then the diagnostics goes to:
        1. eDS_ToStdout -- standard output stream
        2. eDS_ToStderr -- standard error stream
        3. eDS_ToMemory -- the application memory
        4. eDS_Disable -- nowhere
        5. eDS_User -- wherever it went before the AppMain() call
        6. eDS_ToSyslog -- system log daemon
3. After the configuration file is loaded, and if it has an alternative location for the log files, then switch to logging to that location. See the list of logfile-related configuration parameters.

The boolean TryRootLogFirst argument in the [LOG] section of the application's config file changes the order of locations to be tested. If TryRootLogFirst is set, the application will try to open the log file under /log first. Only if this fails, then the location specified in the config file will be used.

Note:

- If the logging destination is switched, then a message containing both the old and new locations is logged to both locations.

- Before the application configuration is loaded, a copy of all diagnostics is saved in memory. If the log destination is changed by the application configuration, then the saved diagnostics are dumped to the final log destination.

Setting Diagnostic Severity Levels

Each diagnostic message has its own severity level (EDiagSev), which is compared to a global severity threshold to determine whether or not its message should be posted. Six levels of severity are defined by the EDiagSev enumeration:

```
/// Severity level for the posted diagnostics.
enum EDiagSev {
 eDiag_Info = 0, ///< Informational message
 eDiag_Warning, ///< Warning message
 eDiag_Error, ///< Error message
 eDiag_Critical, ///< Critical error message
 eDiag_Fatal, ///< Fatal error -- guarantees exit(or abort)
 eDiag_Trace, ///< Trace message
 // Limits
 eDiagSevMin = eDiag_Info, ///< Verbosity level for min. severity
 eDiagSevMax = eDiag_Trace ///< Verbosity level for max. severity
};
```

The default is to post only those messages whose severity level exceeds the eDiag_Warning level (i.e. eDiag_Error, eDiag_Critical, and eDiag_Fatal). The global severity threshold for posting messages can be reset using SetDiagPostLevel (EDiagSev postSev). A parallel function, SetDiagDieLevel (EDiagSev dieSev), defines the severity level at which execution will abort.

Tracing is considered to be a special, debug-oriented feature, and therefore messages with severity level eDiag_Trace are not affected by these global post/die levels. Instead, SetDiagTrace (EDiagTrace enable, EDiagTrace default) is used to turn tracing on or off. By default, the tracing is off - unless you assign the environment variable DIAG_TRACE to an arbitrary non-empty string or, alternatively, define a DIAG_TRACE entry in the [DEBUG] section of your registry file.

The severity level can be set directly in POST and TRACE statements, using the severity level manipulators including Info, Warning, Error, Critical, Fatal, and Trace, for example:

```
ERR_POST_X(1, Critical << "Something quite bad has happened.");
```

Diagnostic Messages Filtering

Diagnostic messages from the CNcbiDiag and CException classes can be filtered by the source file path; or by the module, class, or function name. Messages from the CNcbiDiag class can also be filtered by error code. If a CException object is created by chaining to a previous exception, then all exceptions in the chain will be checked against the filter and the exception will pass if any exception in the chain passes (even if one of them is suppressed by a negative condition). The filter can be set by the TRACE_FILTER or POST_FILTER entry in the [DIAG] section of the registry file or during runtime through SetDiagFilter(). Messages with a severity level of eDiag_Fatal are not filtered; messages with a severity level of eDiag_Trace are filtered by TRACE_FILTER; and all other messages are filtered by POST_FILTER. Filter strings contain filtering conditions separated by a space. An empty filter string means that all messages will appear in the log unfiltered. Filtering conditions are processed from left to right until a condition that matches the message is found. If the message does not match any of the conditions, then the message will be filtered out. Filtering conditions in the string may be preceded by an exclamation mark, which reverses the behavior (so if a message matches the condition it will be suppressed). See Table 4 for filtering condition samples and syntax.

For example:

- To log diagnostic messages from source files located in src/corelib with error codes from 101 to 106 and any subcode, use the following filter: “/corelib (101-106.)”.
- To exclude log messages from sources in src/serial and src/dbapi, use this filter: “!/serial !/dbapi”.
- To log messages from sources in src/serial excluding those with error code 802 and subcodes 4 and 10 through 12, and to exclude messages from sources in src/dbapi/driver, use the following filter: “/serial !(802.4,10-12) !/dbapi/driver”.

Log File Format

The format of the log file can be customized. One of the most basic choices is between the "old post format" and the "new post format". The old format essentially posts arbitrary strings whereas the new format adds many standard fields, and structures the messages so they can be automatically indexed for rapid searching and/or error statistics.

The old format is used by default. To use the new format:

```
int main(int argc, const char* argv[])
{
 GetDiagContext().SetOldPostFormat(false); // use the new format

 return CMyApp().AppMain(argc, argv);
}
```

This function should be called before the application's constructor for the setting to be used from the very beginning.

See also:

- the Diagnostic Trace section in the library configuration chapter for details on selecting the format using the environment or registry; and
- the ERR_POST and LOG_POST Macros section for more details on creating the log messages.

The Old Post Format

The old format for log messages is simply a message - prefixed with the severity level if it is an error message:

```
[<severity>: ]<Message>
```

The New Post Format

The new format for the application access log and error postings is:

```
<Common Prefix> <Event:13> <Message>
```

The common prefix has the format:

```
<pid:5>/<tid:3>/<rid:4>/<state:2> <guid:16> <psn:4>/<tsn:4> <time> <host:15>
<client:15> <session:24> <application>
```

Note: Width and padding of standard fields

- To make a good visual alignment, most numeric values are printed zero-padded to some minimal width. For example, <pid:5> means that number 123 gets printed as "00123", and number 1234567 gets printed as "1234567".
- The non-numeric fields for which the width is specified (e.g. <severity:10>) are padded with spaces and are adjusted to the left.

The fields are:

Field	**Description**	**Type or format**
pid	Process ID	Uint8 (decimal)
tid	Thread ID	Uint8 (decimal)
rid	Request ID (e.g. iteration number for a CGI)	int (decimal)
state	Application state code: { AB | AE | RB | R | RE }	string
guid	Globally unique process ID	Int8 (hexadecimal)
psn	Serial number of the posting within the process	int (decimal)
tsn	Serial number of the posting within the thread	int (decimal)
time	Astronomical date and time at which the message was posted	YYYY-MM-DD**T**hh:mm:ss.sss
host	Name of the host where the process runs	string (UNK_HOST if unknown)
client	Client IP address	valid IP address string (UNK_CLIENT if unknown)
session	Session ID	string (UNK_SESSION if unknown)
application	Name of the application (see note below)	string (UNK_APP if unknown)

Note: The application name is set to the executable name (without path and extension) by default. Sometimes however the executable's name can be too generic (like "summary" or "fetch"). To change it use CNcbiApplication::SetProgramDisplayName() function. Better yet, just rename the executable itself. It's a good practice to prefix the application names with something project-specific (like "pc_summary" for PubChem or "efetch" for E-Utils).

The application state codes are:

Code	**Meaning**
AB	application is starting
A	application is running (outside of any request)
AE	application is exiting
RB	request is starting
R	request is being processed
RE	request is exiting

The normal state transitions are:

- AB --> A --> AE
- AB --> A --> { RB --> R --> RE } --> A --> ... --> { RB --> R --> RE } --> A --> AE

The access log events and messages are:

Log Message		**Event / Description**
start		Start of application (see note below)
stop <exit_code> <timespan> [SIG=<exit_signal>]		End of application
where:	exit_code	Application exit code (zero if not set)
	timespan	Application execution time
	exit_signal	Signal number, if application exited due to a signal
extra		Arbitrary information (see note below)
request-start		Start of request (see note below)
request-stop <status> <timespan> <bytes_rd> <bytes_wr>		End of request
where:	status	Exit status of the request (zero if not set)
	timespan	Request execution time (zero if not set)
	bytes_rd	Input data read during the request execution, in bytes (zero if not set)
	bytes_wr	Output data written during the request execution, in bytes (zero if not set)

Note: Make your log data more parsable!

In many cases the logs are collected and stored in the database for analysis. The NCBI log system now implements a special logic to parse (and then index) the user data provided in the request-start and extra log lines. It is therefore recommended that this data be presented in the following format (which is understood by the parser):

```
tag1=value1&tag2=value2&tag3=value3...
```

where all tag and value fields are URL-encoded.

The format for error and trace messages is:

```
<severity:10>: <module>(<err_code>.<err_subcode> | <err_text>) "<file>", line
<line>: <class>::<func> --- <prefixes> <user_message> <err_code_message>
<err_code_explanation>
```

The error and trace message fields are:

Field	**Description**
severity	Message severity = { Trace | Info | Warning | Error | Critical | Fatal | Message[T|I|W|E|C|F] }
module	Module where the post originates from (in most cases the module corresponds to a single library)
err_code, err_subcode	Numeric error code and subcode
err_text	If the error has no numeric code, sometimes it can be represented as text

file, line	File name and line number where the posting occured
class, func	Class and/or function name where the posting occured: {Class:: | Class::Function() | ::Function()}
prefixes	User-defined prefixes for the message
user_message	The message itself
err_code_message	Short error code description
err_code_explanation	Detailed explanation of the error code

Example application events (line continuation characters added for clarity):

```
03960/000/0000/AB 2C2D0F7851AB7E40 0005/0005 2006-09-27T13:41:56.034 \
widget3 UNK_CLIENT UNK_SESSION cgi_sample.cgi \
start
03960/000/0000/RB 2C2D0F7851AB7E40 0008/0008 2006-09-27T13:41:56.456 \
widget3 192.168.0.2 2C2D0F7851AB7E40_0000SID cgi_sample.cgi \
request-start
03960/000/0000/RE 2C2D0F7851AB7E40 0010/0010 2006-09-27T13:41:56.567 \
widget3 192.168.0.2 2C2D0F7851AB7E40_0000SID cgi_sample.cgi \
request-stop 200 0.105005566
03960/000/0000/AE 2C2D0F7851AB7E40 0012/0012 2006-09-27T13:41:56.789 \
widget3 UNK_CLIENT UNK_SESSION cgi_sample.cgi \
stop 0 0.149036509
```

Example diagnostic message:

```
03960/000/0000/AB 2C2D0F7851AB7E40 0006/0006 2006-09-27T13:41:56.055 \
widget3 UNK_CLIENT UNK_SESSION cgi_sample.cgi \
Warning: CGI --- CCgiSampleApplication::Init()
03960/000/0000/R 2C2D0F7851AB7E40 0009/0009 2006-09-27T13:41:56.066 \
widget3 192.168.0.2 2C2D0F7851AB7E40_0000SID cgi_sample.cgi \
Warning: CGI --- CCgiSampleApplication::ProcessRequest()
15176/003/0006/R 2A763B485350C030 0098/0008 2006-10-17T12:59:47.333 \
widget3 192.168.0.2 2C2D0F7851AB7E40_0000SID my_app \
Error: TEST "/home/user/c++/src/corelib/test/my_app.cpp", \
line 81: CMyApp::Thread_Run() --- Message from thread 3, for request 6
```

Controlling the Appearance of Diagnostic Messages using Post Flags

The post flags define additional information that will be inserted into the output messages and appear along with the message body. The standard format of a message is:

```
"<file>", line <line>: <severity>: (<err_code>.<err_subcode>)
[<prefix1>::<prefix2>::<prefixN>] <message>\n
<err_code_message>\n
<err_code_explanation>
```

where the presence of each field in the output is controlled by the post flags EDiagPostFlag associated with the particular diagnostic message. The post flags are:

```
enum EDiagPostFlag {
 eDPF_File = 0x1, ///< Set by default #if _DEBUG; else not set
 eDPF_LongFilename = 0x2, ///< Set by default #if _DEBUG; else not set
 eDPF_Line = 0x4, ///< Set by default #if _DEBUG; else not set
 eDPF_Prefix = 0x8, ///< Set by default (always)
 eDPF_Severity = 0x10, ///< Set by default (always)
 eDPF_ErrorID = 0x20, ///< Module, error code and subcode
 eDPF_DateTime = 0x80, ///< Include date and time
 eDPF_ErrCodeMessage = 0x100, ///< Set by default (always)
 eDPF_ErrCodeExplanation = 0x200, ///< Set by default (always)
 eDPF_ErrCodeUseSeverity = 0x400, ///< Set by default (always)
 eDPF_Location = 0x800, ///< Include class and function
 ///< if any, not set by default
 eDPF_PID = 0x1000, ///< Process ID
 eDPF_TID = 0x2000, ///< Thread ID
 eDPF_SerialNo = 0x4000, ///< Serial # of the post, process-wide
 eDPF_SerialNo_Thread = 0x8000, ///< Serial # of the post, in the thread
 eDPF_RequestId = 0x10000, ///< fcgi iteration number or request ID
 eDPF_Iteration = 0x10000, ///< @deprecated
 eDPF_UID = 0x20000, ///< UID of the log

 eDPF_ErrCode = eDPF_ErrorID, ///< @deprecated
 eDPF_ErrSubCode = eDPF_ErrorID, ///< @deprecated
 /// All flags (except for the "unusual" ones!)
 eDPF_All = 0xFFFFF,

 /// Default flags to use when tracing.
#if defined(NCBI_THREADS)
 eDPF_Trace = 0xF81F,
#else
 eDPF_Trace = 0x581F,
#endif

 /// Print the posted message only; without severity, location, prefix, etc.
 eDPF_Log = 0x0,

 // "Unusual" flags -- not included in "eDPF_All"
 eDPF_PreMergeLines = 0x100000, ///< Remove EOLs before calling handler
 eDPF_MergeLines = 0x200000, ///< Ask diag.handlers to remove EOLs
 eDPF_OmitInfoSev = 0x400000, ///< No sev. indication if eDiag_Info
 eDPF_OmitSeparator = 0x800000, ///< No '---' separator before message

 eDPF_AppLog = 0x1000000, ///< Post message to application log
 eDPF_IsMessage = 0x2000000, ///< Print "Message" severity name.

 /// Hint for the current handler to make message output as atomic as
 /// possible (e.g. for stream and file handlers).
 eDPF_AtomicWrite = 0x4000000,

 /// Use global default flags (merge with).
 /// @sa SetDiagPostFlag(), UnsetDiagPostFlag(), IsSetDiagPostFlag()
 eDPF_Default = 0x10000000,

 /// Important bits which should be taken from the globally set flags
 /// even if a user attempts to override (or forgets to set) them
 /// when calling CNcbiDiag().
 eDPF_ImportantFlagsMask = eDPF_PreMergeLines |
 eDPF_MergeLines |
 eDPF_OmitInfoSev |
 eDPF_OmitSeparator |
 eDPF_AtomicWrite,

 /// Use flags provided by user as-is, do not allow CNcbiDiag to replace
 /// "important" flags by the globally set ones.
 eDPF_UseExactUserFlags = 0x20000000
};
```

The default message format displays only the severity level and the message body. This can be overridden inside the constructor for a specific message, or globally, using SetDiagPostFlag () on a selected flag. For example:

```
 SetDiagPostFlag(eDPF_DateTime); // set flag globally
```

Defining the Output Stream

The logging framework uses a global output stream. The default is to post messages to CERR ouput stream, but the stream destination can be reset at any time using:

```
SetDiagStream(CNcbiOstream* os, bool quick_flush,
 FDiagCleanup cleanup, void* cleanup_data)
```

This function can be called numerous times, thus allowing different sections of the executable to write to different files. At any given time however, all messages will be associated with the same global output stream. Because the messages are completely buffered, each message will appear on whatever stream is active at the time the message actually completes.

And, of course, you can provide (using SetDiagHandler) your own message posting handler CDiagHandler, which does not necessarily write the messages to a standard C++ output stream. To preserve compatibility with old code, SetDiagHandler also continues to accept raw callback functions of type FDiagHandler.

If the output stream is a file, you can optionally split the output into three streams, each written to a separate file:

- Application log - standard events (start, stop, request-start, request-stop and user defined extra events).
- Error log - all messages with severity Warning and above.
- Trace log - messages having severity Info and Trace messages.

All three log files have the same name but different extensions: .log, .err and .trace.

To turn on the log file splitting, call (before the log file initialization):

```
int main(int argc, const char* argv[])
{
 SetSplitLogFile(true);

 return CMyApp().AppMain(argc, argv);
}
```

This function should be called before the application's constructor for the setting to be used from the very beginning.

Tee Output to STDERR

Sometimes it is helpful to generate human-readable diagnostics on the console in addition to storing detailed diagnostics in the machine-parsable log files. In these cases, it is likely that both the message severity required to trigger output and the output format should be different for the log file and the console. For example:

Destination	**Severity**	**Format**
Log File	Error	new (machine-parsable)
Console	Warning	old (human-readable)

To set up this sort of tee, set these configuration parameters (see the library configuration chapter for details):

Configuration Parameter	**Example Value**	**Notes**
DIAG_TEE_TO_STDERR	True	This turns on the tee.
DIAG_OLD_POST_FORMAT	False	This makes the log file use the new format.
DIAG_POST_LEVEL	Error	This sets the minimum severity required to post to the log file.
DIAG_TEE_MIN_SEVERITY	Warning	This sets the minimum severity required to post to the console.

Alternatively, you can use the Console manipulator to indicate that output should go to the console (in human-readable format):

```
ERR_POST_X(1, Console << "My ERR_POST message.");
```

Note: Output sent to the console using this manipulator will also go to the log file if the message severity at least meets the severity threshold for the log file. The effect of the manipulator lasts until the next flush, which typically occurs after each post.

The Message Buffer

Diagnostic messages (i.e. instances of the CNcbiDiag class) have a buffer that is initialized when the message is first instantiated. Additional information can then be appended to the message using the overloaded stream operator <<. Messages can then be terminated explicitly using CNcbiDiag's stream manipulator Endm, or implicitly, when the CNcbiDiag object exits scope.

Implicit message termination also occurs as a side effect of applying one of the severity level manipulators. Whenever the severity level is changed, CNcbiDiag also automatically executes the following two manipulators:

- Endm -- the message is complete and the message buffer will be flushed

- Reset -- empty the contents of the current message buffer

When the message controlled by an instance of CNcbiDiag is complete, CNcbiDiag calls a global callback function (of type FDiagHandler) and passes the message (along with its severity level) as the function arguments. The default callback function posts errors to the currently designated output stream, with the action (continue or abort) determined by the severity level of the message.

Request Exit Status Codes

This section describes the possible values of the request exit codes used in NCBI. They appear in the application access log as:

```
request-stop <status> .....
```

Request exit status codes are either standard or NCBI-specific.

Standard (HTTP-like) status codes

The NCBI request exit codes must conform to the HTTP status codes:

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

NCBI-specific status codes

If the situation cannot be described using one of the standard (HTTP) status codes, then an NCBI specific code should be used.

The NCBI-specific status codes must be different from the standard (HTTP) status codes. At the same time these codes better follow at least the range requirements of the standard (HTTP) status codes, that is they better belong to one of the following ranges:

Range	Description
120 – 199	Informational/provisional response
220 – 299	Success
320 – 399	Redirection
420 – 499	Bad request (client error)
520 – 599	Server Error

So far we have the following NCBI specific status codes:

Value	Description
0	Unknown error
555	NCBI Network Dispatcher refused a request from and outside user which is in its "abusers list"
1000 + errno	Unclassifiable server error when only errno is known (NOTE: the value of errno can be different on different platforms!)

Error codes and their Descriptions

Error codes and subcodes are posted to an output stream only if applicable post flags were set. In addition to error codes, the logging framework can also post text explanations. The CDiagErrCodeInfo class is used to find the error message that corresponds to a given error code/subcode. Such descriptions could be specified directly in the program code or placed in a separate message file. It is even possible to use several such files simultaneously. CDiagErrCodeInfo can also read error descriptions from any input stream(s), not necessarily files.

Preparing an Error Message File

The error message file contains plain ASCII text data. We would suggest using the .msg extension, but this is not mandatory. For example, the message file for an application named SomeApp might be called SomeApp.msg.

The message file must contain a line with the keyword MODULE in it, followed by the name of the module (in our example SomeApp). This line must be placed in the beginning of the file, before any other declarations. Lines with symbol # in the first position are treated as comments and ignored.

Here is an example of the message file:

```
# This is a message file for application "SomeApp"
MODULE SomeApp
# ------ Code 1 ------
$$ NoMemory, 1, Fatal : Memory allocation error
# ------ Code 2 ------
$$ File, 2, Critical : File error
$^ Open, 1 : Error open a specified file
This often indicates that the file simply does not exist.
Or, it may exist but you do not have permission to access
the file in the requested mode.
$^ Read, 2, Error : Error read file
Not sure what would cause this...
$^ Write, 3, Critical
This may indicate that the filesystem is full.
# ------ Code 3 ------
$$ Math, 3
$^ Param, 20
$^ Range, 3
```

Lines beginning with $$ define a top-level error code. Similarly, lines beginning with $^ define subcodes of the top-level error code. In the above example Open is a subcode of File top-level error, which means the error with code 2 and subcode 1.

Both types of lines have similar structure:

```
$$/$^ <mnemonic_name>, <code> [, <severity> ] [: <message> ] \n
[ <explanation> ]
```

where

- mnemonic_name (*required*) Internal name of the error code/subcode. This is used as a part of an error name in a program code - so, it should also be a correct C/C++ identifier.
- code (*required*) Integer identifier of the error.

- severity (*optional*) This may be supplied to specify the severity level of the error. It may be specified as a severity level string (valid values are Info, Warning, Error, Critical, Fatal, Trace) or as an integer in the range from 0 (eDiag_Info) to 5 (eDiag_Trace). While integer values are acceptable, string values are more readable. If the severity level was not specified or could not be recognized, it is ignored, or inherited from a higher level (the severity of a subcode becomes the same as the severity of a top-level error code, which contains this subcode). As long as diagnostic eDPF_ErrCodeUseSeverity flag is set, the severity level specified in the message file overrides the one specified in a program, which allows for runtime customization. In the above example, Critical severity level will be used for all File errors, except Read subcode, which would have Error severity level.
- message (*optional*) Short description of the error. It must be a single-line message. As long as diagnostic eDPF_ErrCodeMessage flag is set, this message is posted as a part of the diagnostic output.
- explanation (*optional*) Following a top-level error code or a subcode definition string, it may be one or several lines of an explanation text. Its purpose is to provide additional information, which could be more detailed description of the error, or possible reasons of the problem. This text is posted in a diagnostic channel only if eDPF_ErrCodeExplanaton flag was set.

Error message files can be automatically read by setting a configuration parameter. You can either define the MessageFile entry in the DEBUG section of the application registry, or set the environment variable NCBI_CONFIG__DEBUG__MessageFile (note the double-underscores and character case).

Defining Custom Handlers using CDiagHandler

The user can install his own handler (of type CDiagHandler,) using SetDiagHandler(). CDiagHandler is a simple abstract class:

```
class CDiagHandler
{
public:
 /// Destructor.
 virtual ~CDiagHandler(void) {}
 /// Post message to handler.
 virtual void Post(const SDiagMessage& mess) = 0;
};
```

where SDiagMessage is a simple struct defined in ncbidiag.hpp whose data members' values are obtained from the CNcbiDiag object. The transfer of data values occurs at the time that Post is invoked. See also the section on Message posting for a more technical discussion.

The ERR_POST and LOG_POST Macros

A family of ERR_POST* macros and a corresponding family of LOG_POST* macros are available for routine error posting. Each family has a set of macros:

- {ERR|LOG}_POST(msg) – for posting a simple message. Note: these macros are deprecated. Use {ERR|LOG}_POST_X instead (except for tests) for more flexible error statistics and logging.
- {ERR|LOG}_POST_X(subcode, msg) – for posting a default error code, a given subcode, and a message. Each call to {ERR|LOG}_POST_X must use a different subcode for proper error statistics and logging. The default error code is selected by

NCBI_USE_ERRCODE_X. The error code is selected from those defined by NCBI_DEFINE_ERRCODE_X in the appropriate header file, e.g. include/corelib/error_codes.h.

- {ERR|LOG}_POST_EX(code, subcode, msg) – for posting a given error code, a given error subcode, and a message. This macro should only be used if you have to use a variable for the subcode, or to specify an error code other than the current default. In all other cases (except for tests), use {ERR|LOG}_POST_X for more flexible error statistics and logging.
- {ERR|LOG}_POST_XX(code, subcode, msg) – these macros must be used in place of {ERR|LOG}_POST_X within header files so that the same error code will be used for header-defined code, regardless of the error codes that including files may use.

The LOG_POST_* macros just write a string to the log file, and are useful if a human-readable log file is desired. The output from the ERR_POST_* macros is not easily read by humans, but facilitates automatic indexing for searching and/or error statistics. There are multiple flags to control the appearance of the message generated by the ERR_POST_* macros.

The LOG_POST_* and ERR_POST_* macros implicitly create a temporary CNcbiDiag object and put the passed "message" into it with a default severity of eDiag_Error. A severity level manipulator can be applied if desired, to modify the message's severity level. For example:

```
long lll = 345;
ERR_POST_X(1, "My ERR_POST message, print long: " << lll);
```

would write to the diagnostic stream something like:

```
Error: (1501.1) My ERR_POST message, print long: 345
```

while:

```
double ddd = 123.345;
ERR_POST_X(1, Warning << "My ERR_POST message, print double: " << ddd);
```

would write to the diagnostic stream something like:

```
Warning: (1501.1) My ERR_POST message, print double: 123.345
```

See the Log File Format section for more information on controlling the format of diagnostics messages.

Note: Most of the above macros make use of the macro definition NCBI_USE_ERRCODE_X. This definition must be present in your source code, and must be defined in terms of an existing error code name. By convention, error code names are defined in header file named error_codes.hpp in the relevant directory, for example include/corelib/error_codes.hpp.

To set up new error codes, pick appropriate names and error code numbers that don't match existing values, and decide how many subcodes you'll need for each error code. For example, the following sets up three error codes to deal with different categories of errors within a library, and specifies the number of subcodes for each category:

```
// Note: The following should be in src/app/my_prog/error_codes.hpp.
...
BEGIN_NCBI_SCOPE
```

```
...
NCBI_DEFINE_ERRCODE_X(MyLib_Cat1, 1501, 5);
NCBI_DEFINE_ERRCODE_X(MyLib_Cat2, 1502, 6);
NCBI_DEFINE_ERRCODE_X(MyLib_Cat3, 1503, 1);
// where:
// MyLib_* -- the error code names
// 1501, etc -- the error code numbers, typically starting at N*100+1
// 5, etc -- how many subcodes you need for the given error code
...
END_NCBI_SCOPE
```

Now you can use the error code in your library's implementation:

```
// The following should be in your source files.
...
// include the relevant error_codes header, for example:
#include <include/corelib/error_codes.hpp>
...
#define NCBI_USE_ERRCODE_X MyLib_Cat1 // sets the default error code for this
file
...
 ERR_POST_X(5, Critical << "Your message here."); // uses the default error
code
```

Generally, the default error code and the ERR_POST_X macro should be used. If it is necessary to use a non-default error code, that error code and the appropriate subcode may be used with the ErrCode manipulator in the ERR_POST macro. For example:

```
// use a non-default error code (1501 in this example) and subcode 3
ERR_POST(ErrCode(1501, 3) << "My error message.");
```

The _TRACE macro

The _TRACE(message) macro is a debugging tool that allows the user to insert trace statements that will only be posted if the code was compiled in debug mode, and provided that the tracing has been turned on. If DIAG_TRACE is defined as an environment variable, or as an entry in the [DEBUG] section of your configuration file (*.ini), the initial state of tracing is on. By default, if no such variable or registry entry is defined, tracing is off. SetDiagTrace (EDiagTrace enable, EDiagTrace default) is used to turn tracing on/off.

Just like ERR_POST, the _TRACE macro takes a message, and the message will be posted only if tracing has been enabled. For example:

```
SetDiagTrace(eDT_Disable);
_TRACE("Testing the _TRACE macro");
SetDiagTrace(eDT_Enable);
_TRACE("Testing the _TRACE macro AGAIN");
```

Here, only the second trace message will be posted, as tracing is disabled when the first _TRACE() macro call is executed.

Stack Traces

CStackTrace objects have special formatting: a "Stack trace:" line is added before the stack trace and standard indentation is used. This formatting is also used when printing the stack trace for exceptions.

Using stack traces with diagnostics is discussed in the following topics:

- Printing a Stack Trace
- Obtaining a Stack Trace for Exceptions

Printing a Stack Trace

A stack trace can be saved simply by creating a CStackTrace object. Then the object can be posted in an error message, for example:

```
ERR_POST_X(1, Error << "Your message here." << CStackTrace());
```

An example of a stack trace output on Linux:

```
Error: (1501.1) Your message here.
 Stack trace:
 ./my_prog ???:0 ncbi::CStackTraceImpl::CStackTraceImpl() offset=0x5D
 ./my_prog ???:0 ncbi::CStackTrace::CStackTrace(std::string const&)
offset=0x28
 ./my_prog ???:0 CMyProg::Run() offset=0xAF3
 ./my_prog ???:0 ncbi::CNcbiApplication::x_TryMain(ncbi::EAppMyProgStream,
char const*, int*, bool*) offset=0x6C8
 ./my_prog ???:0 ncbi::CNcbiApplication::AppMain(int, char const* const*,
char const* const*, ncbi::EAppMyProgStream, char const*, std::string const&)
offset=0x11BA
 ./my_prog ???:0 main offset=0x60
 /lib64/tls/libc.so.6 ???:0 __libc_start_main offset=0xEA
 ./my_prog ???:0 std::__throw_logic_error(char const*) offset=0x62
```

Obtaining a Stack Trace for Exceptions

The stack trace can be saved by CException and derived classes automatically if the exception's severity is equal to or above the level set in the EXCEPTION_STACK_TRACE_LEVEL environment variable or configuration parameter. The default level is eDiag_Critical, so that most exceptions do not save the stack trace (the default exception's severity is eDiag_Error).

When printing an exception, the diagnostics code checks if a stack trace is available and if so, automatically prints the stack trace along with the exception.

An example of an exception with a stack trace on Linux:

```
Error: (106.16) Application's execution failed
NCBI C++ Exception:
 Error: (CMyException::eMyErrorXyz) Your message here.
 Stack trace:
 ./my_prog ???:0 ncbi::CStackTraceImpl::CStackTraceImpl() offset=0x5D
 ./my_prog ???:0 ncbi::CStackTrace::CStackTrace(std::string const&)
offset=0x28
 ./my_prog ???:0 ncbi::CException::x_GetStackTrace() offset=0x86
 ./my_prog ???:0 ncbi::CException::x_Init(ncbi::CTestCompileInfo const&,
std::string const&, ncbi::CException const*, ncbi::ETestSev) offset=0xE9
 ./my_prog ???:0 ncbi::CException::CException(ncbi::CTestCompileInfo const&,
ncbi::CException const*, ncbi::CException::EErrCode, std::string const&,
ncbi::ETestSev) offset=0x119
 ./my_prog ???:0 CMyException::CMyException(ncbi::CTestCompileInfo const&,
ncbi::CException const*, CMyException::EErrCode, std::string const&,
ncbi::ETestSev) offset=0x43
 ./my_prog ???:0 CMyTestTest::Run() offset=0xD3A
 ./my_prog ???:0 ncbi::CNcbiApplication::x_TryMain(ncbi::EAppTestStream, char
const*, int*, bool*) offset=0x6C8
 ./my_prog ???:0 ncbi::CNcbiApplication::AppMain(int, char const* const*,
char const* const*, ncbi::EAppTestStream, char const*, std::string const&)
offset=0x11BA
 ./my_prog ???:0 main offset=0x60
 /lib64/tls/libc.so.6 ???:0 __libc_start_main offset=0xEA
 ./my_prog ???:0 std::__throw_logic_error(char const*) offset=0x62
```

Debug Macros

A number of debug macros such as _TRACE, _TROUBLE, _ASSERT, _VERIFY, _DEBUG_ARG can be used when the _DEBUG macro is defined.

These macros are part of CORELIB. However, they are discussed in a separate chapter on Debugging, Exceptions, and Error Handling.

Handling Exceptions

The CORELIB defines an extended exception handling mechanism based on the C++ std::exception, but which considerably extends this mechanism to provide a backlog, history of unfinished tasks, and more meaningful reporting on the exception itself.

While the extended exception handling mechanism is part of CORELIB, it is discussed in a separate chapter on Debugging, Exceptions, and Error Handling.

Defining the Standard NCBI C++ types and their Limits

The following section provides a reference to the files and limit values used to in the C++ Toolkit to write portable code. An introduction to the scope of some of these portability definitions is presented the introduction chapter.

The following topics are discussed in this section:

- Headers Files containing Portability Definitions
- Built-in Integral Types
- Auxiliary Types
- Fixed-size Integer Types
- The "Ncbi_BigScalar" Type
- Encouraged and Discouraged Types

Headers Files containing Portability Definitions

- corelib/ncbitype.h -- definitions of NCBI fixed-size integer types

- corelib/ncbi_limits.h -- numeric limits for:
  - NCBI fixed-size integer types
  - built-in integer types
  - built-in floating-point types
- corelib/ncbi_limits.hpp -- temporary (and incomplete) replacement for the Standard C ++ Template Library's API

Built-in Integral Types

We encourage the use of standard C/C++ types shown in Table 5, and we state that the following assumptions (no less, no more) on their sizes and limits will be valid for all supported platforms:

Auxiliary Types

Use type "bool" to represent boolean values. It accepts one of { false, true }.

Use type "size_t" to represent a size of memory structure, e.g. obtained as a result of sizeof operation.

Use type "SIZE_TYPE" to represent a size of standard C++ "string" - this is a portable substitution for "std::string::size_type".

Fixed-size Integer Types

Sometimes it is necessary to use an integer type which:

- has a well-known fixed size(and lower/upper limits)
- be just the same on all platforms(but maybe a byte/bit order, depending on the processor architecture)

NCBI C++ standard headers provide the fixed-size integer types shown in Table 6:

In Table 7, the "kM*_*" are constants of relevant fixed-size integer type. They are guaranteed to be equal to the appropriate preprocessor constants from the old NCBI C headers ("INT*_M*"). Please also note that the mentioned "INT*_M*" are not defined in the C++ headers - in order to discourage their use in the C++ code.

The "Ncbi_BigScalar" Type

NCBI C++ standard headers also define a special type "Ncbi_BigScalar". The only assumption that can be made(and used in your code) is that "Ncbi_BigScalar" variable has a size which is enough to hold any integral, floating-point or pointer variable like "Int8", or "double"("long double"), or "void*". This type can be useful e.g. to hold a callback data of arbitrary fundamental type; however, in general, the use of this type is discouraged.

Encouraged and Discouraged Types

For the sake of code portability and for better compatibility with the third-party and system libraries, one should follow the following set of rules:

- Use standard C/C++ integer types "char", "signed char", "unsigned char", "short", "unsigned short", "int", "unsigned int" in **any** case where the assumptions made for them in Table 5 are enough.
- It is not recommended to use "long" type unless it is absolutely necessary (e.g. in the lower-level or third-party code), and even if you have to, then try to localize the use of "long" as much as possible.

- The same(as for "long") is for the fixed-size types enlisted in Table 6. If you have to use these in your code, try to keep them inside your modules and avoid mixing them with standard C/C++ types (as in assignments, function arg-by-value passing and in arithmetic expressions) as much as possible.
- For the policy on other types see in sections "Auxiliary types" and "Floating point types".

Understanding Smart Pointers: the CObject and CRef Classes

This section provides reference information on the use of CRef and CObject classes. For an overview of these classes refer to the introductory chapter.

The following is a list of topics discussed in this section:

STL auto_ptrs

C programmers are well-acquainted with the advantages and pitfalls of using pointers. As is often the case, the good news is also the bad news:

- memory can be dynamically allocated as needed, but may not be deallocated as needed, due to unanticipated execution paths;
- void pointers allow heterogeneous function arguments of different types, but type information may not be there when you need it.

C++ adds some additional considerations to pointer management: STL containers cannot hold reference objects, so you are left with the choice of using either pointers or copies of objects. Neither choice is attractive, as pointers can cause memory leaks and the copy constructor may be expensive.

The idea behind a C++ smart pointer is to create a wrapper class capable of holding a pointer. The wrapper class's constructors and destructors can then handle memory management as the object goes in and out of scope. The problem with this solution is that it does not handle multiple pointers to the same resource properly, and it raises the issue of ownership. This is essentially what the auto_ptr offers, but this strategy is only safe to use when the resource maps to a single pointer variable.

For example, the following code has two very serious problems:

```
int* ip = new int(5);
{
 auto_ptr<int> a1(ip);
 auto_ptr<int> a2(ip);
}
*ip = 10/(*ip);
```

The first problem occurs inside the block where the two auto_ptrs are defined. Both are referencing the same variable pointed to by yet another C pointer, and each considers itself to be the owner of that reference. Thus, when the block is exited, the delete[] operation is executed twice for the same pointer.

Even if this first problem did not occur - for example if only one auto_ptr had been defined - the second problem occurs when we try to dereference ip. The delete operation occurring as the block exits has now freed the dynamic memory to which ip points, so *ip now references unallocated memory.

The problem with using auto_ptr is that it provides semantics of strict ownership. When an auto_ptr is destructed, it deletes the object it points to, and therefore the object should not be pointed to simultaneously by others. Two or more auto_ptrs should not own the same object; that is, point to the same object. This can occur if two auto_ptrs are initialized to the same object, as seen in the above example where auto pointers a1 and a2 are both initialized with ip. In using auto_ptr, the programmer must ensure that situations similar to the above do not occur.

The CRef (*) Class

These issues are addressed in the NCBI C++ Toolkit by using reference-counted smart pointers: a resource cannot be deallocated until **all** references to it have ceased to exist. The implementation of a smart pointer in the NCBI C++ Toolkit is actually divided between two classes: CRef and CObject.

The CRef class essentially provides a pointer interface to a CObject, while the CObject actually stores the data and maintains the reference count to it. The constructor used to create a new CRef pointing to a particular CObject automatically increments the object's reference count. Similarly, the CRef destructor automatically decrements the reference count. In both cases however, the modification of the reference count is implemented by a member function of the CObject. The CRef class itself does not have direct access to the reference count and contains only a single data member - its pointer to a CObject. In addition to the CRef class's constructors and destructors, its interface to the CObject pointer includes access/mutate functions such as:

```
bool Empty()
bool NotEmpty()
CObject* GetPointer()
CObject& GetObject()
CObject* Release()
void Reset(CObject* newPtr)
void Reset(void)
operator bool()
bool operator!()
CRefBase& operator=(const CRefBase& ref)
```

Both the Release() and Reset() functions set the CRef object's m_ptr to 0, thus effectively removing the reference to its CObject. There are important distinctions between these two functions however. The Release() method removes the reference without destroying the object, while the Reset() method may lead to the destruction of the object if there are no other references to it.

If the CObject's internal reference count is 1 at the time Release() is invoked, that reference count will be decremented to 0, and a pointer to the CObject is returned. The Release() method can throw two types of exceptions: (1) a null pointer exception if m_ptr is already 0, and (2) an Illegal CObject::ReleaseReference() exception if there are currently other references to that object. An object must be free of all references (but this one) before it can be "released". In contrast, the Reset(void) function simply resets the CRef's m_ptr to 0, decrements the CObject's reference count, and, if the CObject has no other references and was dynamically allocated, deletes the CObject.

Each member function of the CRef class also has a const implementation that is invoked when the pointer is to a const object. In addition, there is also a CConstRef class that parallels the CRef class. Both CRef and CConstRef are implemented as template classes, where the template argument specifies the type of object which will be pointed to. For example, in the section on Traversing an ASN.1 Data Structure we examined the structure of the CBiostruc class and found the following type definition

```
typedef list< CRef< ::CBiostruc_id > > TId;
```

As described there, this typedef defines TId to be a list of pointers to CBiostruc_id objects. And as you might expect, CBiostruc_id is a specialized subclass of CObject.

The CObject (*) Class

The CObject class serves as a base class for all objects requiring a reference count. There is little overhead entailed by deriving a new class from this base class, and most objects in the NCBI C++ Toolkit are derived from the CObject class. For example, CNCBINode is a direct descendant of CObject, and all of the other HTML classes descend either directly or indirectly from CNCBINode. Similarly, all of the ASN.1 classes defined in the include/objects directory, as well as many of the classes defined in the include/serial directory are derived either directly or indirectly from the CObject class.

The CObject class contains a single private data member, the reference counter, and a set of member functions which provide an interface to the reference counter. As such, it is truly a base class which has no stand-alone utility, as it does not even provide allocation for data values. It is the descendant classes, which inherit all the functionality of the CObject class, that provide the necessary richness in representation and allocation required for the widely diverse set of objects implemented in the NCBI C++ Toolkit. Nevertheless, it is often necessary to use smart pointers on simple data types, such as int, string etc. The CObjectFor class, described below, was designed for this purpose.

The CObjectFor (*) class: using smart pointers for standard types

The CObjectFor class is derived directly from CObject, and is implemented as a template class whose argument specifies the standard type that will be pointed to. In addition to the reference counter inherited from its parent class, CObjectFor has a private data member of the parameterized type, and a member function GetData() to access it.

An example program, smart.cpp, uses the CRef and CObjectFor classes, and demonstrates the differences in memory management that arise using auto_ptr and CRef. Using an auto_ptr to reference an int, the program tests whether or not the reference is still accessible after an auxilliary auto_ptr which goes out of scope has also been used to reference it. The same sequence is then tested using CRef objects instead.

In the first case, the original auto_ptr, orig_ap, becomes NULL at the moment when ownership is transferred to copy_ap by the copy constructor. Using CRef objects however, the reference contained in the original CRef remains accessible (via orig) in all blocks where orig is defined. Moreover, the reference itself, i.e. the object pointed to, continues to exist until **all** references to it have been removed.

When to use CRefs and auto_ptrs

There is some overhead in using CRef and auto_ptr, and these objects should only be used where needed. Memory leaks are generally caused as a result of unexpected execution paths. For example:

```
{
 int *num = new int(5);
 ComplexFunction (num);
 delete num;
 ...
}
```

If ComplexFunction() executes normally, control returns to the block where it was invoked, and memory is freed by the delete statement. Unforeseen events however, may trigger exceptions, causing control to pass elsewhere. In these cases, the delete statement may never be reached. The use of a CRef or an auto_ptr is appropriate for these situations, as they both guarantee that the object will be destroyed when the reference goes out of scope.

One situation where they may not be required is when a pointer is embedded inside another object. If that object's destructor also handles the deallocation of its embedded objects, then it is sufficient to use a CRef on the containing object only.

CRef Pitfalls

Inadvertent Object Destruction

When the last reference to a CRef object goes out of scope or the CRef is otherwise marked for garbage collection, the object to which the CRef points is also destroyed. This feature helps to prevent memory leaks, but it also requires care in the use of CRefs within methods and functions.

```
class CMy : public CObject
{
.....
};
void f(CMy* a)
{
 CRef b = a;
 return;
}
.....
 CMy* a = new CMy();
 f(a);
 // the object "a" is now destroyed!
```

In this example the function f() establishes a local CRef to the CMy object a. On exiting f() the CRefb is destroyed, including the implied destruction of the CMy objects a. To avoid this behavior, pass a CRef to the function f() instead of a normal pointer variable:

```
CRef a = new CMy();
f(a);
// the CMy object pointed to by "a" is not destroyed!
```

Atomic Counters

The CORELIB implements efficient atomic counters that are used for CObject reference counts. The classes CAtomicCounter and CMutableAtomicCounter provide respectively a base atomic counter class, and a mutable atomic counter for multithreaded applications. These classes are used to in reference counted smart pointers.

The CAtomicCounter base class provides the base methods Get(), Set(), Add() for atomic counters:

```
class CAtomicCounter
{
public:
 ///< Alias TValue for TNCBIAtomicValue
 typedef TNCBIAtomicValue TValue;
 /// Get atomic counter value.
 TValue Get(void) const THROWS_NONE;
 /// Set atomic counter value.
 void Set(TValue new_value) THROWS_NONE;
 /// Atomically add value (=delta), and return new counter value.
 TValue Add(int delta) THROWS_NONE;
 .......
};
```

TNCBIAtomicValue is defined as a macro and its definition is platform dependent. If threads are not used (Macro NCBI_NO_THREADS defined), TNCBIAtomicValue is an unsigned int value. If threads are used, then a number of defines in file "ncbictr.hpp" ensure that a platform specific definition is selected for TNCBIAtomicValue.

The CMutableAtomicCounter uses the CAtomicCounter as its internal structure of the atomic counter but declares this counter value as mutable. The Get(), Set(), Add() methods for CMutableAtomicCounter are implemented by calls to the corresponding Get(), Set(), Add() methods for the CAtomicCounter:

```
class CMutableAtomicCounter
{
public:
 typedef CAtomicCounter::TValue TValue; ///< Alias TValue simplifies syntax
 /// Get atomic counter value.
 TValue Get(void) const THROWS_NONE
 { return m_Counter.Get(); }
 /// Set atomic counter value.
 void Set(TValue new_value) const THROWS_NONE
 { m_Counter.Set(new_value); }
 /// Atomically add value (=delta), and return new counter value.
 TValue Add(int delta) const THROWS_NONE
 { return m_Counter.Add(delta); }
private:
 ...
};
```

Portable mechanisms for loading DLLs

The CDll class defines a portable way of dynamically loading shared libraries and finding entry points for functions in the library. Currently this portable behavior is defined for Unix-like systems and Windows only. On Unix-like systems, loading of the shared library is implemented using the Unix system call dlopen() and the entry point address obtained using the Unix system call dlsym(). On Windows systems the system call LoadLibraray() is used to load the library, and the system call GetProcAddress() is used to get a function's entry point address.

All methods of CDll class, except the destructor, throw the exception CCoreException::eDll on error.

You can specify when to load the DLL - when the CDll object is created (loading in the constructor), or by an explicit call to CDll::Load(). You can also specify whether the DLL is unloaded automatically when CDll's destructor is called or if the DLL should remain loaded in memory. This behavior is controlled by arguments to CDll's constructor.

The following additional topics are described in this section:

- CDll Constructor
- CDll Basename
- Other CDll Methods

CDll Constructor

The CDll class has four constructors:

```
CDll(const string& name, TFlags flags);
```

and

```
CDll(const string& path, const string& name, TFlags flags);
```

and

```
CDll(const string& name,
ELoad when_to_load = eLoadNow,
EAutoUnload auto_unload = eNoAutoUnload,
EBasename treate_as = eBasename);
```

and

```
CDll(const string& path, const string& name,
ELoad when_to_load = eLoadNow,
EAutoUnload auto_unload = eNoAutoUnload,
EBasename treate_as = eBasename);
```

The first and second constructor forms are the same with the exception that the second constructor uses two parameters - the path and name parameters - to build a path to the DLL, whereas in the first constructor, the name parameter contains the full path to the DLL. The third and fourth forms are likewise similar.

The first pair of constructors differ from the second pair in that the first two take a single parameter that is a set of flags, whereas the second pair take three separate parameters for flags. The first two are newer, and the last two are provided for backward compatibility.

The parameter when_to_load is defined as an enum type of ELoad and has the values eLoadNow or eLoadLater. When eLoadNow is passed to the constructor (default value), the DLL is loaded in the constructor; otherwise it has to be loaded via an explicit call to the Load () method.

The parameter auto_load is defined as an enum type of EAutoLoad and has the values eAutoUnload or eNoAutoUnload. When eAutoUnload is passed to the constructor (default value), the DLL is unloaded in the destructor; otherwise it will remain loaded in memory.

The parameter treat_as is defined as an enum type of EBasename and has the values eBasename or eExactName. When eBasename is passed to the constructor (default value), the name parameter is treated as a basename if it looks like one; otherwise the exact name or "as is" value is used with no addition of prefix or suffix.

The parameter flags is defined as an enum type of EFlags and has the values fLoadNow, fLoadLater, fAutoUnload, fNoAutoUnload, fBaseName, fExactName, fGlobal, fLocal, and fDefault. The flags fLoadNow, fLoadLater, fAutoUnload, fNoAutoUnload, fBaseName, and fExactName correspond to the similarly named enum values as described in the above paragraphs. The flag fGlobal indicates that the DLL should be loaded as RTLD_GLOBAL, while the flag fLocal indicates that the DLL should be loaded as RTLD_LOCAL. The flag fDefault is defined as:

```
fDefault = fLoadNow | fNoAutoUnload | fBaseName | fGlobal
```

CDll Basename

The DLL name is considered the basename if it does not contain embedded '/', '\', or ':' symbols. Also, in this case, if the DLL name does not match the pattern "lib*.so", "lib*.so.*", or "*.dll" and if eExactName flag is not passed to the constructor, then it will be automatically transformed according to the following rules:

OS	**Rule**
Unix-like	<name> -> lib<name>.so
Windows	<name> -> <name>.dll

If the DLL is specified by its basename, then it will be searched after the transformation described above in the following locations:

- Unix:
  - The directories that are listed in the LD_LIBRARY_PATH environment variable which are analyzed once at the process startup.
  - The directory from which the application loaded
  - Hard-coded (e.g. with `ldconfig' on Linux) paths
- Windows:
  - The directory from which the application is loaded
  - The current directory
  - The Windows system directory

- The Windows directory
- The directories that are listed in the PATH environment variable

Other CDll Methods

Two methods mentioned earlier for the CDll class are the Load() and Unload() methods. The Load() method loads the DLL using the name specified in the constructor's DLL name parameter. The Load() method is expected to be used when the DLL is not explictly loaded in the constructor. That is, when the CDll constructor is passed the eLoadLater parameter. If the Load() is called more than once without calling Unload() in between, then it will do nothing. The syntax of the Load() methods is

```
void Load(void);
```

The Unload() method unloads that DLL whose name was specified in the constructor's DLL name parameter. The Unload() method is expected to be used when the DLL is not explicitly unloaded in the destructor. This occurs, when the CDll constructor is passed the eNoAutoUnload parameter. If the Unload() is called when the DLL is not loaded, then it will do nothing. The syntax of the Unload() methods is

```
void Unload(void);
```

Once the DLL is loaded, you can call the DLL's functions by first getting the function's entry point (address), and using this to call the function. The function template GetEntryPoint() method is used to get the entry point address and is defined as:

```
template <class TPointer>
TPointer GetEntryPoint(const string& name, TPointer* entry_ptr);
```

This method returns the entry point's address on success, or NULL on error. If the DLL is not loaded when this method is called, then this method will call Load() to load the DLL which can result in throwing an exception if Load() fails.

Some sample code illustrating the use of these methods is shown in src/corelib/test/test_ncbidll.cpp

Executing Commands and Spawning Processes using the CExec class

The CExec defines a portable execute class that can be used to execute system commands and spawn new processes.

The following topics relating to the CExec class are discussed, next:

- Executing a System Command using the System() Method
- Defining Spawned Process Modes (EMode type)
- Spawning a Process using SpawnX() Methods
- Waiting for a Process to Terminate using the Wait() method

Executing a System Command using the System() Method

You can use the class-wide CExec::System() method to execute a system command:

```
static int System(const char* cmdline);
```

CExec::System() returns the executed command's exit code and throws an exception if the command failed to execute. If cmdline is a null pointer, CExec::System() checks if the shell (command interpreter) exists and is executable. If the shell is available, System() returns a non-zero value; otherwise, it returns 0.

Defining Spawned Process Modes (EMode type)

The spawned process can be created in several modes defined by the enum type EMode. The meanings of the enum values for EMode type are:

- eOverlay: This mode overlays the calling process with new process, destroying the calling process.
- eWait: This mode suspends the calling thread until execution of a new process is complete. That is, the called process is called synchronously.
- eNoWait: This is the opposite of eWait. This mode continues to execute the calling process concurrently with the new called process in an asynchronous fashion.
- eDetach: This mode continues to execute the calling process and new process is "detached" and run in background with no access to console or keyboard. Calls to Wait () against new process will fail. This is an asynchronous spawn.

Spawning a Process using SpawnX() Methods

A new process can be spawned by calling any of the class-wide methods named SpawnX() which have the form:

```
static int SpawnX(const EMode mode,
 const char *cmdname,
 const char *argv,
 ...
 );
```

The parameter mode has the meanings discussed in the section Defining Spawned Process Modes (EMode type). The parameter cmdname is the command-line string to start the process, and parameter argv is the argument vector containing arguments to the process.

The X in the function name is a one to three letter suffix indicating the type of the spawn function. Each of the letters in the suffix X, for SpawnX() have the following meanings:

- L: The letter "L" as suffix refers to the fact that command-line arguments are passed separately as arguments.
- E: The letter "E" as suffix refers to the fact that environment pointer, envp, is passed as an array of pointers to environment settings to the new process. The NULL environment pointer indicates that the new process will inherit the parents' process's environment.
- P: The letter "P" as suffix refers to the fact that the PATH environment variable is used to find file to execute. Note that on a Unix-like system this feature works in functions without letter "P" in the function name.
- V: The letter "V" as suffix refers to the fact that the number of command-line arguments is variable.

Using the above letter combinations as suffixes, the following spawn functions are defined:

- SpawnL(): In the SpawnL() version, the command-line arguments are passed individually. SpawnL() is typically used when number of parameters to the new process is known in advance.

- SpawnLE(): In the SpawnLE() version, the command-line arguments and environment pointer are passed individually. SpawnLE() is typically used when number of parameters to the new process and individual environment parameter settings are known in advance.
- SpawnLP(): In the SpawnLP() version, the command-line arguments are passed individually and the PATH environment variable is used to find the file to execute. SpawnLP() is typically used when number of parameters to the new process is known in advance but the exact path to the executable is not known.
- SpawnLPE(): In the SpawnLPE() the command-line arguments and environment pointer are passed individually, and the PATH environment variable is used to find the file to execute. SpawnLPE() is typically used when the number of parameters to the new process and individual environment parameter settings are known in advance, but the exact path to the executable is not known.
- SpawnV(): In the SpawnV() version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1.
- SpawnVE(): In the SpawnVE() version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1. The individual environment parameter settings are known in advance and passed explicitly.
- SpawnVP(): In the SpawnVP() version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1. The PATH environment variable is used to find the file to execute.
- SpawnVPE(): In the SpawnVPE() version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1. The PATH environment variable is used to find the file to execute, and the environment is passed via an environment vector pointer.

Refer to the include/corelib/ncbiexec.hpp file to view the exact form of the SpawnX() function calls.

Some sample code illustrating the use of these methods is shown in src/corelib/test/test_ncbiexec.cpp

Waiting for a Process to Terminate using the Wait() method

The CExec class defines a Wait() method that causes a process to wait until the child process terminates:

```
static int Wait(const int pid);
```

The argument to the Wait() method is the pid (process ID) of the child process on which the caller is waiting to terminate. Wait() returns immediately if the specified child process has already terminated and returns an exit code of the child process, if there are no errors; or a -1, if an error has occurred.

Implementing Parallelism using Threads and Synchronization Mechanisms

This section provides reference information on how to add multithreading to your application and how to use basic synchronization objects. For an overview of these concepts refer to the introductory topic on this subject.

Note that all classes are defined in include/corelib/ncbithr.hpp and include/corelib/ncbimtx.hpp.

The following topics are discussed in this section:

Using Threads

CThread class is defined in include/corelib/ncbithr.hpp. The CThread class provides all basic thread functionality: thread creation, launching, termination, and cleanup. To create user-defined thread one needs only to provide the thread's Main() function and, in some cases, create a new constructor to transfer data to the thread object, and override OnExit() method for thread-specific data cleanup. To create a custom thread:

1. Derive your class from CThread, override Main() and, if necessary, OnExit() methods.
2. Create thread object in your application. You can do this only with new operator, since static or in-stack thread objects are prohibited (see below). The best way to reference thread objects is to use CRef<CThread> class.
3. Call Run() to start the thread execution.
4. Call Detach() to let the thread run independently (it will destroy itself on termination then), or use Join() to wait for the thread termination.

The code should look like:

```
#include <corelib/ncbistd.hpp>
#include <corelib/ncbithr.hpp>
USING_NCBI_SCOPE;
class CMyThread : public CThread
{
public:
 CMyThread(int index) : m_Index(index) {}
 virtual void* Main(void);
 virtual void OnExit(void);
private:
 int m_Index;
 int* heap_var;
};
void* CMyThread::Main(void)
{
 cout << "Thread " << m_Index << endl;
 heap_var = new int; // to be destroyed by OnExit()
 *heap_var = 12345;
 int* return_value = new int; // return to the main thread
 *return_value = m_Index;
 return return_value;
}
void CMyThread::OnExit(void)
{
 delete heap_var;
}
int main(void)
{
 CMyThread* thread = new CMyThread(33);
 thread->Run();
 int* result;
 thread->Join(reinterpret_cast<void**>(&result));
 cout << "Returned value: " << *result << endl;
 delete result;
 return 0;
}
```

The above simple application will start one child thread, passing 33 as the index value. The thread prints "Thread 33" message, allocates and initializes two integer variables, and terminates. The thread's Main() function returns a pointer to one of the allocated values. This pointer is then passed to Join() method and can be used by another thread. The other integer allocated by Main() is destroyed by OnExit() method.

It is important not to terminate the program until there are running threads. Program termination will cause all the running threads to terminate also. In the above example Join() function is used to wait for the child thread termination.

The following subsections discuss the individual classes in more detail.

CThread (*) class public methods

CThread(void) Create the thread object (without running it). bool Run(void) Spawn the new thread, initialize internal CThread data and launch user-provided Main(). The method guarantees that the new thread will start before it returns to the calling function. void Detach (void) Inform the thread that user does not need to wait for its termination. Detached thread will destroy itself after termination. If Detach() is called for a thread, which has already terminated, it will be scheduled for destruction immediately. Only one call to Detach() is allowed for each thread object. void Join(void** exit_data) Wait for the thread termination. Join() will store the void pointer as returned by the user's Main() method, or passed to the Exit () function to the exit_data. Then the thread will be scheduled for destruction. Only one call to Join() is allowed for each thread object. If called more than once, Join() will cause a runtime error. static void Exit(void* exit_data) This function may be called by a thread object itself to terminate the thread. The thread will be terminated and, if already detached, scheduled for destruction. exit_data value is transferred to the Join() function as if it was returned by the Main(). Exit() will also call virtual method OnExit() to execute user-provided cleanup code (if any). bool Discard(void) Schedules the thread object for destruction if it has not been run yet. This function is provided since there is no other way to delete a thread object without running it. On success, return true. If the thread has already been run, Discard() do nothing and return false. static CThread::TID GetSelf(void) This method returns a unique thread ID. This ID may be then used to identify threads, for example, to track the owner of a shared resource. Since the main thread has no associated CThread object, a special value of 0 (zero) is reserved for the main thread ID.

CThread (*) class protected methods

virtual void* Main(void)Main() is the thread's main function (just like an application main() function). This method is not defined in the CThread class. It must be provided by derived user-defined class. The return value is passed to the Join() function (and thus may be used by another thread for some sort of inter-thread communication). virtual void OnExit(void) This method is called (in the context of the thread) just before the thread termination to cleanup thread-specific resources. OnExit() is NOT called by Discard(), since the thread has not been run in this case and there are no thread-specific data to destroy. virtual ~CThread(void) The destructor is protected to avoid thread object premature destruction. For this reason, no thread object can be static or stack-allocated. It is important to declare any CThread derived class destructor as protected.

Thread Life Cycle

Figure 2 shows a typical thread life cycle. The figure demonstrates that thread constructors are called from the parent thread. The child thread is spawned by the Run() function only. Then, the user-provided Main() method (containing code created by user) gets executed. The thread's destructor may be called in the context of either parent or child thread depending on the state of the thread at the moment when Join() or Detach() is called.

There are two possible ways to terminate a thread. By default, after user-provided Main() function return, the Exit() is called implicitly to terminate the thread. User functions can call CThread::Exit() directly. Since Exit() is a static method, the calling function does not need to be a thread class member or have a reference to the thread object. Exit() will terminate the thread in which context it is called.

The CThread destructor is protected. The same must be true for any user-defined thread class in order to prohibit creation of static or automatic thread objects. For the same reason, a thread object can not be destroyed by explicit delete. All threads destroy themselves on termination, detaching, or joining.

On thread termination, Exit() checks if the thread has been detached and, if this is true, destroys the thread object. If the thread has not been detached, the thread object will remain "zombie" unless detached or joined. Either Detach() or Join() will destroy the object if the thread has been terminated. One should keep in mind, that it is not safe to use the thread object after a call to Join() or Detach() since the object may happen to be destroyed. To avoid this situation, the CRef<CThread> can be used. The thread object will not be destroyed until there is at least one CRef to the object (although it may be terminated and scheduled for destruction).

In other words, a thread object will be destroyed when all of the following conditions are satisfied:

- the thread has been run and terminated by an implicit or explicit call to Exit()
- the thread has been detached or joined
- no CRef references the thread object

Which thread will actually destroy a thread object depends on several conditions. If the thread has been detached before termination, the Exit() method will destroy it, provided there are no CRef references to the object. When joined, the thread will be destroyed in the context of a joining thread. If Detach() is called after thread termination, it will destroy the thread in the context of detaching thread. And, finally, if there are several CRef objects referencing the same thread, it will be destroyed after the last CRef release.

This means that cleaning up thread-specific data can not be done from the thread destructor. One should override OnExit() method instead. OnExit() is guaranteed to be called in the context of the thread before the thread termination. The destructor can be used to cleanup non-thread-local data only.

There is one more possibility to destroy a thread. If a thread has been created, but does not need to be run, one can use Discard() method to destroy the thread object without running it. Again, the object will not be destroyed until there are CRefs referencing it.

Referencing Thread Objects

It should be emphasized that regular (C) pointer to a thread object is not reliable. The thread may terminate at unpredictable moment, destroying itself. There is no possibility to safely access thread object after Join() using C pointers. The only solution to this problem is to use CRef class. CThread class provides a mechanism to prevent premature destruction if there are CRef references to the thread object.

Thread local storage (CTls<> class [*])

The library provides a template class to store thread specific data: CTls<>. This means that each thread can keep its own data in the same TLS object. To perform any kind of cleanup one can provide cleanup function and additional cleanup data when storing a value in the TLS object. The following example demonstrates the usage of TLS:

```
CRef< CTls<int> > tls(new CTls<int>);
void TlsCleanup(int* p_value, void* /* data */ )
{
 delete p_value;
}
...
void* CMyThread::Main()
{
 int* p_value = new int;
 *p_value = 1;
 tls->SetValue(p_value, TlsCleanup);
 ...
 p_value = new int;
 *p_value = 2;
 tls->SetValue(p_value, TlsCleanup);
 ...
 if (*tls->GetValue() == 2) {
 ...
 }
 ...
}
```

In the above example the second call to SetValue() will cause the TlsCleanup() to deallocate the first integer variable. To cleanup the last value stored in each TLS, the CThread::Exit() function will automatically call CTls<>::Reset() for each TLS used by the thread.

By default, all TLS objects are destroyed on program termination, since in most cases it is not guaranteed that a TLS object is not (or will not be) used by a thread. For the same reason the CTls<> destructor is protected, so that no TLS can be created in the stack memory. The best way of keeping TLS objects is to use CRef.

Calling Discard() will schedule the TLS to be destroyed as soon as there are no CRef references to the object left. The method should be used with care.

Mutexes

The ncbimtx.hpp defines platform-independent mutex classes, CMutex, CFastMutex, CMutexGuard, and CFastMutexGuard. These mutex classes are in turn built on the platform-dependent mutex classes SSystemMutex and SSystemFastMutex.

In addition to the mutex classes, there are a number of classes that can be used for explicit locks such as the CRWLock, CAutoRW, CReadLockGuard, CWriteLockGuard and the platform-dependent read/write lock, CInternalRWLock.

Finally, there is the CSemaphore class which is an application-wide semaphore.

These classes are discussed in the subsections that follow:

- CMutex
- CFastMutex
- SSystemMutex and SSystemFastMutex
- CMutexGuard and CFastMutexGuard
- Lock Classes

CMutex

The CMutex class provides the API for acquiring a mutex. This mutex allows nesting with runtime checks so recursive locks by the same thread is possible. This mutex checks the mutex owner before unlocking. CMutex is slower than CFastMutex and should be used when performance is less important than data protection. If performance is more important than data protection, use CFastMutex, instead.

The main methods for CMutex operation are Lock(), TryLock() and Unlock():

```
void Lock(void);
bool TryLock(void);
void Unlock(void);
```

The Lock() mutex method is used by a thread to acquire a lock. The lock can be acquired only if the mutex is unlocked; that is, not in use. If a thread has acquired a lock before, the lock counter is incremented. This is called nesting. The lock counter is only decremented when the same thread issues an Unlock(). In other words, each call to Lock() must have a corresponding Unlock() by the same thread. If the mutex has been locked by another thread, then the thread must wait until it is unlocked. When the mutex is unlocked, the waiting thread can acquire the lock. This, then, is like a lock on an unlocked mutex.

The TryLock() mutex can be used to probe the mutex to see if a lock is possible, and if it is, acquire a lock on the mutex. If the mutex has already been locked, TryLock() returns FALSE. If the mutex is unlocked, than TryLock() acquires a lock on the mutex just as Lock() does, and returns TRUE.

The Unlock() method is used to decrease the lock counter if the mutex has been acquired by this thread. When the lock counter becomes zero, then the mutex is completely released (unlocked). If the mutex is not locked or locked by another thread, then the exception CMutexException (eOwner) is thrown.

The CMutex uses the functionality of CFastMutex. Because CMutex allows nested locks and performs checks of mutex owner it is somewhat slower than CFastMutex, but capable of protecting complicated code, and safer to use. To guarantee for a mutex release, CMutexGuard can be used. The mutex is locked by the CMutexGuard constructor and unlocked by its destructor. Macro DEFINE_STATIC_MUTEX(id) will define static mutex variable with name id. Macro DECLARE_CLASS_STATIC_MUTEX(id) will declare static class member of mutex type name id. Macro DEFINE_CLASS_STATIC_MUTEX(class, id) will define class static mutex variable class::id. The following example demonstrates usage of CMutex, including lock nesting:

```
static int Count = 0;
DEFINE_STATIC_MUTEX(CountMutex);

void Add2(void)
{
 CMutexGuard guard(CountMutex);
 Count += 2;
 if (Count < 20) {
 Add3();
 }
}

void Add3(void)
{
 CMutexGuard guard(CountMutex);
 Count += 3;
 if (Count < 20) {
 Add2();
 }
}
```

This example will result in several nested locks of the same mutex with the guaranteed release of each lock.

It is important not to unlock the mutex protected by a mutex guard. CFastMutexGuard and CMutexGuard both unlock the associated mutex on destruction. It the mutex is already unlocked this will cause a runtime error. Instead of unlocking the mutex directly one can use CFastMutexGuard::Release() or CMutexGuard::Release() method. These methods unlock the mutex and unlink it from the guard.

In addition to usual Lock() and Unlock() methods, the CMutex class implements a method to test the mutex state before locking it. TryLock() method attempts to acquire the mutex for the calling thread and returns true on success (this includes nested locks by the same thread) or false if the mutex has been acquired by another thread. After a successful TryLock() the mutex should be unlocked like after regular Lock().

CFastMutex

The CFastMutex class provides the API for acquiring a mutex. Unlike CMutex, this mutex does not permit nesting and does not check the mutex owner before unlocking. CFastMutex is, however, faster than CMutex and should be used when performance is more important than data protection. If performance is less important than data protection, use CMutex, instead.

The main methods for CFastMutex operation are Lock(), TryLock() and Unlock():

```
void Lock(void);
bool TryLock(void);
void Unlock(void);
```

The Lock() mutex method is used by a thread to acquire a lock without any nesting or ownership checks.

The TryLock() mutex can be used to probe the mutex to see if a lock is possible, and if it is, acquire a lock on the mutex. If the mutex has already been locked, TryLock() returns FALSE. If the mutex is unlocked, than TryLock() acquires a lock on the mutex just as Lock() does, and returns TRUE. The locking is done without any nesting or ownership checks.

The Unlock() method is used to unlock the mutex without any nesting or ownership checks.

The CFastMutex should be used only to protect small and simple parts of code. To guarantee for the mutex release the CFastMutexGuard class may be used. The mutex is locked by the CFastMutexGuard constructor and unlocked by its destructor. To avoid problems with initialization of static objects on different platforms, special macro definitions are intended to be used to declare static mutexes. Macro DEFINE_STATIC_FAST_MUTEX(id) will define static mutex variable with name id. Macro DECLARE_CLASS_STATIC_FAST_MUTEX(id) will declare static class member of mutex type with name id. Macro DEFINE_CLASS_STATIC_FAST_MUTEX(class, id) will define static class mutex variable class::id. The example below demonstrates how to protect an integer variable with the fast mutex:

```
void ThreadSafe(void)
{
 static int Count = 0;
 DEFINE_STATIC_FAST_MUTEX(CountMutex);
 ...
 {{
 CFastMutexGuard guard(CountMutex);
 Count++;
 }}
 ...
}
```

SSystemMutex and SSystemFastMutex

The CMutex class is built on the platform-dependent mutex class, SSystemMutex. The SSystemMutex is in turn built using the SSystemFastMutex class with additional provisions for keeping track of the thread ownership using the CThreadSystemID, and a counter for the number of in the same thread locks (nested or recursive locks).

Each of the SSystemMutex and SSystemFastMutex classes have the Lock(), TryLock() and Unlock() methods that are platform specific. These methods are used by the platform independent classes, CMutex and CFastMutex to provide locking and unlocking services.

CMutexGuard and CFastMutexGuard

The CMutexGuard and the CFastMutexGuard classes provide platform independent read and write lock guards to the mutexes. These classes are aliased as typedefs TReadLockGuard and TWriteLockGuard in the CMutexGuard and the CFastMutexGuard classes.

Lock Classes

This class implements sharing a resource between multiple reading and writing threads. The following rules are used for locking:

- if unlocked, the RWLock can be acquired for either R-lock or W-lock
- if R-locked, the RWLock can be R-locked by the same thread or other threads
- if W-locked, the RWLock can not be acquired by other threads (a call to ReadLock() or WriteLock() by another thread will suspend that thread until the RW-lock release).
- R-lock after W-lock by the same thread is allowed but treated as a nested W-lock
- W-lock after R-lock by the same thread results in a runtime error

Like CMutex, CRWLock also provides methods for checking its current state: TryReadLock () and TryWriteLock(). Both methods try to acquire the RW-lock, returning true on success (the RW-lock becomes R-locked or W-locked) or false if the RW-lock can not be acquired for the calling thread.

The following subsections describe these locks in more detail:

- CRWLock
- CAutoRW
- CReadLockGuard
- CWriteLockGuard
- CInternalRWLock
- CSemaphore

CRWLock

The CRWLock class allows read-after-write (R-after-W) locks for multiple readers or a single writer with recursive locks. The R-after-W lock is considered to be a recursive Write-lock. The write-after-read (W-after-R) is not permitted and can be caught when _DEBUG is defined. When _DEBUG is not defined, it does not always detect the W-after-R correctly, so a deadlock can occur in these circumstances. Therefore, it is important to test your application in the _DEBUG mode first.

The main methods in the class API are ReadLock(), WriteLock(), TryReadLock(), TryWriteLock() and Unlock().

```
void ReadLock(void);
void WriteLock(void);
bool TryReadLock(void);
bool TryWriteLock(void);
void Unlock(void);
```

The ReadLock() is used to acquire a read lock. If a write lock has already been acquired by another thread, then this thread waits until it is released.

The WriteLock() is used to acquire a write lock. If a read or write lock has already been acquired by another thread, then this thread waits until it is released.

The TryReadLock() and TryWriteLock() methods are used to try and acquire a read or write lock, respectively, if at all possible. If a lock cannot be acquired, they immediately return with a FALSE value and do not wait to acquire a lock like the ReadLock() and WriteLock() methods. If a lock is successfully acquired, a TRUE value is returned.

As expected from the name, the Unlock() method releases the RW-lock.

CAutoRW

The CAutoRW class is used to provide a Read Write lock that is automatically released by the CAutoRW class' destructor. The locking mechanism is provided by a CRWLock object that is initialized when the CAutoRW class constructor is called.

An acquired lock can be released by an explicit call to the class Release() method. The lock can also be released by the class destructor. When the destructor is called the lock if successfully acquired and not already released by Release() is released.

CReadLockGuard

The CReadLockGuard class is used to provide a basic read lock guard that can be used by other classes. This class is derived from the CAutoRW class.

The class constructor can be passed a CRWLock object on which a read lock is acquired, and which is registered to be released by the class destructor. The class's Guard() method can also be called with a CRWLock object and if this is not the same as the already registered CRWLock object, the old registered object is released, and the new CRWLock object is registered and a read lock acquired on it.

CWriteLockGuard

The CWriteLockGuard class is used to provide a basic write lock guard that can be used by other classes. The CWriteLockGuard class is similar to the CReadLockGuard class except that it provides a write lock instead of a read lock. This class is derived from the CAutoRW class.

The class constructor can be passed a CRWLock object on which a write lock is acquired, and which is registered to be released by the class destructor. The class's Guard() method can also be called with a CRWLock object and if this is not the same as the already registered CRWLock object, the old registered object is released, and the new CRWLock object is registered and a write lock acquired on it.

CInternalRWLock

The CInternalRWLock class holds platform dependent RW-lock data such as data on semaphores and mutexes. This class is not meant to be used directly by user applications. This class is used by other classes such as the CRWLock class.

CSemaphore

The CSemaphore class implements a general purpose counting semaphore. The constructor is passed an initial count for the semaphore and a maximum semaphore count.

When the Wait() method is executed for the semaphore, the counter is decremented by one. If the semaphore's count is zero then the thread waits until it is not zero. A variation on the Wait () method is the TryWait() method which is used to prevent long waits. The TryWait() can be passed a timeout value in seconds and nanoseconds:

```
bool TryWait(unsigned int timeout_sec = 0, unsigned int timeout_nsec = 0);
```

The TryWait() method can wait for the specified time for the semaphore's count to exceed zero. If that happens, the counter is decremented by one and TryWait() returns TRUE; otherwise, it returns FALSE.

The semaphore count is incremented by the Post() method and an exception is thrown if the maximum count is exceeded.

Working with File and Directories Using CFile and CDir

An application may need to work with files and directories. The CORELIB provides a number of portable classes to model a system file and directory. The base class for the files and directories is CDirEntry. Other classes such as CDir and CFile that deal with directories and files are derived form this base class.

The following sections discuss the file and directory classes in more detail:

- Executing a System Command using the System() Method
- Defining Spawned Process Modes (EMode type)
- Spawning a Process using SpawnX() Methods
- Waiting for a Process to Terminate using the Wait() method

CDirEntry class

This class models the directory entry structure for the file system and assumes that the path argument has the following form, where any or all components may be missing:

```
<dir><title><ext>
```

where:

- <dir> -- is the file path ("/usr/local/bin/" or "c:\windows\")
- <title> -- is the file name without ext ("autoexec")
- <ext> -- is the file extension (".bat" - whatever goes after the last dot)

The supported filename formats are for the Windows, Unix, and Mac file systems.

The CDirEntry class provides the base methods such as the following for dealing with the components of a path name :

- GetPath(): Get pathname.
- GetDir(): Get the Directory component for this directory entry.
- GetBase(): Get the base entry name without extension.
- GetName(): Get the base entry name with extension.
- GetExt(): Get the extension name.
- MakePath(): Given the components of a path, combine them to create a path string.
- SplitPath(): Given a path string, split them into its constituent components.

- GetPathSeparator(): Get path separator symbol specific for the platform such as a '\' or '/'.
- IsPathSeparator(): Check character "c" as path separator symbol specific for the platform.
- AddTrailingPathSeparator(): Add a trailing path separator, if needed.
- ConvertToOSPath(): Convert relative "path" on any OS to current OS dependent relative path.
- IsAbsolutePath(): Note that the "path" must be for current OS.
- ConcatPath(): Concatenate the two parts of the path for the current OS.
- ConcatPathEx(): Concatenate the two parts of the path for any OS.
- MatchesMask(): Match "name" against the filename "mask".
- Rename(): Rename entry to specified "new_path".
- Remove(): Remove the directory entry.

The last method on the list, the Remove() method accepts an enumeration type parameter, EDirRemoveMode, which specifies the extent of the directory removal operation - you can delete only an empty directory, only files in a directory but not any subdirectories, or remove the entire directory tree:

```
/// Directory remove mode.
enum EDirRemoveMode {
 eOnlyEmpty, ///< Remove only empty directory
 eNonRecursive, ///< Remove all files in directory, but not remove
 ///< subdirectories and files in it
 eRecursive ///< Remove all files and subdirectories
};
```

CDirEntry knows about different types of files or directory entries. Most of these file types are modeled after the Unix file system but can also handle the file system types for the Windows platform. The different file system types are represented by the enumeration type EType which is defined as follows :

```
/// Which directory entry type.
enum EType {
 eFile = 0, ///< Regular file
 eDir, ///< Directory
 ePipe, ///< Pipe
 eLink, ///< Symbolic link (Unix only)
 eSocket, ///< Socket (Unix only)
 eDoor, ///< Door (Unix only)
 eBlockSpecial, ///< Block special (Unix only)
 eCharSpecial, ///< Character special
 //
 eUnknown ///< Unknown type
};
```

CDirEntry knows about permission settings for a directory entry. Again, these are modeled after the Unix file system. The different permissions are represented by the enumeration type EMode which is defined as follows :

```
/// Directory entry's access permissions.
enum EMode {
 fExecute = 1, ///< Execute permission
 fWrite = 2, ///< Write permission
 fRead = 4, ///< Read permission
 // initial defaults for dirs
 fDefaultDirUser = fRead | fExecute | fWrite,
 ///< Default user permission for dir.
 fDefaultDirGroup = fRead | fExecute,
 ///< Default group permission for dir.
 fDefaultDirOther = fRead | fExecute,
 ///< Default other permission for dir.
 // initial defaults for non-dir entries (files, etc.)
 fDefaultUser = fRead | fWrite,
 ///< Default user permission for file
 fDefaultGroup = fRead,
 ///< Default group permission for file
 fDefaultOther = fRead,
 ///< Default other permission for file
 fDefault = 8 ///< Special flag: ignore all other flags,
 ///< use current default mode
};
typedef unsigned int TMode; ///< Binary OR of "EMode"
```

The directory entry permissions of read(r), write(w), execute(x), are defined for the "user", "group" and "others" The initial default permission for directories is "rwx" for "user", "rx" for "group" and "rx" for "others". These defaults allow a user to create directory entries while the "group" and "others" can only change to the directory and read a listing of the directory contents. The initial default permission for files is "rw" for "user", "r" for "group" and "r" for "others". These defaults allow a user to read and write to a file while the "group" and "others" can only read the file.

These directory permissions handle most situations but don't handle all permission types. For example, there is currently no provision for handling the Unix "sticky bit" or the "suid" or "sgid" bits. Moreover, operating systems such as Windows NT/2000/2003 and Solaris use Access Control Lists (ACL) settings for files. There is no provision in CDirEntry to handle ACLs

Other methods in CDirEntry deal specifically with checking the attributes of a directory entry such as the following methods:

- IsFile(): Check if directory entry is a file.
- IsDir(): Check if directory entry is a directory.
- GetType(): Get type of directory entry. This returns an EType value.
- GetTime(): Get time stamp of directory entry.
- GetMode(): Get permission mode for the directory entry.
- SetMode(): Set permission mode for the directory entry.
- static void SetDefaultModeGlobal(): Set default mode globally for all CDirEntry objects. This is a class-wide static method and applies to all objects of this class.
- SetDefaultMode(): Set mode for this one object only.

These methods are inherited by the derived classes CDir and CFile that are used to access directories and files, respectively.

CFile class

The CFile is derived from the base class, CDirEntry. Besides inheriting the methods discussed in the previous section, the following new methods specific to files are defined in the CFile class:

- Exists(): Check existence for a file.
- GetLength(): Get size of file.
- GetTmpName(): Get temporary file name.
- GetTmpNameEx(): Get temporary file name in a specific directory and having a specified prefix value.
- CreateTmpFile(): Create temporary file and return pointer to corresponding stream.
- CreateTmpFileEx(): Create temporary file and return pointer to corresponding stream. You can additionally specify the directory in which to create the temporary file and the prefix to use for the temporary file name.

The methods CreateTmpFile() and CreateTmpFileEx() allow the creation of either a text or binary file. These two types of files are defined by the enumeration type, ETextBinary, and the methods accept a parameter of this type to indicate the type of file to be created:

```
/// What type of temporary file to create.
enum ETextBinary {
 eText, ///<Create text file
 eBinary ///< Create binary file
};
```

Additionally, you can specify the type of operations (read, write) that should be permitted on the temporary files. These are defined by the enumeration type, EAllowRead, and the CreateTmpFile() and CreateTmpFileEx() methods accept a parameter of this type to indicate the type operations that are permitted:

```
/// Which operations to allow on temporary file.
enum EAllowRead {
 eAllowRead, ///< Allow read and write
 eWriteOnly ///< Allow write only
};
```

CDir class

The CDir is derived from the base class, CDirEntry. Besides inheriting the methods discussed in the CDirEntry section, the following new methods specific to directories are defined in the CDir class:

- Exists(): Check existence for a directory.
- GetHome(): Get the user's home directory.
- GetCwd(): Get the current working directory.
- GetEntries(): Get directory entries based on a specified mask parameter. Retuns a vector of pointers to CDirEntry objects defined by TEntries
- Create(): Create the directory using the directory name passed in the constructor.

- CreatePath(): Create the directory path recursively possibly more than one at a time.
- Remove(): Delete existing directory.

The last method on the list, the Remove() method accepts an enumeration type parameter, EDirRemoveMode, defined in the CDirEntry class which specifies the extent of the directory removal operation - you can delete only an empty directory, only files in a directory but not any subdirectories, or remove the entire directory tree.

CMemoryFile class

The CMemoryFile is derived from the base class, CDirEntry. This class creates a virtual image of a disk file in memory that allow normal file operations to be permitted, but the file operations are actually performed on the image of the file in memory. This can result in considerable improvements in speed when there are many "disk intensive" file operations being performed on a file which is mapped to memory.

Besides inheriting the methods discussed in the CDirEntry section, the following new methods specific to memory mapped are defined in the CMemoryFile class:

- IsSupported(): Check if memory-mapping is supported by the C++ Toolkit on this platform.
- GetPtr(): Get pointer to beginning of data in the memory mapped file.
- GetSize(): Get size of the mapped area.
- Flush(): Flush by writing all modified copies of memory pages to the underlying file.
- Unmap(): Unmap file if it has already been mapped.
- MemMapAdvise(): Advise on memory map usage.
- MemMapAdviseAddr(): Advise on memory map usage for specified region.

The methods MemMapAdvise() and MemMapAdviseAddr() allow one to advise on the expected usage pattern for the memory mapped file. The expected usage pattern is defined by the enumeration type, EMemMapAdvise, and these methods accept a parameter of this type to indicate the usage pattern:

```
/// What type of data access pattern will be used for mapped region.
///
/// Advises the VM system that the a certain region of user mapped memory
/// will be accessed following a type of pattern. The VM system uses this
/// information to optimize work with mapped memory.
///
/// NOTE: Now works on Unix platform only.
typedef enum {
 eMMA_Normal, ///< No further special treatment
 eMMA_Random, ///< Expect random page references
 eMMA_Sequential, ///< Expect sequential page references
 eMMA_WillNeed, ///< Will need these pages
 eMMA_DontNeed ///< Don't need these pages
} EMemMapAdvise;
```

The memory usage advice is implemented on Unix platforms only, and is not supported on Windows platforms.

String APIs

The ncbistr.hpp file defines a number of useful constants, types and functions for handling string types. Most of the string functions are defined as class-wides static members of the class NStr.

The following sections provide additional details on string APIs

- String Constants
- NStr Class
- UTF-8 Strings
- PCase and PNocase

String Constants

For convenience, two types of empty strings are provided. A C-language style string that terminates with the null character ('\0') and a C++ style empty string.

The C-language style empty string constants are NcbiEmptyCStr which is a macro definition for the NCBI_NS_NCBI::kEmptyCStr. So the NcbiEmptyStr and kEmptyCStr are, for all practical purposes, equivalent.

The C++-language style empty string constants are NcbiEmptyString and the kEmptyStr which are macro definitions for the NCBI_NS_NCBI::CNcbiEmptyString::Get() method that returns an empty string. So the NcbiEmptyString and kEmptyStr are, for all practical purposes, equivalent.

The SIZE_TYPE is an alias for the string::size_type, and the NPOS defines a constant that is returned when a substring search fails, or to indicate an unspecified string position.

NStr Class

The NStr class encapsulates a number of class-wide static methods. These include string concatenation, string conversion, string comparison, string search functions. Most of these string operations should be familiar to developers by name. For details, see the NStr static methods documentation.

UTF-8 Strings

The CStringUTF8 class extends the C++ string class and provides support for Unicode Transformation Format-8 (UTF-8) strings.

This class supports constructors where the input argument is a string reference, char* pointer, and wide string, and wide character pointers. Wide string support exists if the macro HAVE_WSTRING is defined:

```
CStringUTF8(const string& src);
CStringUTF8(const char* src);
CStringUTF8(const wstring& src);
CStringUTF8(const wchar_t* src);
```

The CStringUTF8 class defines assignment(=) and append-to string (+=) operators where the string assigned or appended can be a CStringUTF8 reference, string reference, char* pointer, wstring reference, wchar_t* pointer.

Conversion to ASCII from CStringUTF8 is defined by the AsAscii() method. This method can throw a StringException with error codes 'eFormat' or 'eConvert' if the string has a wrong UTF-8 format or cannot be converted to ASCII.

```
string AsAscii(void) const;
wstring AsUnicode(void) const
```

PCase and PNocase

The PCase and PNocase structures define case-sensitive and case-insensitive comparison functions, respectively. These comparison functions are the Compare(), Less(), Equals(), operator(). The Compare() returns an integer (-1 for less than, 0 for equal to, 1 for greater than). The Less() and Equals() return a TRUE if the first string is less than or equal to the second string. The operator() returns TRUE if the first string is less than the second.

A convenience template function AStrEquiv is defined that accepts the two classes to be compared as template parameters and a third template parameter that can be the comparison class such as the PCase and PNocase defined above.

Portable Time Class

The ncbitime.hpp defines CTime, the standard Date/Time class that also can be used to represent elapsed time. Please note that the CTime class works for dates after 1/1/1900 and should not be used for elapsed time prior to this date. Also, since Mac OS 9 does not support the daylight savings flag, CTime does not support daylight savings on this platform.

The subsections that follow discuss the following topics:

- CTime Class Constructors
- Other CTime Methods

CTime Class Constructors

The CTime class defines three basic constructors that accept commonly used time description arguments and some explicit conversion and copy constructors. The basic constructors are the following:

- Constructor 1:
  CTime(EInitMode mode = eEmpty,
  ETimeZone tz = eLocal,
  ETimeZonePrecision tzp = eTZPrecisionDefault);
- Constructor 2:
  CTime(int year,
  int month,
  int day,
  int hour = 0,
  int minute = 0,
  int second = 0,
  long nanosecond = 0,
  ETimeZone tz = Local,
  ETimeZonePrecision tzp = eTZPrecisionDefault);
- Constructor 3:
  CTime(int year,
  int yearDayNumber,

ETimeZone tz = eLocal,
ETimeZonePrecision tzp = eTZPrecisionDefault);

In Constructor 1, the EInitMode is an enumeration type defined in the CTime class that can be used to specify whether to build the time object with empty time value (eEmpty) or current time (eCurrent). The ETimeZone is an enumeration type also defined in the CTime class that is used to specify the local time zone (eLocal) or GMT (eGmt. The ETimeZonePrecision is an enumeration type also defined in the CTime class that can be used to specify the time zone precision to be used for adjusting the daylight savings time. The default value is eNone, which means that daylight savings do not affect time calculations.

Constructor 2 differs from Constructor 1 with respect to how the timestamp is specified. Here the time stamp is explictly specified as the year, month, day, hour, minute, second, and nanosecond values. The other parameters of type ETimeZone and ETimeZonePrecision have the meanings discussed in the previous paragraph.

Constructor 3 allows the timestamp to be constructed as the Nth day (yearDayNumber) of a year(year). The other parameters of type EtimeZone and ETimeZonePrecision have the meanings discussed in the previous paragraph.

The explicit conversion constructor allows the conversion to be made from a string representation of time. The default value of the format string is kEmptyStr, which implies that the format string has the format "M/D/Y h:m:s". As one would expect, the format specifiers M, D, Y, h, m, and s have the meanings month, day, year, hour, minute, and second, respectively:

```
explicit CTime(const string& str,
 const string& fmt = kEmptyStr,
 ETimeZone tz = eLocal,
 ETimeZonePrecision tzp = eTZPrecisionDefault);
```

There is also a copy constructor defined that permits copy operations for CTime objects.

Other CTime Methods

Once the CTime object is constructed, it can be accessed using the SetTimeT() and GetTimeT() methods. The SetTimeT() method is used to set the CTime with the timestamp passed by the time_t parameter. The GetTimeT() method returns the time stored in the CTime object as a time_t value. The time_t value measures seconds since January 1, 1900; therefore, do not use these methods if the timestamp is less than 1900. Also, time formats are in GMT time format.

A series of methods that set the time using the database formats TDBTimeI and TDBTimeU are also defined. These database time formats contain local time only and are defined as typedefs in ncbitime.hpp. The mutator methods are SetTimeDBI() and SetTimeDBU(), and the accessor methods are GetTimeDBI() and GetTimeDBU().

You can set the time to the current time using the SetCurrent() method, or set it to "empty" using the Clear() method. If you want to measure time as days only and strip the hour, minute, and second information, you can use Truncate() method.

You can get or set the current time format using the GetFormat() and SetFormat() methods.

You can get and set the individual components of time, such as year, day, month, hour, minute, second, and nanosecond. The accessor methods for these components are named after the component itself, and their meanings are obvious, e.g., Year() for getting the year component, Month() for getting the month component, Day() for getting the day component, Hour() for getting the hour component, Minute() for getting the minute component, Second() for getting the second component, and NanoSecond() for getting the nanosecond component. The corresponding mutator methods for setting the individual components are the same as the accessor, except that they have the prefix "Set" before them. For example, the mutator method for setting the day is SetDay(). A word of caution on setting the individual components: You can easily set the timestamp to invalid values, such as changing the number of days in the month of February to 29 when it is not a leap year, or 30 or 31.

A number of methods are available to get useful information from a CTime object. To get a day's year number (1 to 366) use YearDayNumber(). To get the week number in a year, use YearWeekNumber(). To get the week number in a month, use MonthWeekNumber(). You can get the day of week (Sunday=0) by using DayOfWeek(), or the number of days in the current month by using DaysInMonth().

There are times when you need to add months, days, hours, minutes, or seconds to an existing CTime object. You can do this by using the AddXXX() methods, where the "XXX" is the time component such as "Year", "Month", "Day", "Hour", "Minute", "Second", "NanoSecond" that is to be added to. Be aware that because the number of days in a month can vary, adding months may change the day number in the timestamp. Operator methods for adding to (+=), subtracting from (-=), incrementing (++), and decrementing (--) days are also available.

If you need to compare two timestamps, you can use the operator methods for equality (==), in-equality (!=), earlier than (<), later than (>), or a combination test, such as earlier than or equal to (<=) or later than or equal to (>=).

You can measure the difference between two timestamps in days, hours, minutes, seconds, or nanoseconds. The timestamp difference methods have the form DiffXXX(), where "XXX" is the time unit in which you want the difference calculated such as "Day", "Hour", "Minute", "Second", or "NanoSecond". Thus, DiffHour() can be used to calculate the difference in hours.

There are times when you may need to do a check on the timestamp. You can use IsLeap() to check if the time is in a leap year, or if it is empty by using IsEmpty(), or if it is valid by using IsValid(), or if it is local time by using IsLocalTime(), or if it is GMT time by using IsGmtTime ().

If you need to work with time zones explicitly, you can use GetTimeZoneFormat() to get the current time zone format, and SetTimeZoneFormat() to change it. You can use GetTimeZonePrecision() to get the current time zone precision and SetTimeZonePrecision() to change it. To get the time zone difference between local time and GMT, use TimeZoneOffset (). To get current time as local time use GetLocalTime(), and as GMT time use GetGmtTime (). To convert current time to a specified time zone, use ToTime(), or to convert to local time use ToLocalTime().

Also defined for CTime are assignment operators to assign a CTime object to another CTime and an assignment operator where the right hand side is a time value string.

Template Utilities

The ncbiutil.hpp file defines a number of useful template functions, classes, and struct definitions that are used in other parts of the library.

The following topics are discussed in this section:

- Function Objects

- Template Functions

Function Objects

The p_equal_to and pair_equal_to are template function classes that are derived from the standard binary_function base class. The p_equal_to checks for equality of objects pointed to by a pointer and pair_equal_to checks whether a pair's second element matches a given value. Another PPtrLess function class allows comparison of objects pointed to by a smart pointer.

The CNameGetter template defines the function GetKey(), which returns the name attribute for the template parameter.

Template Functions

Defined here are a number of inline template functions that make it easier to perform common operations on map objects.

NotNull() checks for a null pointer value and throws a CCoreException, if a null value is detected. If the pointer value is not null, it is simply returned.

GetMapElement() searches a map object for an element and returns the element, if found. If the element is not found, it returns a default value, which is usually set to 0 (null).

SetMapElement() sets the map element. If the element to be set is null, its existing key is erased.

InsertMapElement() inserts a new map element.

GetMapString() and SetMapString() are similar to the more general GetMapElement() and SetMapElement(), except that they search a map object for a string. In the case of GetMapString (), it returns a string, if found, and an empty string ("") if not found.

There are three overloads for the DeleteElements() template function. One overload accepts a container (list, vector, set, multiset) of pointers and deletes all elements in the container and clears the container afterwards. The other overloads work with map and multimap objects. In each case, they delete the pointers in the map object and clear the map container afterwards.

The AutoMap() template function works with a cache pointed to auto_ptr. It retrieves the result from the cache, and if the cache is empty, it inserts a value into the cache from a specified source.

A FindBestChoice() template function is defined that returns the best choice (lowest score) value in the container. The container and scoring functions are specified as template parameters. The FindBestChoice() in turn uses the CBestChoiceTracker template class, which uses the standard unary_function as its base class. The CBestChoiceTracker contains the logic to record the scoring function and keep track of the current value and the best score.

Miscellaneous Types and Macros

The ncbimisc.hpp file defines a number of useful enumeration types and macros that are used in other parts of the library.

The following topics are discussed in this section:

- Miscellaneous Enumeration Types
- AutoPtr Class
- ITERATE Macros

- Sequence Position Types

Miscellaneous Enumeration Types

The enum type EOwnership defines the constants eNoOwnership and eTakeOwnership. These are used to specify relationships between objects.

The enum type ENullable defines the constants eNullable and eNotNullable. These are used to specify if a data element can hold a null or not-null value.

AutoPtr Class

The ncbimisc.hpp file defines an auto_ptr class if the HAVE_NO_AUTO_PTR macro is undefined. This is useful in replacing the std::auto_ptr of STL for compilers with poor "auto_ptr" implementation. Section STL auto_ptrs discusses details on the use of auto_ptr.

Another class related to the auto_ptr class is the AutoPtr class. The Standard auto_ptr class from STL does not allow the auto_ptr to be put in STL containers such as list, vector, map etc. Because of the nature of how ownership works in an auto_ptr class, the copy constructor and assignment operator of AutoPtr modify the state of the source AutoPtr object as it transfers ownership to the target AutoPtr object.

A certain amount of flexibility has been provided in terms of how the pointer is to be deleted. This is done by passing a second argument to the AutoPtr template. This second argument allows the passing of a functor object that defines the deletion of the object. You can define "malloc" pointers in AutoPtr, or you can use an ArrayDeleter template class to properly delete an array of objects using "delete[]". By default, the internal pointer will be deleted using the "delete" operator.

ITERATE macros

When working with STL (or STL-like) container classes, it is common to use a for-statement to iterate through the elements in a container, for example:

```
for (Type::const_iterator it = cont.begin(); it != cont.end(); ++it)
```

However, there are a number of ways that iterating in this way can fail. For example, suppose the function GetNames() returns a vector of strings by value and is used like this:

```
for (vector<string>::iterator it = GetNames().begin(); it != GetNames().end
(); ++it)
```

This code has the serious problem that the termination condition will never be met because every time GetNames() is called a new object is created, and therefore neither the initial iterator returned by begin() nor the iterator returned by operator++() will ever match the iterator returned by end(). Code like this is not common but does occasionally get written, resulting in a bug and wasted time.

A simpler criticism of the for-statement approach is that the call to end() is repeated unnecessarily.

Therefore, to make it easier to write code that will correctly and efficiently iterate through the elements in STL and STL-like containers, the ITERATE and NON_CONST_ITERATE macros were defined. Using ITERATE , the for-statement at the start of this section becomes simply:

```
ITERATE(Type, it, cont)
```

Note: The container argument must be an lvalue and may be evaluated more than once, so it must always evaluate to the same container instance.

ITERATE uses a constant iterator; NON_CONST_ITERATE uses a non-constant iterator.

The ITERATE and NON_CONST_ITERATE macros are defined in include/corelib/ncbimisc.hpp, along with related macros including NON_CONST_SET_ITERATE, ERASE_ITERATE, VECTOR_ERASE, REVERSE_ITERATE, ITERATE_SIMPLE, and more.

Sequence Position Types

The TSeqPos and and TSignedSeqPos are defined to specify sequence locations and length. TSeqPos is defined as an unsigned int, and TSignedSqPos is a signed int that should be used only when negative values are a possibility for reporting differences between positions, or for error reporting, although exceptions are generally better for error reporting.

Containers

The Container classes are template classes that provide many useful container types. The template parameter refers to the types of objects whose collection is being described. An overview of some of the container classes is presented in the introductory chapter on the C++ Toolkit.

The following classes are described in this section:

- template<typename Coordinate> class CRange
- template<typename Object, typename Coordinate = int> class CRangeMap
- template<typename Object, typename Coordinate = int> class CRangeMultiMap
- class CIntervalTree

template<typename Coordinate> class CRange

Class for storing information about some interval (from:to). From and to points are inclusive.

Typedefs

```
position_type
```

synonym of Coordinate.

Methods

```
CRange();
CRange(position_type from, position_type to);
```

constructors

```
static position_type GetEmptyFrom();
static position_type GetEmptyTo();
static position_type GetWholeFrom();
static position_type GetWholeTo();
```

get special coordinate values

```
static CRange<position_type> GetEmpty();
static CRange<position_type> GetWhole();
```

get special interval objects

```
bool HaveEmptyBound() const;
```

check if any bound have special 'empty' value

```
bool HaveInfiniteBound() const;
```

check if any bound have special 'whole' value

```
bool Empty() const;
```

check if interval is empty (any bound have special 'empty' value or left bound greater then right bound)

```
bool Regular() const;
```

check if interval's bounds are not special and length is positive

```
position_type GetFrom() const;
position_type GetTo() const;
position_type GetLength() const;
```

get parameters of interval

```
CRange<position_type>& SetFrom();
CRange<position_type>& SetTo();
```

set bounds of interval

```
CRange<position_type>& SetLength();
```

set length of interval leaving left bound (from) unchanged

```
CRange<position_type>& SetLengthDown();
```

set length of interval leaving right bound (to) unchanged

```
bool IntersectingWith(CRange<position_type> range) const;
```

check if non empty intervals intersect

```
bool IntersectingWithPossiblyEmpty(CRange<position_type> range) const;
```

check if intervals intersect

template<typename Object, typename Coordinate = int> class CRangeMap

Class for storing and retrieving data using interval as key. Also allows efficient iteration over intervals intersecting with specified interval. Time of iteration is proportional to amount of intervals produced by iterator. In some cases, algorithm is not so efficient and may slowdown.

template<typename Object, typename Coordinate = int> class CRangeMultiMap

Almost the same as CRangeMap but allows several values have the same key interval.

class CIntervalTree

Class with the same functionality as CRangeMap although with different algorithm. It is faster and its speed is not affected by type of data but it uses more memory (triple as CRangeMap) and, as a result, less efficient when amount of interval in set is quite big. It uses about 140 bytes per interval for 64 bit program so you can calculate if CIntervalTree is acceptable. For example, it becomes less efficient than CRangeMap when total memory becomes greater than processor cache.

Thread Pools

CThreadPool is the main class that implements a pool of threads. It executes any tasks derived from the CThreadPool_Task class. The number of threads in pool is controlled by special holder of this policy: object derived from CThreadPool_Controller (default implementation is CThreadPool_Controller_PID based on Proportional-Integral-Derivative algorithm). All threads executing by CThreadPool are the instances of CThreadPool_Thread class or its derivatives.

The following classes are discussed in this section:

- CThreadPool
- CThreadPool_Task
- CThreadPool_Thread
- CThreadPool_Controller
- CThreadPool_Controller_PID

Class CThreadPool

Main class implementing functionality of pool of threads. CThreadPool can be created in 2 ways:

- with minimum and maximum limits on count of simultaneously working threads and default object controlling the number of threads in pool during CThreadPool lifecycle (instance of CThreadPool_Controller_PID);
- with custom object controlling the number of threads (instance of class derived from CThreadPool_Controller). This object will control among all other the minimum and maximum limits on count of simultaneously working threads.

Both constructors take additional parameter - maximum number of tasks waiting in the inner CThreadPool's queue for their execution. When this limit will be reached next call to AddTask () will block until some task from queue is executed and there is free room for new task.

CThreadPool has the ability to execute among ordinary tasks some exclusive ones. After call to RequestExclusiveExecution() all threads in pool will suspend their work (finishing currently executing tasks) and exclusive task will be executed in the special exclusive thread.

If there's necessity to implement some special per-thread logic in CThreadPool then class can be derived to override virtual method CreateThread() in which some custom object derived from CThreadPool_Thread can be created.

Class CThreadPool_Task

Abstract class derived from CObject, encapsulating task for execution in a CThreadPool. The pure virtual method EStatus Execute(void) is called when some thread in pool becomes free and ready to execute this task. The lifetime of the task is controlled inside pool by CRef<> classes.

Class CThreadPool_Thread

Base class for a thread running inside CThreadPool and executing its tasks. Class can be derived to implement some per-thread functionality in CThreadPool. For this purpose there are protected virtual methods Initialize() and Finalize() which are called at the start and finish of the thread correspondingly. And there are methods GetPool() and GetCurrentTask() for application needs.

Class CThreadPool_Controller

Abstract base class for implementations of policies of threads creation and deletion inside pool.

Class CThreadPool_Controller_PID

Default object controlling number of threads working in the pool. Implementation is based on Proportional-Integral-Derivative algorithm for keeping in memory just threads that are necessary for efficient work.

Miscellaneous Classes

The following classes are discussed in this section. For an overview of these classes see the Lightweight Strings and the Checksum sections in the introductory chapter on the C++ Toolkit.

- class CTempString
- class CChecksum

class CTempString

Class CTempString implements a light-weight string on top of a storage buffer whose lifetime management is known and controlled.

CTempString is designed to avoid memory allocation but provide a string interaction interface congruent with std::basic_string<char>.

As such, CTempString provides a const-only access interface to its underlying storage. Care has been taken to avoid allocations and other expensive operations wherever possible.

CTempString has constructors from std::string and C-style string, which do not copy the string data but keep char pointer and string length.This way the construction and destruction are very efficient.

Take into account, that the character string array kept by CTempString object must remain valid and unchanged during whole lifetime of the CTempString object.

It's convenient to use the class CTempString as an argument of API functions so that no allocation or deallocation will take place on of the function call.

class CChecksum

Class for CRC32 checksum calculation. It also has methods for adding and checking checkum line in text files.

Input/Output Utility Classes

This section provides reference information on a number of Input/Output Utility classes. For an overview of these classes see the Stream Support section in the introductory chapter on the C++ Toolkit.

- class CIStreamBuffer
- class COStreamBuffer
- class CByteSource
- class CStreamByteSource
- class CFStreamByteSource
- class CFileByteSource
- class CMemoryByteSource
- class CByteSourceReader
- class CSubSourceCollector

class CIStreamBuffer

Class for additional buffering of standard C++ input streams (sometimes standard C++ iostreams performance quite bad). Uses CByteSource as a data source.

class COStreamBuffer

Class for additional buffering of standard C++ output streams (sometimes standard C++ iostreams performance quite bad).

class CByteSource

Abstract class for abstract source of byte data (file, stream, memory etc).

class CStreamByteSource

CByteSource subclass for reading from C++ istream.

class CFStreamByteSource

CByteSource subclass for reading from C++ ifstream.

class CFileByteSource

CByteSource subclass for reading from named file.

class CMemoryByteSource

CByteSource subclass for reading from memory buffer.

class CByteSourceReader

Abstract class for reading data from CByteSource.

class CSubSourceCollector

Abstract class for obtaining piece of CByteSource as separate source.

Using the C++ Toolkit from a Third Party Application Framework

The NCBI C++ Toolkit includes an API, via corelib/ncbi_toolkit.hpp, that provides an easy way to initialize the NCBI C++ Toolkit internals to use the Toolkit from other application frameworks. This is particularly helpful when those frameworks provide their own logging.

To initialize the NCBI C++ Toolkit internal infrastructure use the function:

```
void NcbiToolkit_Init
 (int argc,
 const TNcbiToolkit_XChar* const* argv,
 const TNcbiToolkit_XChar* const* envp = NULL,
 INcbiToolkit_LogHandler* log_handler = NULL);
```

where the parameter meanings are:

Parameter	Meaning
argc	Argument count [argc in a regular main(argc, argv)].
argv	Argument vector [argv in a regular main(argc, argv)].
envp	Environment pointer [envp in a regular main(argc, argv, envp)]; a null pointer (the default) corresponds to the standard system array (environ on most Unix platforms).
log_handler	Handler for diagnostic messages that are emitted by the C++ Toolkit code.

Note: The TNcbiToolkit_XChar parameter type is used for compatibility with applications that use Unicode under Windows.

When your application is finished using the NCBI C++ Toolkit, be sure to release the Toolkit resources by calling:

```
void NcbiToolkit_Fini(void);
```

The following program illustrates how to forward the NCBI C++ Toolkit logging to another application framework:

```
#include <ncbi_pch.hpp>
#include <iostream>
#include <corelib/ncbi_toolkit.hpp>
#include <corelib/ncbifile.hpp>

using namespace std;
using namespace ncbi;

class MyLogHandler : public INcbiToolkit_LogHandler
{
public:
 void Post(const CNcbiToolkit_LogMessage& msg)
 {
```

```
 // This is where you could pass log messages generated by the
 // NCBI C++ Toolkit to another application framework, e.g.:
 // some_framework::ERR_POST(msg.Message());
 // In this demo, I'll just print out the message.
 cout << "Log message from C++ Toolkit:\n" << msg.Message() << endl;
 }
};

int main(int argc,
 const TNcbiToolkit_XChar* const* argv,
 const TNcbiToolkit_XChar* const* envp)
{
 // Initialize the NCBI C++ Toolkit application framework.
 MyLogHandler log_handler;
 NcbiToolkit_Init(argc,argv,envp,&log_handler);

 // Use a part of the NCBI C++ Toolkit that will cause a log message.
 // This will cause MyLogHandler::Post() to get called, where the log
 // message can get passed to the third party application framework.
 CFileAPI::SetLogging(eOn);
 CDirEntry baddir(CDirEntry("<bad>"));
 baddir.Stat(0);

 // Release resources used by the NCBI C++ Toolkit application framework.
 NcbiToolkit_Fini();

 return 0;
}
```

Note: This API is in the ncbi namespace.

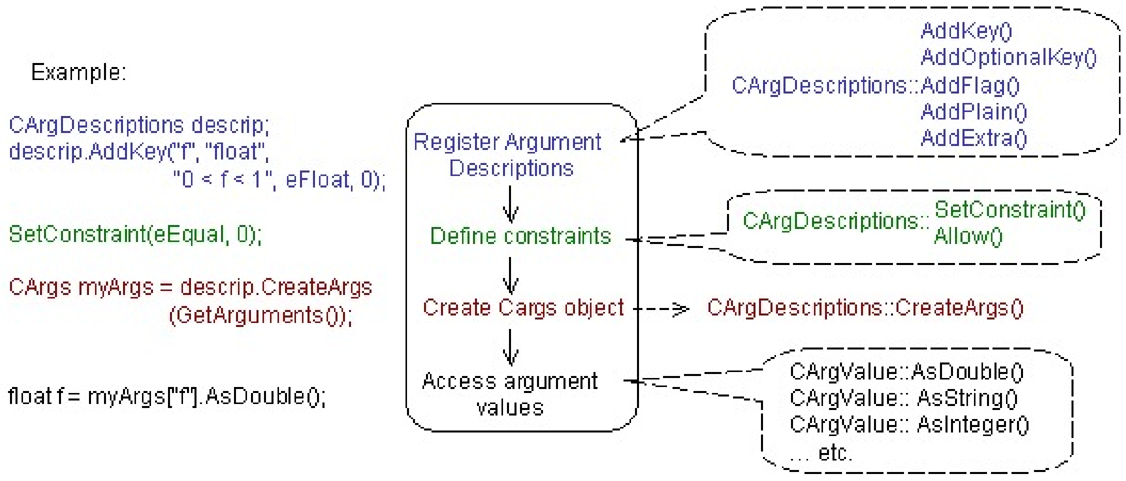

Example:
CArgDescriptions descrip;
descrip.AddKey("f", "float", "0 < f < 1", eFloat, 0);
SetConstraint(eEqual, 0);
CArgs myArgs = descrip.CreateArgs (GetArguments());
float f = myArgs["f"].AsDouble();
Register Argument Descriptions
Define constraints
Create Cargs object
Access argument values
CArgDescriptions::AddKey()
AddOptionalKey()
AddFlag()
AddPlain()
AddExtra()
CArgDescriptions::SetConstraint()
Allow()
CArgDescriptions::CreateArgs()
CArgValue::AsDouble()
CArgValue:: AsString()
CArgValue:: AsInteger()
... etc.

Figure 1. Argument processing class relations.

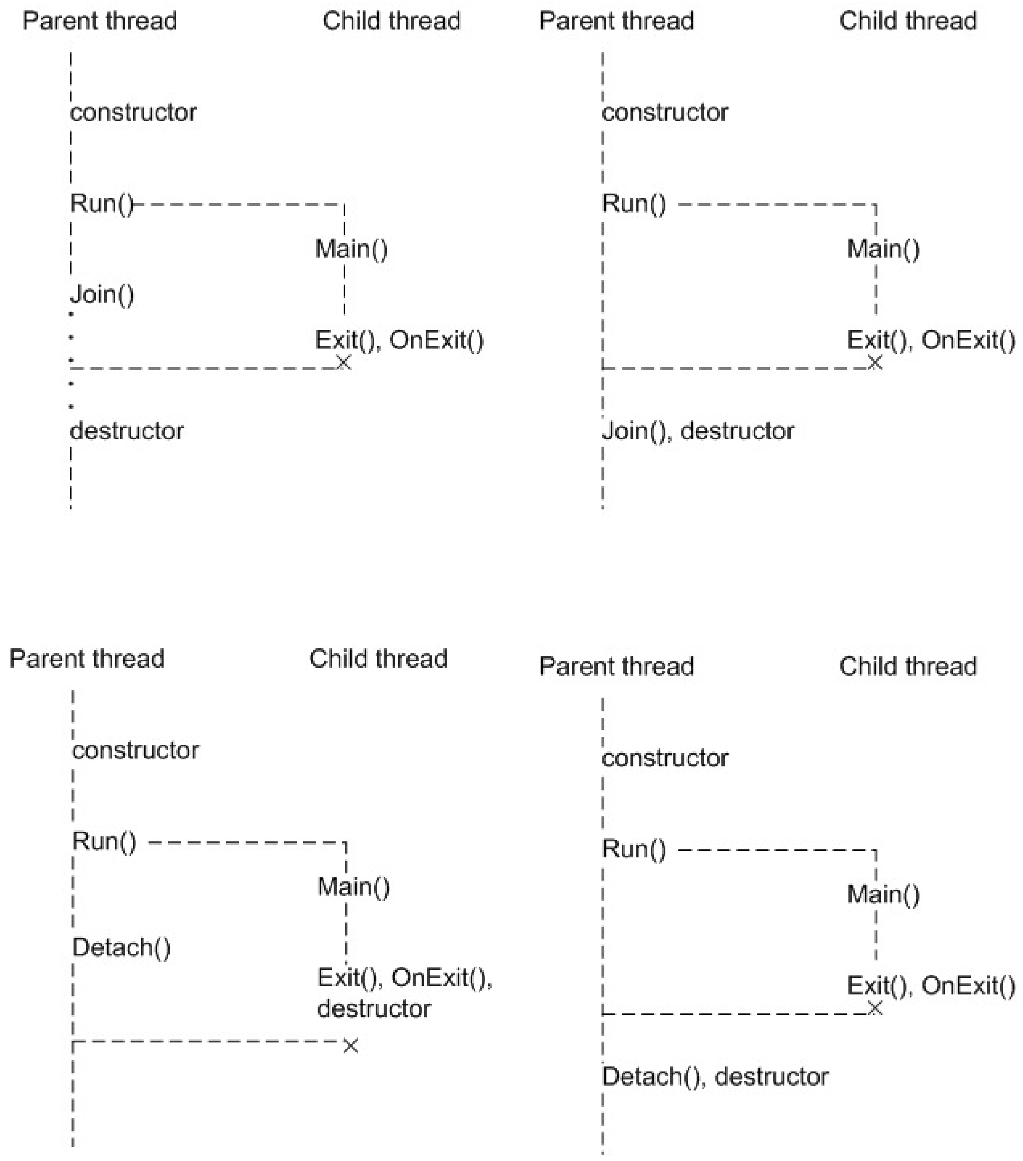


Figure 2. Thread Life Cycle

Table 1. Example of Command-line Arguments

Command-Line Parameters	**File Content**
-gi "Integer" (GI id of the Seq-Entry to examine) OPTIONAL ARGUMENTS: -h (Print this USAGE message; ignore other arguments) -reconstruct (Reconstruct title) -accession (Prepend accession) -organism (Append organism name)	-gi 10200 -reconstruct -accession -organism

Please note:

File must contain Macintosh-style line breaks.

No extra spaces are allowed after argument ("-accession" and not "-accession ").

Arguments must be followed by an empty terminating line.

Table 2. Location of configuration files

conf	**Where to Look for the config File**
empty [default]	Compose the config file name from the base application name plus .ini. Also try to strip file extensions, e.g., for the application named my_app.cgi.exe try subsequently: my_app.cgi.exe.ini, my_app.cgi.ini, my_app.ini. Using these names, search in directories as described in the "Otherwise" case for non-empty conf (see below).
NULL	Do not even try to load the registry at all
non-empty	If conf contains a path, then try to load from the config file named conf (only and exactly!). If the path is not fully qualified and it starts from ../ or ./, then look for the config file starting from the current working dir. **Otherwise** (only a basename, without path), the config file will be searched for in the following places (in the order of preference): 1. current work directory; 2. user home directory; 3. directory defined by environment variable NCBI; 4. system directory; 5. program directory.

Table 3. Standard command-line options for the default instance of CArgDescriptions

Flag	Description	Example
-h	Print description of the application's command-line parameters.	theapp -h
-logfile	Redirect program's log into the specified file.	theapp -logfile theapp_log
-conffile	Read the program's configuration data from the specified file.	theapp -conffile theapp_cfg

Table 4. Filter String Samples

Filter	Description	Matches	Non Matches
/corelib	Log message from source file located in src/corelib or include/corelib or subdirectories	• src/corelib/ncbidiag.cpp • src/corelib/test/test_ncbiexec.cpp • include/corelib/ncbidiag.hpp	• src/cgi/cgiapp.cpp
/corelib/test	Log message from source file located in src/corelib/test or include/corelib/test or subdirectories	• src/corelib/test/test_ncbiexec.cpp	• src/corelib/ncbidiag.cpp • include/corelib/ncbidiag.hpp • src/cgi/cgiapp.cpp
/corelib/	Log message from source file located in src/corelib or include/corelib	• src/corelib/ncbidiag.cpp • include/corelib/ncbidiag.hpp	• src/corelib/test/test_ncbiexec.cpp • src/cgi/cgiapp.cpp
corelib	Log message with module name set to "corelib" and any class or function name	• corelib • corelib::CNcbiDiag • corelib::CNcbiDiag::GetModule()	• CNcbiDiag • CNcbiDiag::GetModule() • GetModule()
corelib::CNcbiDiag	Log message with module name set to "corelib", class name set to "CNcbiDiag" and any function name	• corelib::CNcbiDiag • corelib::CNcbiDiag::GetModule()	• corelib • CNcbiDiag • CNcbiDiag::GetModule() • GetModule()
::CNcbiDiag	Log message with class name set to "CNcbiDiag" and any module or function name	• corelib::CNcbiDiag • corelib::CNcbiDiag::GetModule() • CNcbiDiag • CNcbiDiag::GetModule()	• corelib • GetModule()
?	Log message with module name not set and any class or function name	• CNcbiDiag • CNcbiDiag::GetModule() • GetModule()	• corelib • corelib::CNcbiDiag • corelib::CNcbiDiag::GetModule() • corelib::CNcbiDiag::GetModule()
corelib::?	Log message with module name set to "corelib", class name not set and any function name	• corelib • corelib::GetModule()	• corelib::CNcbiDiag • corelib::CNcbiDiag::GetModule() • CNcbiDiag::GetModule() • GetModule()

GetModule()	Log message with function name set to "GetModule" and any class or module name	• corelib::GetModule() • CNcbiDiag::GetModule() • GetModule()	• Corelib • corelib::CNcbiDiag • CNcbiDiag
(20.11)	Log messages with error code 20 and subcode 11	• ErrCode(20,11)	• ErrCode(20,10) • ErrCode(123,11)
(20-80.)	Log messages with error code from 20 to 80 and any subcode	• ErrCode(20,11) • ErrCode(20,10) • ErrCode(51,1)	• ErrCode(123,11)
(20-80,120,311-400.1-50,60)	Log messages with error code from 20 to 80, 120, from 311 to 400 and subcode from 1 to 50 and 60	• ErrCode(20,11) • ErrCode(321,60)	• ErrCode(20,51) • ErrCode(321,61)

Table 5. Standard C/C++ Types

Name	Size(bytes)	Min	Max	Note
char	1	kMin_Char (0 or -128)	kMax_Char (256 or 127)	It can be either signed or unsigned! Use it wherever you don't care of +/- (e.g. in character strings).
signed char	1	kMin_SChar (-128)	kMax_SChar (127)	
unsigned char	1	kMin_UChar (0)	kMax_UChar (255)	
short, signed short	2 or more	kMin_Short (-32768 or less)	kMax_Short (32767 or greater)	Use "int" if size isn't critical
usigned short	2 or more	kMin_UShort (0)	kMax_UShort (65535 or greater)	Use "unsigned int" if size isn't critical
int, signed int	4 or more	kMin_Int (-2147483648 or less)	kMax_Int (2147483647 or greater)	
unsigned int	4 or more	kMin_UInt (0)	kMax_UInt (4294967295 or greater)	
double	4 or more	kMin_Double	kMax_Double	

Types "long" and "float" are **discouraged** to use in the portable code.

Type "long long" is **prohibited** to use in the portable code.

Table 6. Fixed-integer Types

Name	**Size(bytes)**	**Min**	**Max**
Char, Int1	1	kMin_I1	kMax_I1
Uchar, Uint1	1	0	kMax_UI1
Int2	2	kMin_I2	kMax_I2
Uint2	2	0	kMax_UI2
Int4	4	kMin_I4	kMax_I4
Uint4	4	0	kMax_UI4
Int8	8	kMin_I8	kMax_I8
Uint8	8	0	kMax_UI8

Table 7. Correspondence between the kM*_* constants and the old style INT*_M* constants

Constant(NCBI C++)	**Value**	**Define(NCBI C)**
kMin_I1	-128	INT1_MIN
kMax_I1	+127	INT1_MAX
kMax_UI1	+255	UINT1_MAX
kMin_I2	-32768	INT2_MIN
kMax_I2	+32767	INT2_MAX
kMax_UI2	+65535	UINT2_MAX
kMin_I4	-2147483648	INT4_MIN
kMax_I4	+2147483647	INT4_MAX
kMax_UI4	+4294967295	UINT4_MAX
kMin_I8	-9223372036854775808	INT8_MIN
kMax_I8	+9223372036854775807	INT8_MAX
kMax_UI8	+18446744073709551615	UINT8_MAX

9: Networking and IPC

Last Update: June 21, 2013.

Connection Library [Library xconnect: include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Includes a generic socket interface (SOCK), connection object (CONN), and specialized connector constructors (for sockets, files, HTTP, and services) to be used as engines for connections. It also provides access to the load-balancing daemon and NCBI named service dispatching facilities.

Although the core of the Connection Library is written in C and has an underlying C interface, the analogous C++ interfaces have been built to provide objects that work smoothly with the rest of the Toolkit.

Note: Because of security issues, not all links in the public version of this file are accessible by outside NCBI users.

- Overview
- Connections: notion of connection; different types of connections that the library provides; programming API.
  - Socket Connector
  - File Connector
  - HTTP Connector
  - Service Connector
- Debugging Tools and Troubleshooting
- C++ Connection Streams built on top of connection objects.
- Service mapping API: description of service name resolution API.
- Threaded Server Support

Chapter Outline

The following is an outline of the topics presented in this chapter:

- Basic Types and Functionality (for Registry, Logging and MT Locks) ncbi_core[.h | .c], ncbi_types[.h]
- Portable TCP/IP Socket Interface ncbi_socket[.h | .c]
- Connections and CONNECTORs
  - Open and Manage Connections to an Abstract I/O ncbi_connection[.h | .c]
  - Implement CONNECTOR for a ...
    - Abstract I/O ncbi_connector[.h | .c]
    - Network Socket ncbi_socket_connector[.h | .c]
    - FILE Stream ncbi_file_connector[.h | .c]
    - HTTP-based Network Connection ncbi_http_connector[.h | .c]
    - Named NCBI Service ncbi_service_connector[.h | .c]
    - In-memory CONNECTOR ncbi_memory_connector[.h | .c]
- Servers and Services
  - NCBI Server Meta-Address Info ncbi_server_info[.h | p.h | .c]
  - Resolve NCBI Service Name to the Server Meta-Address ncbi_service[.h | p.h | .c]
  - Resolve NCBI Service Name to the Server Meta-Address using NCBI Network Dispatcher (DISPD) ncbi_service[p_dispd.h | _dispd.c]
  - Resolve NCBI Service Name to the Server Meta-Address using NCBI Load-Balancing Service Mapper (LBSM) ncbi_service[p_lbsmd.h | _lbsmd.c | _lbsmd_stub.c]
  - NCBI LBSM client-server data exchange API ncbi_lbsm[.h | .c]
  - Implementation of LBSM Using SYSV IPC (shared memory and semaphores) ncbi_lbsm_ipc[.h | .c]
- Memory Management
  - Memory-Resident FIFO Storage Area ncbi_buffer[.h | .c]
  - Simple Heap Manager With Primitive Garbage Collection ncbi_heapmgr[.h | .c]
- Connection Library Utilities
  - Connection Utilities ncbi_connutil[.h | .c]
  - Send Mail (in accordance with RFC821 [protocol] and RFC822 [headers]) ncbi_sendmail[.h | .c]
  - Auxiliary (optional) Code for ncbi_core.[ch] ncbi_util[.h | .c]
  - Non-ANSI, Widely Used Functions ncbi_ansi_ext[.h | .c]

daemons [src/connect/daemons]

- LBSMD
- DISPD
- Firewall Daemon

Test Cases [src/connect/test]

Overview

The NCBI C++ platform-independent connection library (src/connect and include/connect) consists of two parts:

- Lower-level library written in C (also used as a replacement of the existing connection library in the NCBI C Toolkit)
- Upper-level library written in C++ and using C++ streams

Functionality of the library includes:

- SOCK interface (sockets), which works interchangeably on most UNIX varieties, MS Windows, and Mac
- SERV interface, which provides mapping of symbolic service names into server addresses
- CONN interface, which allows the creation of a connection, the special object capable to do read, write, etc. I/O operations
- C++ streams built on top of the CONN interface

Note: The lowest level (SOCK) interface is not covered in this document. A well-commented API can be found in connect/ncbi_socket.h.

Connections

There are three simple types of connections: socket, file and http; and one hybrid type, service connection.

A connection is created with a call to CONN_Create(), declared in connect/ncbi_connection.h, and returned by a pointer to CONN passed as a second argument:

```
CONN conn; /* connection handle */
EIO_Status status = CONN_Create(connector, &conn);
```

The first argument of this function is a handle of a connector, a special object implementing functionality of the connection being built. Above, for each type of connection there is a special connector in the library. For each connector, one or more "constructors" are defined, each returning the connector's handle. Connectors' constructors are defined in individual header files, such as connect/ncbi_socket_connector.h, connect/ncbi_http_connector.h, connect/ncbi_service_connector.h, etc. CONN_Create() resets all timeouts to the default value kDefaultTimeout.

After successful creation with CONN_Create(), the following calls from CONN API connect/ncbi_connection.h become available. All calls (except CONN_GetTimeout() and CONN_GetType()) return an I/O completion status of type EIO_Status. Normal completion has code eIO_Success.

Note: There is no means to "open" a connection: it is done automatically when actually needed, and in most cases at the first I/O operation. But the forming of an actual link between source and destination can be postponed even longer. These details are hidden and made transparent to the connection's user. The connection is seen as a two-way communication channel, which is clear for use immediately after a call to CONN_Create().

Note: If for some reason CONN_Create() failed to create a connection (return code differs from eIO_Success), then the connector passed to this function is left intact, that is, its handle can be used again. Otherwise, if the connection is created successfully, the passed connector handle becomes invalid and cannot be referenced anywhere else throughout the program (with one exception, however: it may be used as a replacing connector in a call to CONN_ReInit() for the same connection).

Note: There are no "destructors" on public connectors. A connector successfully put into connection is deleted automatically, along with that connection by CONN_Close(), or explicitly with a call to CONN_ReInit(), provided that the replacing connector is NULL or different from the original.

```
CONN_Read(CONN conn, void* buf, size_t size, size_t* n_read,
EIO_ReadMethod how)
```

Read or peek data, depending on read method how, up to size bytes from connection to specified buffer buf, return (via pointer argument n_read) the number of bytes actually read. The last argument how can be one of the following:

- eIO_ReadPlain - to read data in a regular way, that is, extracting data from the connection;
- eIO_ReadPeek - to peek data from the connection, i.e., the next read operation will see the data again;
- eIO_ReadPersist - to read exactly (not less than) size bytes or until an error condition occurs.

A return value other than eIO_Success means trouble. Specifically, the return value eIO_Timeout indicates that the operation could not be completed within the allotted amount of time; but some data may, however, be available in the buffer (e.g., in case of persistent reading, as with eIO_ReadPersist), and this is actually the case for any return code.

```
CONN_ReadLine(CONN conn, char* line, size_t size, size_t* n_read)
```

Read up to size bytes from connection into the string buffer pointed to by line. Stop reading if either '\n' or an error is encountered. Replace '\n' with '\0'. Upon return *n_read contains the number of characters written to line, not including the terminating '\0'. If not enough space provided in line to accomodate the '\0'-terminated line, then all size bytes are used up and *n_read is equal to size upon return - this is the only case when line will not be '\0'-terminated.

Return code advises the caller whether another read can be attempted:

- eIO_Success -- read completed successfully, keep reading;
- other code -- an error occurred, and further attempt may fail.

This call utilizes eIO_Read timeout as set by CONN_SetTimeout().

```
CONN_Write(CONN conn, const void* buf, size_t size, size_t* n_written)
```

Write up to size bytes from the buffer buf to the connection. Return the number of actually written bytes in n_written. It may not return eIO_Success if no data at all can be written before the write timeout expired or an error occurred. Parameter how modifies the write behavior:

- eIO_WritePlain - return immediately after having written as little as 1 byte of data, or if an error has occurred;
- eIO_WritePersist - return only after having written all of the data from buf (eIO_Success), or if an error has occurred (fewer bytes written, non-eIO_Success).

Note: See CONN_SetTimeout() for how to set the write timeout.

```
CONN_PushBack(CONN conn, const void* buf, size_t size)
```

Push back size bytes from the buffer buf into connection. Return eIO_Success on success, other code on error.

Note 1: The data pushed back may not necessarily be the same as previously obtained from the connection.

Note 2: Upon a following read operation, the pushed back data are taken out first.

```
CONN_GetPosition(CONN conn, EIO_Event event)
```

Get read (event == eIO_Read) or write (event == eIO_Write) position within the connection. Positions are advanced from 0 on, and only concerning I/O that has caused calling to the actual connector's "read" (i.e. pushbacks never considered, and peeks -- not always) and "write" methods. Special case: eIO_Open as event causes to clear both positions with 0, and to return 0.

```
CONN_Flush(CONN conn)
```

Explicitly flush connection from any pending data written by CONN_Write().

Note 1: CONN_Flush() effectively opens connection (if it wasn't open yet).

Note 2: Connection considered open if underlying connector's "Open" method has successfully executed; an actual data link may not yet exist.

Note 3: CONN_Read() always calls CONN_Flush() before proceeding; so does CONN_Close () but only if the connection is already open.

```
CONN_SetTimeout(CONN conn, EIO_Event action, const STimeout* timeout)
```

Set the timeout on the specified I/O action, eIO_Read, eIO_Write, eIO_ReadWrite, eIO_Open, and eIO_Close. The latter two actions are used in a phase of opening and closing the link, respectively. If the connection cannot be read (written, established, closed) within the specified period, eIO_Timeout would result from connection I/O calls. A timeout can be passed as the NULL-pointer. This special case denotes an infinite value for that timeout. Also, a special value kDefaultTimeout may be used for any timeout. This value specifies the timeout set by default for the current connection type.

```
CONN_GetTimeout(CONN conn, EIO_Event action)
```

Obtain (via the return value of type const STimeout*) timeouts set by the CONN_SetTimeout () routine, or active by default (i.e., special value kDefaultTimeout).

Caution: The returned pointer is valid only for the time that the connection handle is valid, i.e., up to a call to CONN_Close().

```
CONN_ReInit(CONN conn, CONNECTOR replacement)
```

This function clears the current contents of a connection and places ("immerse") a new connector into it. The previous connector (if any) is closed first (if open), then gets destroyed, and thus must not be referenced again in the program. As a special case, the new connector can be the same connector, which is currently active within the connection. It this case, the connector is not destroyed; instead, it will be effectively re-opened. If the connector passed as NULL, then the conn handle is kept existing but unusable (the old connector closed and destroyed) and can be CONN_ReInit()'ed later. None of the timeouts are touched by this call.

```
CONN_Wait(CONN conn, EIO_Event event, const STimeout* timeout)
```

Suspend the program until the connection is ready to perform reading (event =eIO_Read) or writing (event = eIO_Write), or until the timeout (if non-NULL) expires. If the timeout is passed as NULL, then the wait time is indefinite.

```
CONN_Status(CONN conn, EIO_Event direction)
```

Provide the information about recent low-level data exchange in the link. The operation direction has to be specified as either eIO_Read or eIO_Write. The necessity of this call arises from the fact that sometimes the return value of a CONN API function does not really tell that the problem has been detected: suppose, the user peeks data into a 100-byte buffer and gets 10 bytes. The return status eIO_Success signals that those 10 bytes were found in the connection okay. But how do you know whether the end-of-file condition occurred during the last operation? This is where CONN_Status() comes in handy. When inquired about the read operation, return value eIO_Closed denotes that EOF was actually hit while making the peek, and those 10 bytes are in fact the only data left untaken, no more are expected to come.

```
CONN_Close(CONN conn)
```

Close the connection by closing the link (if open), deleting underlying connector(s) (if any) and the connection itself. Regardless of the return status (which may indicate certain problems), the connection handle becomes invalid and cannot be reused.

```
CONN_Cancel(CONN conn)
```

Cancel the connection's I/O ability. This is **not** connection closure, but any data extraction or insertion (Read/Write) will be effectively rejected after this call (and eIO_Interrupt will result, same for CONN_Status()). CONN_Close() is still required to release internal connection structures.

```
CONN_GetType(CONN conn)
```

Return character string (null-terminated), verbally representing the current connection type, such as "HTTP", "SOCKET", "SERVICE/HTTP", etc. The unknown connection type gets returned as NULL.

```
CONN_Description(CONN conn)
```

Return a human-readable description of the connection as a character '\0'-terminated string. The string is not guaranteed to have any particular format and is intended solely for something like logging and debugging. Return NULL if the connection cannot provide any description information (or if it is in a bad state). Application program must call free() to deallocate space occupied by the returned string when the description is no longer needed.

```
CONN_SetCallback(CONN conn, ECONN_Callback type,
const SCONN_Callback* new_cb, SCONN_Callback* old_cb)
```

Set user callback function to be invoked upon an event specified by callback type. The old callback (if any) gets returned via the passed pointer old_cb (if not NULL). Callback structure SCONN_Callback has the following fields: callback function func and void* data. Callback function func should have the following prototype:

typedef void (*FCONN_Callback)(CONN conn, ECONN_Callback type, void* data)

When called, both type of callback and data pointer are supplied. The callback types defined at the time of this writing are:

- eCONN_OnClose
- eCONN_OnRead
- eCONN_OnWrite
- eCONN_OnCancel

The callback function is always called prior to the event happening, e.g., a close callback is called when the connection is about to close.

Socket Connector

Constructors are defined in:

```
#include <connect/ncbi_socket_connector.h>
```

A socket connection, based on the socket connector, brings almost direct access to the SOCK API. It allows the user to create a peer-to-peer data channel between two programs, which can be located anywhere on the Internet.

To create a socket connection, user has to create a socket connector first, then pass it to CONN_Create(), as in the following example:

```
#include <connect/ncbi_socket_connector.h>
#include <connect/ncbi_connection.h>

#define MAX_TRY 3 /* Try to connect this many times before giving up */

unsigned short port = 1234;
CONNECTOR socket_connector = SOCK_CreateConnector("host.foo.com", port,
 MAX_TRY);
if (!socket_connector)
 fprintf(stderr, "Cannot create SOCKET connector");
```

```
else {
 CONN conn;
 if (CONN_Create(socket_connector, &conn) != eIO_Success)
 fprintf(stderr, "CONN_Create failed");
 else {
 /* Connection created ok, use CONN_... function */
 /* to access the network */
 ...
 CONN_Close(conn);
 }
}
```

A variant form of this connector's constructor, SOCK_CreateConnectorEx(), takes three more arguments: a pointer to data (of type void*), data size (bytes) to specify the data to be sent as soon as the link has been established, and flags.

The CONN library defines two more constructors, which build SOCKET connectors on top of existing SOCK objects: SOCK_CreateConnectorOnTop() and SOCK_CreateConnectorOnTopEx(), the description of which is intentionally omitted here because SOCK is not discussed either. Please refer to the description in the Toolkit code.

File Connector

Constructors defined in:

```
#include <connect/ncbi_file_connector.h>

CONNECTOR file_connector = FILE_CreateConnector("InFile", "OutFile");
```

This connector could be used for both reading and writing files, when input goes from one file and output goes to another file. (This differs from normal file I/O, when a single handle is used to access only one file, but rather resembles data exchange via socket.)

Extended variant of this connector's constructor, FILE_CreateConnectorEx(), takes an additional argument, a pointer to a structure of type SFILE_ConnAttr describing file connector attributes, such as the initial read position to start from in the input file, an open mode for the output file (append eFCM_Append, truncate eFCM_Truncate, or seek eFCM_Seek to start writing at a specified file position), and the position in the output file, where to begin output. The attribute pointer passed as NULL is equivalent to a call to FILE_CreateConnector(), which reads from the very beginning of the input file and entirely overwrites the output file (if any), implicitly using eFCM_Truncate.

Connection-related parameters for higher-level connection protocols

The network information structure (from connect/ncbi_connutil.h) defines parameters of the connection point, where a server is running. See the Library Configuration chapter for descriptions of the corresponding configuration parameters.

Note: Not all parameters of the structure shown below apply to every network connector.

```
/* Network connection related configurable info struct.
 * ATTENTION: Do NOT fill out this structure (SConnNetInfo) "from scratch"!
 * Instead, use ConnNetInfo_Create() described below to create
 * it, and then fix (hard-code) some fields, if really necessary.
 * NOTE1: Not every field may be fully utilized throughout the library.
 * NOTE2: HTTP passwords can be either clear text or Base64 encoded values
 * enclosed in square brackets [] (which are not Base-64 charset).
 * For encoding / decoding, one can use command line open ssl:
 * echo "password|base64value" | openssl enc {-e|-d} -base64
 * or an online tool (search the Web for "base64 online").
 */
typedef struct {
 char client_host[256]; /* effective client hostname ('\0'=def)*/
 EURLScheme scheme; /* only pre-defined types (limited) */
 char user[64]; /* username (if specified) */
 char pass[64]; /* password (if any) */
 char host[256]; /* host to connect to */
 unsigned short port; /* port to connect to, host byte order */
 char path[1024]; /* service: path(e.g. to a CGI script)*/
 char args[1024]; /* service: args(e.g. for a CGI script)*/
 EReqMethod req_method; /* method to use in the request (HTTP) */
 const STimeout* timeout; /* ptr to i/o tmo (infinite if NULL) */
 unsigned short max_try; /* max. # of attempts to connect (>= 1)*/
 char http_proxy_host[256]; /* hostname of HTTP proxy server */
 unsigned short http_proxy_port; /* port # of HTTP proxy server */
 char http_proxy_user[64]; /* http proxy username */
 char http_proxy_pass[64]; /* http proxy password */
 char proxy_host[256]; /* CERN-like (non-transp) f/w proxy srv*/
 EDebugPrintout debug_printout; /* printout some debug info */
 int/*bool*/ stateless; /* to connect in HTTP-like fashion only*/
 int/*bool*/ firewall; /* to use firewall/relay in connects */
 int/*bool*/ lb_disable; /* to disable local load-balancing */
 const char* http_user_header; /* user header to add to HTTP request */
 const char* http_referer; /* default referrer (when not spec'd) */

 /* the following field(s) are for the internal use only -- don't touch! */
 STimeout tmo; /* default storage for finite timeout */
 const char svc[1]; /* service which this info created for */
} SConnNetInfo;
```

Caution: Unlike other "static fields" of this structure, http_user_header (if non-NULL) is assumed to be dynamically allocated on the heap (via a call to malloc, calloc, or a related function, such as strdup).

ConnNetInfo convenience API

Although users can create and fill out this structure via field-by-field assignments, there is, however, a better, easier, much safer, and configurable way (the interface is defined in connect/ncbi_connutil.h) to deal with this structure:

- ConnNetInfo_Create(const char* service)

Create and return a pointer to new SConnNetInfo structure, filled with parameters specific either for a named service or by default (if the service is specified as NULL - most likely the case for ordinary HTTP connections). Parameters for the structure are taken from (in the order of precedence):

- Environment variables of the form <service>_CONN_<name>, where name is the name of the field;
- Service-specific registry section (see below the note about the registry) named [service] using the key CONN_<name>;
- environment variable of the form CONN_<name>
- registry section named [CONN] using name as a key
- default value applied, if none of the above resulted in a successful match

Search for the keys in both environment and registry is not case-sensitive, but the values of the keys are case sensitive (except for enumerated types and boolean values, which can be of any, even mixed, case). Boolean fields accept 1, "ON", "YES", and "TRUE" as true values; all other values are treated as false. In addition to a floating point number treated as a number of seconds, timeout can accept (case-insensitively) keyword "INFINITE".

Note: The first two steps in the above sequence are skipped if the service name is passed as NULL.

Caution: The library can not provide reasonable default values for path and args when the structure is used for HTTP connectors.

- ConnNetInfo_Destroy(SConnNetInfo* info)

Delete and free the info structure via a passed pointer (note that the HTTP user header http_user_header is freed as well).

- ConnNetInfo_SetUserHeader(SConnNetInfo* info, const char* new_user_header)

Set the new HTTP user header (freeing the previous one, if any) by cloning the passed string argument and storing it in the http_user_header field. New_user_header passed as NULL resets the field.

- ConnNetInfo_Clone(SConnNetInfo* info)

Create and return a pointer to a new SConnNetInfo structure, which is an exact copy of the passed structure. This function is recognizes the dynamic nature of the HTTP user header field.

Note about the registry. The registry used by the connect library is separate from the CNcbiRegistry class. To learn more about the difference and how to use both objects together in a single program, please see Using NCBI C and C++ Toolkits Together.

HTTP Connector

Constructors defined in:

```
#include <connect/ncbi_http_connector.h>
```

The simplest form of this connector's constructor takes three parameters:

```
CONNECTOR HTTP_CreateConnector(const SConnNetInfo* net_info,
 const char* user_header,
 THCC_Flags flags);
```

a pointer to the network information structure (can be NULL), a pointer to a custom HTTP tag-value(s) called a user-header, and a bitmask of various flags. The user-header has to be in the form "HTTP-Tag: Tag-value\r\n", or even multiple tag-values delimited and terminated by

"\r\n". If specified, the user_header parameter overrides the corresponding field in the passed net_info.

The following fields of SConnNetInfo pertain to the HTTP connector: client_host, host, port, path, args, req_method (can be one of "GET", "POST", and "ANY"), timeout, max_try (analog of maximal try parameter for the socket connector), http_proxy_host, http_proxy_port, and debug_printout (values are "NONE" to disable any trace printout of the connection data, "SOME" to enable printing of SConnNetInfo structure before each connection attempt, and "DATA" to print both headers and data of the HTTP packets in addition to dumps of SConnNetInfo structures). Values of other fields are ignored.

HTTP connector's flags

Argument flags in the HTTP connector's constructor is a bitwise OR of the following values:

- fHTTP_AutoReconnect Allow multiple request/reply HTTP transactions. (Otherwise, by default, only one request/reply is allowed.)
- fHTTP_SureFlush Always flush a request (may consist solely of HTTP header with no body at all) down to the HTTP server before preforming any read or close operations.
- fHTTP_KeepHeader By default, the HTTP connection sorts out the HTTP header and parses HTTP errors (if any). Thus, reading normally from the connection returns data from the HTTP body only. The flag disables this feature, and the HTTP header is not parsed but instead is passed "as is" to the application on a call to CONN_Read().
- fHTTP_UrlDecodeInput Decode input data passed in HTTP body from the HTTP server.
- fHTTP_UrlEncodeOutput Encode output data passed in the HTTP body to the HTTP server.
- fHTTP_UrlCodec Perform both decoding and encoding (fHTTP_UrlDecodeInput | fHTTP_UrlEncodeOutput).
- fHTTP_UrlEncodeArgs Encode URL if it contains special characters such as "+". By default, the arguments are passed "as is" (exactly as taken from SConnNetInfo).
- fHTTP_DropUnread Drop unread data, which might exist in connection, before making another request/reply HTTP shot. Normally, the connection first tries to read out the data from the HTTP server entirely, until EOF, and store them in the internal buffer, even if either application did not request the data for reading, or the data were read only partially, so that the next read operation will see the data.
- fHTTP_NoUpread Do not attempt to empty incoming data channel into a temporary intermediate buffer while writing to the outgoing data channel. By default, writing always makes checks that incoming data are available for reading, and those data are extracted and stored in the buffer. This approach avoids I/O deadlock, when writing creates a backward stream of data, which, if unread, blocks the connection entirely.
- fHTTP_Flushable By default all data written to the connection are kept until read begins (even though Flush() might have been called in between the writes); with this flag set, Flush() will result the data to be actually sent to server side, so the following write will form new request, and not get added to the previous one.
- fHTTP_InsecureRedirect For security reasons the following redirects comprise security risk and, thus, are prohibited: switching from https to http, and re-posting data (regardless of the transport, either http or https); this flag allows such redirects (if needed) to be honored.

- fHTTP_NoAutoRetry Do not attempt any auto-retries in case of failing connections (this flag effectively means having SConnNetInfo::max_try set to 1).
- fHTTP_DetachableTunnel SOCK_Close() won't close the OS handle.

The HTTP connection will be established using the following URL: http://host:port/path?args

Note that path has to have a leading slash "/" as the first character, that is, only "http://" and "?" are added by the connector; all other characters appear exactly as specified (but maybe encoded with fHTTP_UrlEncodeArgs). The question mark does not appear if the URL has no arguments.

A more elaborate form of the HTTP connector's constructor has the following prototype:

```
typedef int/*bool*/ (*FHTTP_ParseHeader)
(const char* http_header, /* HTTP header to parse, '\0'-terminated */
 void* user_data, /* supplemental user data */
 int server_error /* != 0 if HTTP error */
 );

typedef int/*bool*/ (*FHTTP_Adjust)
(SConnNetInfo* net_info, /* net_info to adjust (in place) */
 void* user_data, /* supplemental user data */
 unsigned int failure_count /* how many failures since open */
 );

typedef void (*FHTTP_Cleanup)
(void* user_data /* supplemental user data for cleanup */
 );

CONNECTOR HTTP_CreateConnectorEx
(const SConnNetInfo* net_info,
 THTTP_Flags flags,
 FHTTP_ParseHeader parse_header, /* may be NULL, then no addtl. parsing */
 void* user_data, /* user data for HTTP callbacks (CBs) */
 FHTTP_Adjust adjust, /* may be NULL, then no adjustments */
 FHTTP_Cleanup cleanup /* may be NULL, then no cleanup */
 );
```

This form is assumed to be used rarely by the users directly, but it provides rich access to the internal management of HTTP connections.

The first two arguments are identical to their counterparts in the arguments number one and three of HTTP_CreateConnector(). The HTTP user header field (if any) is taken directly from the http_user_header field of SConnNetInfo, a pointer to which is passed as net_info (which in turn can be passed as NULL, meaning to use the environment, registry, and defaults as described above).

The third parameter specifies a callback to be activated to parse the HTTP reply header (passed as a single string, with CR-LF (carriage return/line feed) characters incorporated to divide header lines). This callback also gets some custom data user_data as supplied in the fourth argument of the connector's constructor and a boolean value of true if the parsed response code from the server was not okay. The callback can return false (zero), which is considered the same way as having an error from the HTTP server. However, the pre-parsed error condition (passed in server_error) is retained, even if the return value of the callback is true, i.e. the callback is unable to "fix" the error code from the server. This callback is **not called** if fHTTP_KeepHeader is set in flags.

The fifth argument is a callback, which is in control when an attempt to connect to the HTTP server has failed. On entry, this callback has current SConnNetInfo, which is requested for an adjusted in an attempt that the connection to the HTTP server will finally succeed. That is, the callback can change anything in the info structure, and the modified structure will be kept for all further connection attempts, until changed by this callback again. The number (starting from 1) of successive failed attempts is given in the argument of the last callback. The callback return value true (non-zero) means a successful adjustment. The return value false (zero) stops connection attempts and returns an error to the application.

When the connector is being destroyed, the custom object user_data can be destroyed in the callback, specified as the last argument of the extended constructor.

Note: Any callback may be specified as NULL, which means that no action is foreseen by the application, and default behavior occurs.

Service Connector

Constructors defined in:

```
#include <connect/ncbi_service_connector.h>
```

This is the most complex connector in the library. It can initiate data exchange between an application and a named NCBI service, and the data link can be either wrapped in HTTP or be just a byte-stream (similar to a socket). In fact, this connector sits on top of either HTTP or SOCKET connectors.

The library provides two forms of the connector's constructor:

```
SERVICE_CreateConnector(const char* service_name);
SERVICE_CreateConnectorEx(
 const char* service_name,
 /* The registered name of an NCBI service */
 TSERV_Type types, /* Accepted server types, bitmask */
 const SConnNetInfo* net_info, /* Connection parameters */
 const SSERVICE_Extra* params); /* Additional set of parameters, may be NULL
*/
```

The first form is equivalent to SERVICE_CreateConnectorEx(service_name, fSERV_Any, 0, 0). A named NCBI service is a CGI program or a stand-alone server (can be one of two supported types), which runs at the NCBI site and is accessible by the outside world. A special dispatcher (which runs on the NCBI Web servers) performs automatic switching to the appropriate server without the client having to know the actual connection point. The client just uses the main entry gate of the NCBI Web (usually, www.ncbi.nlm.nih.gov) with a request to have a service "service_name". Then, depending on the service availability, the request will be declined, rejected, or honored by switching and routing the client to the machine actually running the server.

To the client, the entire process of dispatching is completely transparent (for example, try clicking several times on either of the latter two links and see that the error replies are actually sent from different hosts, and the successful processing of the first link is done by one of several hosts running the bouncing service).

Note: Services can be redirected.

The Dispatching Protocol per se is implemented on top of HTTP protocol and is parsed by a CGI program dispd.cgi (or another dispatching CGI), which is available on the NCBI Web. On every server running the named services, another program, called the load-balancing daemon (lbsmd), is executing. This daemon supports having the same service running on different machines and provides a choice of the one machine that is less loaded. When dispd.cgi receives a request for a named service, it first consults the load-balancing table, which is broadcasted by each load-balancing daemon and populated in a network-wide form on each server. When the corresponding server is found, the client request can be passed, or a dedicated connection to the server can be established. The dispatching is made in such a way that it can be also used directly from most Internet browsers.

The named service facility uses the following distinction of server types:

- HTTP servers, which are usually CGI programs:
  - HTTP_GET servers are those accepting requests only using the HTTP GET method.
  - HTTP_POST servers are those accepting requests only using the HTTP POST method.
  - HTTP servers are those accepting both of either GET or POST methods.
- NCBID servers are those run by a special CGI engine, called ncbid.cgi, a configurable program (now integrated within dispd.cgi itself) that can convert byte-stream output from another program (server) started by the request from a dispatcher, to an HTTP-compliant reply (that is, a packet having both HTTP header and body, and suitable, for example, for Web browsers).
- STANDALONE servers, similar to mailing daemons, are those listening to the network, on their own, for incoming connections.
- FIREWALL servers are the special pseudo-servers, not existing in reality, but that are created and used internally by the dispatcher software to indicate that only a firewall connection mode can be used to access the requested service.
- DNS servers are beyond the scope of this document because they are used to declare domain names, which are used internally at the NCBI site to help load-balancing based on DNS lookup (see here).

A formal description of these types is given in connect/ncbi_server_info.h:

```
/* Server types
 */
typedef enum {
 fSERV_Ncbid = 0x01,
 fSERV_Standalone = 0x02,
 fSERV_HttpGet = 0x04,
 fSERV_HttpPost = 0x08,
 fSERV_Http = fSERV_HttpGet | fSERV_HttpPost,
 fSERV_Firewall = 0x10,
 fSERV_Dns = 0x20
} ESERV_Type;
```

```
#define fSERV_Any 0
#define fSERV_StatelessOnly 0x80
typedef unsigned TSERV_Type; /* bit-wise OR of "ESERV_Type" flags */
```

The bitwise OR of the ESERV_Type flags can be used to restrict the search for the servers, matching the requested service name. These flags passed as argument types are used by the dispatcher when figuring out which server is acceptable for the client. A special value 0 (or, better fSERV_Any) states no such preference whatsoever. A special bit-value fSERV_StatelessOnly set, together with other bits or just alone, specifies that the servers should be of stateless (HTTP-like) type only, and it is the client who is responsible for keeping track of the logical sequence of the transactions.

The following code fragment establishes a service connection to the named service "io_bounce", using only stateless servers:

```
CONNECTOR c;
CONN conn;
if(!(c = SERVICE_CreateConnectorEx("io_bounce", fSERV_StatelessOnly, 0, 0)))
{
 fprintf(stderr, "No such service available");
} else if (CONN_Create(c, &conn) != eIO_Success) {
 fprintf(stderr, "Failed to create connection");
} else {
 static const char buffer[] = "Data to pass to the server";
 size_t n_written;
 CONN_Write(conn, buffer, sizeof(buffer) - 1, &n_written);
 ...
}
```

The real type of the data channel can be obtained via a call to CONN_GetType().

Note: In the above example, the client has no assumption of how the data actually passed to the server. The server could be of any type in principle, even a stand-alone server, which was used in the request/reply mode of one-shot transactions. If necessary, such wrapping would have been made by the dispatching facility as well.

The next-to-last parameter of the extended constructor is the network info, described in the section devoted to the HTTP connector. The service connector uses all fields of this structure, except for http_user_header, and the following assumptions apply:

- path specifies the dispatcher program (defaulted to dispd.cgi)
- args specifies parameters for the requested service (this is service specific, no defaults)
- stateless is used to set the fSERV_StatelessOnly flag in the server type bitmask, if it was not set there already (which is convenient for modifying the dispatch by using environment and/or registry, if the flag is not set, yet allows hardcoding of the flag at compile time by setting it in the constructor's types argument explicitly)
- lb_disable set to true (non-zero) means to always use the remote dispatcher (via network connection), even if locally running load-balancing daemon is available (by default, the local load-balancing deamon is consulted first to resolve the name of the service)
- firewall set to true (non-zero) disables the direct connection to the service; instead,

- — a connection to a proxy firewall daemon (fwdaemon), running at the NCBI site, is initiated to pass the data in stream mode;
- — or data get relayed via the dispatcher, if the stateless server is used

- http_user_header merged not to conflict with special dispatcher parameter.

As with the HTTP connector, if the network information structure is specified as NULL, the default values are obtained as described above, as with the call to ConnNetInfo_Create (service_name).

Normally, the last parameter of SERVICE_CreateConnectorEx() is left NULL, which sets all additional parameters to their default values. Among others, this includes the default procedure of choosing an appropriate server when the connector is looking for a mapping of the service name into a server address. To see how this parameter can be used to change the mapping procedure, please see the service mapping API section. The library provides an additional interface to the service mapper, which can be found in connect/ncbi_service.h.

Note: Requesting fSERV_Firewall in the types parameter effectively selects the firewall mode regardless of the network parameters, loaded via the SConnNetInfo structure.

Service Redirection

Services can be redirected without changing any code - for example, to test production code with a test service, or for debugging. Services are redirected using the <service>_CONN_SERVICE_NAME environment variable or the [<service>] CONN_SERVICE_NAME registry entry (see the connection library configuration section). The client application will use the original service name, but the connection will actually be made to the redirected-to service.

Debugging Tools and Troubleshooting

Each connector (except for the FILE connector) provides a means to view data flow in the connection. In case of the SOCKET connector, debugging information can be turned on by the last argument in SOCK_CreateConnectorEx() or by using the global routine SOCK_SetDataLoggingAPI() (declared in connect/ncbi_socket.h)

Note: In the latter case, every socket (including sockets implicitly used by other connectors such as HTTP or SERVICE) will generate debug printouts.

In case of HTTP or SERVICE connectors, which use SConnNetInfo, debugging can be activated directly from the environment by setting CONN_DEBUG_PRINTOUT to TRUE or SOME. Similarly, a registry key DEBUG_PRINTOUT with a value of either TRUE or SOME found in the section [CONN] will have the same effect: it turns on logging of the connection parameters each time the link is established. When set to ALL, this variable (or key) also turns on debugging output on all underlying sockets ever created during the life of the connection. The value FALSE (default) turns debugging printouts off. Moreover, for the SERVICE connector, the debugging output option can be set on a per-service basis using <service>_CONN_DEBUG_PRINTOUT environment variables or individual registry sections [<service>] and the key CONN_DEBUG_PRINTOUT in them.

Note: Debugging printouts can only be controlled in a described way via environment or registry if and only if SConnNetInfo is always created with the use of convenience routines.

Debugging output is always sent to the same destination, the CORE log file, which is a C object shared between both C and C++ Toolkits. As said previously, the logger is an abstract object, i.e. it is empty and cannot produce any output if not populated accordingly. The library defines a few calls gathered in connect/ncbi_util.h, which allow the logger to output via the FILE file pointer, such as stderr: CORE_SetLOGFILE() for example, as shown in test_ncbi_service_connector.c, or to be a regular file on disk. Moreover, both Toolkits define interfaces to deal with registries, loggers, and locks that use native objects of each toolkit and use them as replacements for the objects of the corresponding abstract layers.

There is a common problem that has been reported several times and actually concerns network configuration rather than representing a malfunction in the library. If a test program, which connects to a named NCBI service, is not getting anything back from the NCBI site, one first has to check whether there is a transparent proxying/caching between the host and NCBI. Because the service dispatching is implemented on top of the ordinary HTTP protocol, the transparent proxying may latch unsuccessful service searches (which can happen and may not indicate a real problem) as error responses from the NCBI server. Afterwards, instead of actually connecting to NCBI, the proxy returns those cached errors (or sometimes just an empty document), which breaks the service dispatcher code. In most cases, there are configurable ways to exclude certain URLs from proxying and caching, and they are subject for discussion with a local network administrator.

Here is another tip: Make sure that all custom HTTP header tags (as passed, for example, in the SConnNetInfo::user_header field) have "\r\n" as tag separators (including the last tag). Many proxy servers (including transparent proxies, of which the user may not even be aware) are known to be sensitive to whether each and every HTTP tag is closed by "\r\n" (and not by a single "\n" character). Otherwise, the HTTP packet may be treated as a defective one and can be discarded.

Additional discussion on parameters of the service dispatcher as well as the trouble shooting tips can be found here.

C++ Connection Streams

Using connections and connectors (via the entirely procedural approach) in C++ programs overkills the power of the language. Therefore, we provide C++ users with the stream classes, all derived from a standard iostream class, and as a result, these can be used with all common stream I/O operators, manipulators, etc.

The declarations of the stream's constructors can be found in connect/ncbi_conn_stream.hpp. We tried to keep the same number and order of the constructor's parameters, as they appear in the corresponding connector's constructors in C.

The code below is a C++ style example from the previous section devoted to the service connector:

```
#include <connect/ncbi_conn_stream.hpp>
try {
 CConn_ServiceStream
 ios("io_bounce", fSERV_StatelessOnly, 0);
 ios << "Data to be passed to the server";
} STD_CATCH_ALL ("Connection problem");
```

Note: The stream constructor may show an exception if, for instance, the requested service is not found, or some other kind of problem arose. To see the actual reason, we used a standard toolkit macro STD_CATCH_ALL(), which prints the message and problem description into the log file (cerr, by default).

Service mapping API

The API defined in connect/ncbi_service.h is designed to map the required service name into the server address. Internally, the mapping is done either directly or indirectly by means of the load-balancing daemon, running at the NCBI site. For the client, the mapping is seen as reading from an iterator created by a call to SERV_Open(), similar to the following fragment (for more examples, please refer to the test program test_ncbi_disp.c):

```
#include <connect/ncbi_service.h>
SERV_ITER iter = SERV_Open("my_service", fSERV_Any, SERV_ANYHOST, 0);
int n = 0;
if (iter != 0) {
 const SSERV_Info * info;
 while ((info = SERV_GetNextInfo(iter)) != 0) {
 char* str = SERV_WriteInfo(info);
 printf("Server = `%s'\n", str);
 free(str);
 n++;
 }
 SERV_Close(iter);
}
if (!iter || !n)
 printf("Service not found\n");
```

Note: Services can be redirected.

Note: A non-NULL iterator returned from SERV_Open() **does not** yet guarantee that the service is available, whereas the NULL iterator definitely means that the service does not exist.

As shown in the above example, a loop over reading from the iterator results in the sequence of successive structures (none of which is to be freed by the program!) that along with the conversion functions SERV_ReadInfo(), SERV_WriteInfo() and others are defined in connect/ncbi_server_info.h. Structure SSERV_Info describes a server that implements the requested service. NULL gets returned when no more servers (if any) could be found. The iterator returns servers in the order that the load-balancing algorithm arranges them. Each server has a rating, and the larger the rating the better the chance for the server to be coming out first (but not necessarily in the order of the rates).

Note: Servers returned from the iterator are all of the requested type, with only one exception: they can include servers of type fSERV_Firewall, even if this type has not been explicitly requested. Therefore, the application must sort these servers out. But if fSERV_Firewall is set in the types, then the search is done for whichever else types are requested, and with the assumption that the client has chosen a firewall connection mode, regardless of the network parameters supplied in SConnNetInfo or read out from either the registry or environment.

Note: A search for servers of type fSERV_Dns is not inclusive with fSERV_Any specified as a server type. That is, servers of type DNS are only returned if specifically requested in the server mask at the time the iterator was opened.

There is a simplified version of SERV_Open(), called SERV_OpenSimple(), as well as an advanced version, called SERV_OpenEx(). The former takes only one argument, the service name. The latter takes two more arguments, which describe the set of servers **not** to be returned from the iterator (server descriptors that to be excluded).

There is also an advanced version of SERV_GetNextInfo(), called SERV_GetNextInfoEx(), that, via its second argument, provides the ability to get many host parameters, among which is so-called host environment; a "\0"-terminated string, consisting of a set of lines separated by "\n" characters and specified in the configuration file of the load-balancing daemon of the host, where the returned server has been found. The typical line within the set has a form "name=value" and resembles very much the shell environment, where its name comes after. The host environment could be very handy for passing additional information to applications if the host has some limitations or requires special handling, should the server be selected and used on this host. The example below shall give an idea. At the time of writing, getting the host information is only implemented when the service is obtained via direct access to the load-balancing daemon. Unlike returned server descriptors, the returned host information handle is not a constant object and must be explicitly freed by the application when no longer needed. All operations (getter methods) that are defined on the host information handle are declared in connect/ncbi_host_info.h. If the server descriptor was obtained using dispatching CGI (indirect dispatching, see below), then the host information handle is always returned as NULL (no host information available).

The back end of the service mapping API is split into two independent parts: *direct* access to LBSMD, if the one is both available on the current host and is not disabled by parameter lb_disable at the iterator opening. If LBSMD is either unavailable or disabled, the second (*indirect*) part of the back-end API is used, which involves a connection to the dispatching CGI, which in turn connects to LBSMD to carry out the request. An attempt to use the CGI is done only if the net_info argument is provided as non-NULL in the calls to SERV_Open() or SERV_OpenEx().

Note: In a call to SERV_OpenSimple(), net_info gets created internally before an upcall to SERV_Open() and thus CGI dispatching is likely to happen, unless either net_info could not be constructed from the environment, or the environment variable CONN_LB_DISABLE (or a service-specific one, or either of the corresponding registry keys) is set to TRUE.

Note: In the above conditions, the network service name resolution is also undertaken if the service name could not be resolved (because the service could not be found or because of some other error) with the use of locally running LBSMD.

The following code example uses both a service connector and the service mapping API to access certain services using an alternate way (other than the connector's default) of choosing appropriate servers. By default, the service connector opens an internal service iterator and then tries to connect to the next server, which SERV_GetNextInfo() returns when given the iterator. That is, the server with a higher rate is tried first. If user provides a pointer to structure SSERVICE_Extra as the last parameter of the connector's constructor, then the user-supplied routine (if any) can be called instead to obtain the next server. The routine is also given a supplemental custom argument data taken from SSERVICE_Extra. The (intentionally simplified) example below tries to create a connector to an imaginary service "my_service" with the restriction that the server has to additionally have a certain (user-specified) database present. The database name is taken from the LBSMD host environment keyed by "my_service_env", the first word of which is assumed to be the required database name.

```
#include <connect/ncbi_service_connector.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>


#define ENV_DB_KEY "my_service_env="
```

```
/* This routine gets called when connector is about to be destructed. */
static void s_CleanupData(void* data)
{
 free(data); /* we kept database name there */
}

/* This routine gets called on each internal close of the connector
 * (which may be followed by a subsequent internal open).
 */
static void s_Reset(void* data)
{
 /* just see that reset happens by printing DB name */
 printf("Connection reset, DB: %s\n", (char*) data);
}

/* 'Data' is what we supplied among extra-parameters in connector's
 * constructor.
 * 'Iter' is an internal service iterator used by service connector;
 * it must remain open.
 */
static const SSERV_Info* s_GetNextInfo(void* data, SERV_ITER iter)
{
 const char* db_name = (const char*) data;
 size_t len = strlen(db_name);
 SSERV_Info* info;
 HOST_INFO hinfo;
 int reset = 0;
 for (;;) {
 while ((info = SERV_GetNextInfoEx(iter, &hinfo)) != 0) {
 const char* env = HINFO_Environment(hinfo);
 const char* c;
 for (c = env; c; c = strchr(c, '\n')) {
 if (strncmp(c == env ? c : ++c, ENV_DB_KEY,
 sizeof(ENV_DB_KEY)-1) == 0) {
 /* Our keyword has been detected in environment */
 /* for this host */
 c += sizeof(ENV_DB_KEY) - 1;
 while (*c && isspace(*c))
 c++;
 if (strncmp(c, db_name, len) == 0 && !isalnum(c + len)) {
 /* Database name match */
 free(hinfo); /* must be freed explicitly */
 return info;
 }
 }
 }
 if (hinfo)
 free(hinfo); /* must be freed explicitly */
 }
 if (reset)
 break; /* coming to reset 2 times in a row means no server fit */
 SERV_Reset(iter);
 reset = 1;
 }
 return 0; /* no match found */
}

int main(int argc, char* argv[])
{
 char* db_name = strdup(argv[1]);
 SSERVICE_Extra params;
 CONNECTOR c;
 CONN conn;
 memset(&params, 0, sizeof(params));
 params.data = db_name; /* custom data, anything */
 params.reset = s_Reset; /* reset routine, may be NULL */
 params.cleanup = s_CleanupData; /* cleanup routine, may be NULL*/
 params.get_next_info = s_GetNextInfo; /* custom iterator routine */
 if (!(c = SERVICE_CreateConnectorEx("my_service",
 fSERV_Any, NULL, &params))) {
 fprintf(stderr, "Cannot create connector");
 exit(1);
 }
 if (CONN_Create(c, &conn) != eIO_Success) {
 fprintf(stderr, "Cannot create connection");
 exit(1);
 }
 /* Now we can use CONN_Read(),CONN_Write() etc to deal with
 * connection, and we are assured that the connection is made
 * only to the server on such a host which has "db_name"
 * specified in configuration file of LBSMD.
 */
 ...
 CONN_Close(conn);
 /* this also calls cleanup of user data provided in params */
 return 0;
}
```

Note: No network (indirect) mapping occurs in the above example because net_info is passed as NULL to the connector's constructor.

Local specification of the LBSM table

The LBSM table can also be specified locally, in config file and/or environment variables.

Service lookup process now involves looking up through the following sources, in this order:

- Local environment/registry settings;
- LBSM table (only in-house; this step does not exist in the outside builds);
- Network dispatcher.

Only one source containing the information about the service is used; the next source is only tried if the previous one did not yield in any servers (for the service).

Step 1 is disabled by default, to enable it set CONN_LOCAL_ENABLE environment variable to "1" (or "ON, or "YES", or "TRUE") or add LOCAL_ENABLE=1 to [CONN] section in .ini file. Steps 2 and 3 are enabled by default. To disable them use CONN_LBSMD_DISABLE and/or CONN_DISPD_DISABLE set to "1" in the environment or LBSMD_DISABLE=1 and/or DISPD_DISABLE=1 under the section "[CONN]" in the registry, respectively.

Note: Alternatively, and for the sake of backward compatibility with older application, the use of local LBSM table can be controlled by CONN_LB_DISABLE={0,1} in the environment or LB_DISABLE={0,1} in the "[CONN]" section of the registry, or individually on a per service basis:

For a service "ANAME", ANAME_CONN_LB_DISABLE={0,1} in the environment, or CONN_LB_DISABLE={0,1} in the section "[ANAME]" in the registry (to affect setting of this particular service, and no others).

The local environment / registry settings for service "ANAME" are screened in the following order:

- A value of environment variable "ANAME_CONN_LOCAL_SERVER_n";
- A value of registry key "CONN_LOCAL_SERVER_n" in the registry section "[ANAME]"

Note that service names are not case sensitive, yet the environment variables are looked up all capitalized.

An entry found in the environment takes precedence over an entry (for the same "n") found in the registry. "n" counts from 0 to 100, and need not to be sequential.

All service entries can be (optionally) grouped together in a list as a value of either:

- Environment variable "CONN_LOCAL_SERVICES", or
- Registry key "LOCAL_SERVICES" under the registry section "[CONN]".

The list of local services is only used in cases of wildcard searches, or in cases of reverse lookups, and is never consulted in regular cases of forward searches by a complete service name.

Examples:

1. In .ini file

```
[CONN]
LOCAL_ENABLE=yes
LOCAL_SERVICES="MSSQL10 MSSQL14 MSSQL15 MSSQL16 MSSQL17"

[MSSQL10]
CONN_LOCAL_SERVER_6="DNS mssql10:1433 L=yes"

[MSSQL15]
CONN_LOCAL_SERVER_9="DNS mssql15:1433 L=yes"
```

Note that entries for MSSQL14, 16, and 17 are not shown, and they are not required (inexistent definitions for declared services are simply ignored).

2. In environment set the following variables (equivalent to the .ini fragment above but having a higher precedence):

```
CONN_LOCAL_ENABLE=yes
CONN_LOCAL_SERVICES="MSSQL10 MSSQL14 MSSQL15 MSSQL16 MSSQL17"
MSSQL10_CONN_LOCAL_SERVER_6="DNS mssql10:1433 L=yes"
MSSQL15_CONN_LOCAL_SERVER_9="DNS mssql15:1433 L=yes"
```

You can also look at the detailed description of LBSMD and a sample configuration file.

Threaded Server Support

This library also provides CServer, an abstract base class for multithreaded network servers. Here is a demonstration of its use. For more information, see the Implementing a Server with CServer section.

10: Database Access - SQL, Berkley DB

Last Update: July 31, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Database Access [Library dbapi: include | src]

The DBAPI library provides the underlying user-layer and driver API for the NCBI database connectivity project. The project's goal is to provide access to various relational database management systems (RDBMS) with a single uniform user interface. Consult the detailed documentation for details of the supported DBAPI drivers.

The BDB library is part of the NCBI C++ Toolkit and serves as a high-level interface to the Berkeley DB. The primary purpose of the library is to provide tools for work with flatfile, federated databases. The BDB library incorporates a number of Berkeley DB services; for a detailed understanding of how it works, study the original Berkeley DB documentation from http://www.oracle.com/database/berkeley-db/db/. The BDB library is compatible with Berkeley DB v. 4.1 and higher. The BDB library, as it is right now, is architecturally different from the dbapi library and does not follow its design. The BDB is intended for use by software developers who need small-footprint, high-performance database capabilities with zero administration. The database in this case becomes tightly integrated with the application code.

Chapter Outline

The following is an outline of the topics presented in this chapter:

DBAPI Overview

DBAPI is a consistent, object-oriented programming interface to multiple back-end databases. It encapsulates leading relational database vendors' APIs and is universal for all applications regardless of which database is used. It frees developers from dealing with the low-level details of a particular database vendor's API, allowing them to concentrate on domain-specific issues and build appropriate data models. It allows developers to write programs that are reusable with many different types of relational databases and to drill down to the native database APIs for added control when needed.

DBAPI has open SQL interface. It takes advantage of database-specific features to maximize performance and allows tight control over statements and their binding and execution semantics.

DBAPI has "Native" Access Modules for Sybase, Microsoft SQL Server, SQLITE, and ODBC. It provides native, high-performance implementations for supported vendor databases. It allows porting to other databases with minimal code changes.

DBAPI is split into low-layer and user-layer.

In addition, a simplified C++ API (SDBAPI) layer is provided for cases where the full DBAPI feature set is not required.

See the DBAPI configuration parameters reference for details on configuring the DBAPI library.

See the DBAPI sample programs for example code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/

Security

Preventing SQL Injection

Much has been written about networked database security - in particular, SQL injection. Please see the common resources for more information.

When using DBAPI or SDBAPI, the two most important rules for protecting against SQL injection are:

1. Never construct a SQL statement from user-supplied input if the same functionality can be achieved by passing the user input to stored procedures or parameterized SQL.
2. If constructing a SQL statement from user-supplied input cannot be avoided, then you MUST sanitize the user input.

The following sample programs illustrates how to protect against SQL injection for basic SQL statements using SDBAPI and DBAPI:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/sdbapi/sdbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp

See the Security FAQ for more information.

Using Kerberos with DBAPI

Individual users (i.e. not service accounts) within NCBI can use Kerberos with DBAPI, provided the following conditions are met:

1. The database must allow them to connect using Kerberos. (Email dbhelp@ncbi.nlm.nih.gov if you need help with this.)
2. DBAPI must be configured to enable Kerberos.
   a. Either the NCBI_CONFIG__DBAPI__CAN_USE_KERBEROS environment variable must be set to true; or
   b. the can_use_kerberos entry in the dbapi section of the application configuration file must be set to true.

3. Their Kerberos ticket must not be expired.
4. They must pass an empty string for the user name.

This is also covered in the DBAPI section of the Library Configuration chapter.

Simple Database Access via C++

This section shows how to execute a simple static SQL query using the simplified database API (SDBAPI). Note that database load-balancing is performed automatically and transparenty when using SDBAPI.

C++ source files using SDBAPI should contain:

```
#include <dbapi/simple/sdbapi.hpp>
```

Application makefiles should contain:

```
LIB = $(SDBAPI_LIB) xconnect xutil xncbi
LIBS = $(SDBAPI_LIBS) $(NETWORK_LIBS) $(DL_LIBS) $(ORIG_LIBS)
```

This example connects to a load-balanced service:

```
CDatabase db("dbapi://" + my_username + ":" + my_password +
 "@" + my_servicename + "/" + my_dbname);
db.Connect();
CQuery query = db.NewQuery();
query.SetSql("select title from Journal");
query.Execute();
ITERATE(CQuery, it, query.MultiSet()) {
 string col1 = it[1].AsString(); // Note: uses 1-based index !
 NcbiCout << col1 << NcbiEndl;
}
```

Note: SDBAPI always uses load balancing - you don't have to call DBLB_INSTALL_DEFAULT().

See the SDBAPI sample programs for more example code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/sdbapi/

Database Load-Balancing (DBLB)

Setting up Load-Balancing of Database Servers

For the following to be clear, it is important to distinguish between a database name, an underlying (actual) server name (e.g. MSSQL17), which hosts a variety of databases, a database server alias, and a service name. A server alias may be moved to a different underlying server. The server alias is often used with sqsh, and the GUI tools, such as SQL Management studio. The service name is used by the load-balancer to look up the underlying server to use, and is the name that should be used by an application. The server aliases and service names often share a common prefix and would look similar, and in fact for reasons presented below, there should be at least one server alias that is identical to the service name.

The following steps must be done prior to database load-balancing:

1. Ask the DBAs to add your service name (e.g. YOURSERVICE) to the load-balancer configuration database. Typically, the names are clear, for example, there are server aliases YOURSERVICE1, and YOURSERVICE2 that already exist, and databases that have “YOURSERVICE” as an embedded string, but if not, the databases providing the service and the server aliases involved should be given. Note that if databases are moved to different underlying servers, both the server aliases, and the load-balancer configuration which points to those servers are both moved, synchronously.
2. Tell the DBAs which of the server aliases point to the server that should be used, if the load-balancer is unavailable, as the DBAPI will look for a server alias with the same name as the service, in that case.
3. The DBAs will also ask for a DNS name to match the service name as a backup connection method, should everything else fail.

Using Database Load-Balancing from C++

For simplest access, see the section on using SDBAPI above. SDBAPI uses the database load-balancing by default.

If more flexibility is required, and you want to activate the database load-balancing for the more general NCBI DBAPI:

1. Before the very first DBAPI connection attempt, call:

   #include <dbapi/driver/dbapi_svc_mapper.hpp>
   DBLB_INSTALL_DEFAULT();
2. Link '$(XCONNEXT)' and 'xconnect' libraries to your application.

If steps (1) and (2) above are done then the DBAPI connection methods (such as Connect() or ConnectValidated()) will attempt to resolve the passed server name as a load-balanced service name.

Note: If steps (1) and (2) above are not done, or if DBLB library is not available (such as in the publicly distributed code base), or if the passed server name cannot be resolved as a load-balanced service name, then the regular database server name resolution will be used – i.e. the passed name will first be interpreted as a server alias (using the "interfaces" file), and if that fails, it will be interpreted as a DNS name. Note however that by default if the service name resolves (exists), then the regular database server name resolution will not be used as a fallback, even if DBAPI can't connect (for whatever reason) to the servers that the service resolves to.

Example:

```
#include <dbapi/driver/dbapi_svc_mapper.hpp>

DBLB_INSTALL_DEFAULT();
IDataSource* ds = dm.CreateDs("ftds");
IConnection* conn = ds->CreateConnection();

// (Use of validator here is optional but generally encouraged.)
CTrivialConnValidator my_validator(my_databasename);

conn->ConnectValidated(my_validator, my_username, my_password,
my_servicename);
```

Load-Balanced Database Access via Python and Perl

Load-balanced database access can be achieved from scripts by using the utility ncbi_dblb, which is located under /opt/machine/lbsm/bin:

Here are some sample scripts that demonstrate using ncbi_dblb to retrieve the server name for a given load-balanced name:

From Python:

```
#!/usr/bin/env python

import subprocess, sys

if len(sys.argv) > 1:
 # Use the -q option to fetch only the server name.
 cmd = ['/opt/machine/lbsm/bin/ncbi_dblb', '-q', sys.argv[1]]
 srv = subprocess.Popen(cmd, stdout=subprocess.PIPE).communicate()[0].strip()
 # Do whatever is needed with the server name...
 print 'Server: "' + srv + '"'
```

From Perl:

```
#!/usr/bin/env perl -w

use strict;

if (@ARGV) {
 # Use the -q option to fetch only the server name.
 my $cmd = '/opt/machine/lbsm/bin/ncbi_dblb -q ' . $ARGV[0];
 my $srv = `$cmd`; chomp($srv);
 # Do whatever is needed with the server name...
 print 'Server: "' . $srv . '"'
}
```

Advantages of using DBLB

C++ Specific

- A database-level verification mechanism.
- Latch onto the same database server for the life of your process. It's often useful to avoid possible inter-server data discrepancy. The "latch-on" mechanism can be relaxed or turned off if needed.
- Automatic connection retries. If a connection to the selected server cannot be established, the API will try again with other servers (unless it is against the chosen "latch-on" strategy).
- The default connection strategy is *configurable*. You can change its parameters using a configuration file, environment variables, and/or programmatically. You can also configure locally for your application ad-hoc mappings to the database servers (this is usually not recommended but can come in handy in emergency cases or for debugging).
- If needed, you can implement your own customized mapper. Components of the default connection strategy can be used separately, or in combination with each other and with the user-created strategies, if necessary.

General

- Connecting to the database servers by server name and/or "interfaces" file based aliases still works the same as it used to.
- Automatic avoidance of unresponsive database servers. This prevents your application from hanging for up to 30 seconds on the network timeout.
- Independence from the database "interfaces" file. A centrally maintained service directory is used instead, which is accessible locally and/or via network. It also dynamically checks database servers' availability and excludes unresponsive servers.

How it works (by default)

The following steps are performed each time a request is made to establish a load-balanced connection to a named database service:

1. The requests will first go through the DBLB mechanism that tries to match the requested service name against the services known to the NCBI Load Balancer and/or those described in the application's configuration file.
2. If the requested service name is unknown to the load balancer then this name will be used "as is".
3. However, if this service name is known to the DBLB then the DBLB will try to establish a connection to the database server that it deems the most suitable. If the service is handled by the NCBI load-balancer, then the unresponsive servers will be weeded out, and a load on the machines that run the servers may be taken into account too.
4. C++ only: If the connection cannot be established, then DBLB will automatically retry the connection, now using another suitable database server.
5. This procedure may be repeated several times, during which there will be only one attempt to connect to each database.
6. C++ only: Once a database connection is successfully established it will be "latched-on". This means that when you will try to connect to the same service or alias within the same application again then you will be connected to the same database server (this can be relaxed or turned off completely).
7. For example, you can connect to the "PMC" service which is currently mapped to two servers. The server names are provided dynamically by the NCBI load-balancer, so you never have to change your configuration or recompile your application if either a service configuration or an "interfaces" file get changed.
8. C++ only: If ConnectValidated() is used to connect to a database, then requests to establish database connections will first go through the server-level load-balancing mechanism. On successful login to server, the database connection will be validated against the validator. If the validator does not "approve" the connection, then DBAPI will automatically close this connection and repeat this login/validate attempt with the next server, and so on, until a "good" (successful login + successful validation) connection is found. If you want to validate a connection against more than one validator/database, then you can combine validators. Class CConnValidatorCoR was developed to allow combining of other validators into a chain.

NCBI DBAPI User-Layer Reference

Object hierarchy

See Figure 1.

Includes

For most purposes it is sufficient to include one file in the user source file: dbapi.hpp.

```
#include <dbapi/dbapi.hpp>
```

For static linkage the following include file is also necessary:

```
#include <dbapi/driver/drivers.hpp>
```

Objects

All objects are returned as pointers to their respective interfaces. The null (0) value is valid, meaning that no object was returned.

Object Life Cycle

In general, any child object is valid only in the scope of its parent object. This is because most of the objects share the same internal structures. There is no need to delete every object explicitly, as all created objects will be deleted upon program exit. Specifically, all objects are derived from the static CDriverManager object, and will be destroyed when CDriverManager is destroyed. It is possible to delete any object from the framework and it is deleted along with all derived objects. For example, when an IConnection object is deleted, all derived IStatement, ICallableStatement and IResultSet objects will be deleted too. If the number of the objects (for instance IResultSet) is very high, it is recommended to delete them explicitly or enclose in the auto_ptr<...> template. For each object a Close() method is provided. It disposes of internal resources, required for the proper library cleanup, but leaves the framework intact. After calling Close() the object becomes invalid. This method may be necessary when the database cleanup and framework cleanup are performed in different places of the code.

CVariant Type

The CVariant type is used to represent any database data type (except BLOBs). It is an object, not a pointer, so it behaves like a primitive C++ type. Basic comparison operators are supported (==, !=, <) for identical internal types. If types are not identical, CVariantException is thrown. CVariant has a set of getters to extract a value of a particular type, e.g. GetInt4(), GetByte(), GetString(), etc. If GetString() is called for a different type, like DateTime or integer it tries to convert it to a string. If it doesn't succeed, CVariantException is thrown. There is a set of factory methods (static functions) for creating CVariant objects of a particular type, such as CVariant::BigInt(), CVariant::SmallDateTime(), CVariant::VarBinary() etc. For more details please see the comments in variant.hpp file.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_object.cpp

Choosing the Driver

There are several drivers for working with different SQL servers on different platforms. The ones presently implemented are "ctlib" (Sybase), "dblib"(MS SQL, Sybase), "ftds" (MS SQL, Sybase, cross platform). For static linkage these drivers should be registered manually; for dynamic linkage this is not necessary. The CDriverManager object maintains all registered drivers.

```
DBAPI_RegisterDriver_CTLIB();
DBAPI_RegisterDriver_DBLIB();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_context.cpp

Data Source and Connections

The IDataSource interface defines the database platform. To create an object implementing this interface, use the method CreateDs(const string& driver). An IDataSource can create objects represented by an IConnection interface, which is responsible for the connection to the database. It is highly recommended to specify the database name as an argument to the CreateConnection() method, or use the SetDatabase() method of a CConnection object instead of using a regular SQL statement. In the latter case, the library won't be able to track the current database.

```
IDataSource *ds = dm.CreateDs("ctlib");
IConnection *conn = ds->CreateConnection();
conn->Connect("user", "password", "server", "database");
IStatement *stmt = conn->CreateStatement();
```

Every additional call to IConnection::CreateStatement() results in cloning the connection for each statement. These connections inherit the same default database, which was specified in the Connect() or SetDatabase() method. Thus if the default database was changed by calling SetDatabase(), all subsequent cloned connections created by CreateStatement() will inherit this particular default database.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_connection.cpp

Main Loop

The library simulates the main result-retrieving loop of the Sybase client library by using the IStatement::HasMoreResults() method:

```
stmt->Execute("select * from MyTable");
while( stmt->HasMoreResults() ) {
 if( stmt->HasRows() ) {
 IResultSet *rs = stmt->GetResultset();

 // Retrieve results, if any

 while( rs->Next() ) {
 int col1 = rs->GetVariant(1).GetInt4();
 ...
 }
 }
}
```

This method should be called until it returns false, which means that no more results are available. It returns as soon as a result is ready. The type of the result can be obtained by calling the IResultSet::GetResultType() method. Supported result types are eDB_RowResult, eDB_ParamResult, eDB_ComputeResult, eDB_StatusResult, eDB_CursorResult. The method IStatement::GetRowCount() returns the number of updated or deleted rows.

The IStatement::ExecuteUpdate() method is used for SQL statements that do not return rows:

```
stmt->ExecuteUpdate("update...");
int rows = stmt->GetRowCount();
```

The method IStatement::GetResultSet() returns an IResultSet object. The method IResultSet::Next() actually does the fetching, so it should be always called first. It returns false when no more fetch data is available. All column data, except Image and Text is represented by a single CVariant object. The method IResultSet::GetVariant() takes one parameter, the column number. Column numbers start with 1.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_stmt.cpp

Input and Output Parameters

The method ICallableStatement::SetParam(const CVariant& v, const string& name) is used to pass parameters to stored procedures and dynamic SQL statements. To ensure the correct parameter type it is recommended to use CVariant type factories (static methods) to create a CVariant of the required internal type. There is no internal representation for the BIT parameter type, please use TinyInt of Int types with 0 for false and 1 for true respectively. Here are a few examples: CVariant::Int4(Int4 *p), CVariant::TinyInt(UInt1 *p), CVariant::VarChar(const char *p, size_t len) etc.

There are also corresponding constructors, like CVariant::CVariant(Int4 v), CVariant::CVariant(const string& s), ..., but the user must ensure the proper type conversion in the arguments, and not all internal types can be created using constructors.

Output parameters are set by the ICallableStatement::SetOutputParam(const CVariant& v, const string& name) method, where the first argument is a null CVariant of a particular type, e.g. SetOutputParam(CVariant(eDB_SmallInt),"@arg").

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp

Stored Procedures

The ICallableStatement object is used for calling stored procedures. First get the object itself by calling IConnection::PrepareCall(). Then set any parameters. If the parameter name is empty, the calls to SetParam() should be in the exact order of the actual parameters. Retrieve all results in the main loop. Get the status of the stored procedure using the ICallableStatement::GetReturnStatus() method.

```
ICallableStatement *cstmt = conn->PrepareCall("ProcName");
Uint1 byte = 1;
cstmt->SetParam(CVariant("test"), "@test_input");
cstmt->SetParam(CVariant::TinyInt(&byte), "@byte");
```

```
cstmt->SetOutputParam(CVariant(eDB_Int), "@result");
cstmt->Execute();
while(cstmt->HasMoreResults()) {
 if( cstmt->HasRows() ) {
 IResultSet *rs = cstmt->GetResultSet();
 switch( rs->GetResultType() ) {
 case eDB_RowResult:
 while(rs->Next()) {

 // retrieve row results

 }
 break;
 case eDB_ParamResult:
 while(rs->Next()) {

 // Retrieve parameter row

 }
 break;
 }
 }
}

// Get status
int status = cstmt->GetReturnStatus();
```

It is also possible to use IStatement interface to call stored procedures using standard SQL language call. The difference from ICallableStatement is that there is no SetOutputParam() call. The output parameter is passed with a regular SetParam() call having a *non-null* CVariant argument. There is no GetReturnStatus() call in IStatement, so use the result type filter to get it - although note that result sets with type eDB_StatusResult are not always guaranteed to be returned when using the IStatement interface.

```
sql = "exec SampleProc @id, @f, @o output";
stmt->SetParam(CVariant(5), "@id");
stmt->SetParam(CVariant::Float(&f), "@f");
stmt->SetParam(CVariant(5), "@o");
stmt->Execute(sql);
while(stmt->HasMoreResults()) {
 IResultSet *rs = stmt->GetResultSet();

 if( rs == 0 )
 continue;

 switch( rs->GetResultType() ) {
 case eDB_ParamResult:
 while( rs->Next() ) {
 NcbiCout << "Output param: "
 << rs->GetVariant(1).GetInt4()
 << NcbiEndl;
 }
 break;
 case eDB_StatusResult:
 while( rs->Next() ) {
 NcbiCout << "Return status: "
 << rs->GetVariant(1).GetInt4()
 << NcbiEndl;
 }
 break;
 case eDB_RowResult:
 while( rs->Next() ) {
 if( rs->GetVariant(1).GetInt4() == 2121 ) {
 NcbiCout << rs->GetVariant(2).GetString() << "|"
 << rs->GetVariant(3).GetString() << "|"
 << rs->GetVariant(4).GetString() << "|"
 << rs->GetVariant(5).GetString() << "|"
 << rs->GetVariant(6).GetString() << "|"
 << rs->GetVariant(7).GetString() << "|"
 << NcbiEndl;
 } else {
 NcbiCout << rs->GetVariant(1).GetInt4() << "|"
 << rs->GetVariant(2).GetFloat() << "|"
 << rs->GetVariant("date_val").GetString() << "|"
 << NcbiEndl;
 }
 }
 break;
 }
}
stmt->ClearParamList();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_proc.cpp

Cursors

The library currently supports basic cursor features such as setting parameters and cursor update and delete operations.

```
ICursor *cur = conn->CreateCursor("table_cur",
 "select ... for update of ...");
IResultSet *rs = cur->Open();
while(rs->Next()) {
 cur->Update(table, sql_statement_for_update);
}
cur->Close();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_cursor.cpp

Working with BLOBs

Due to the possibly very large size, reading and writing BLOBs requires special treatment. During the fetch the contents of the whole column must be read before advancing to the next one. That's why the columns of type IMAGE and TEXT are not bound to the corresponding variables in the resultset and all subsequent columns are not bound either. So it is recommended to put the BLOB columns at the end of the column list. There are several ways to read BLOBs, using IResultSet::Read(), IResultSet::GetBlobIStream(), and IResultSet::GetBlobReader() methods. The first is the most efficient; it reads data into a supplied buffer until it returns 0 bytes read. The next call will read from the next column. The second method implements the STL istream interface. After each successful column read you should get another istream for the next column. The third implements the C++ Toolkit IReader interface. If the data size is small and double buffering is not a performance issue, the BLOB columns can be bound to CVariant variables by calling IResultSet::BindBlobToVariant(true). In this case the data should be read using CVariant::Read() and CVariant::GetBlobSize(). To write BLOBs there are also several options. To pass a BLOB as a SQL parameter you should store it in a CVariant using CVariant::Append() and CVariant::Truncate() methods. To store a BLOB in the database you should initialize this column first by writing a zero value (0x0) for an IMAGE type or a space value (' ') for a TEXT type. After that you can open a regular IResultSet or ICursor and for each required row update the BLOB using IResultSet::GetBlobOStream(). NOTE: this call opens an additional connection to the database.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_lob.cpp

Updating BLOBs Using Cursors

It is recommended to update BLOBs using cursors, because no additional connections are opened and this is the only way to work with ODBC drivers.

```
ICursor *blobCur = conn->CreateCursor("test",
 "select id, blob from BlobSample for update of blob");
IResultSet *blobRs = blobCur->Open();
while(blobRs->Next()) {
 ostream& out = blobCur->GetBlobOStream(2, blob.size());
 out.write(buf, blob.size());
 out.flush();
}
```

Note that GetBlobOStream() takes the column number as the first argument and this call is invalid until the cursor is open.

Using Bulk Insert

Bulk insert is useful when it is necessary to insert big amounts of data. The IConnection::CreateBulkInsert() takes one parameter, the table name. The number of columns is determined by the number of Bind() calls. The CVariant::Truncate(size_t len) method truncates the internal buffer of CDB_Text and CDB_Image object from the end of the buffer. If no parameter specified, it erases the whole buffer.

```
NcbiCout << "Initializing BlobSample table..." << NcbiEndl;
IBulkInsert *bi = conn->CreateBulkInsert(tbl_name);
CVariant col1 = CVariant(eDB_Int);
CVariant col2 = CVariant(eDB_Text);
bi->Bind(1, &col1);
bi->Bind(2, &col2);
for(int i = 0; i < ROWCOUNT; ++i ) {
 string im = "BLOB data " + NStr::IntToString(i);
 col1 = i;
 col2.Truncate();
 col2.Append(im.c_str(), im.size());
 bi->AddRow();
}
bi->Complete();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_bcp.cpp

Diagnostic Messages

The DBAPI library is integrated with the C++ Toolkit diagnostic and tracing facility. By default all client and server messages are handled by the Toolkit's standard message handler. However it is possible to redirect the DBAPI-specific messages to a single CDB_MultiEx object and retrieve them later at any time. There are two types of redirection, per data source and per connection. The redirection from a data source is enabled by calling IDataSource::SetLogStream(0). After the call all client- and context-specific messages will be stored in the IDataSource object. The IDataSource::GetErrorInfo() method will return the string representation of all accumulated messages and clean up the storage. The IDataSource::GetErrorAsEx() will return a pointer to the underlying CDB_MultiEx object. Retrieving information and cleaning up is left to the developer. Do NOT delete this object. The connection-specific redirection is controlled by calling IConnection::MsgToEx(boolean enable) method. This redirection is useful; for instance, to temporarily disable default messages from the database server. The IConnection::GetErrorInfo() and IConnection::GetErrorAsEx() methods work in the same manner as for the IDataSource

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_msg.cpp

Trace Output

The DBAPI library uses the Toolkit-wide DIAG_TRACE environment variable to do the debug output. To enable it set it to any value. If you have any problems with the DBAPI please include the trace output into your email.

NCBI DBAPI Driver Reference

(Low-level access to the various RDBMSs.)

Overview

The NCBI DBAPI driver library describes and implements a set of objects needed to provide a uniform low-level access to the various relational database management systems (RDBMS). The basic driver functionality is the same as in most other RDBMS client APIs. It allows opening a connection to a server, executing a command (query) on this connection and getting the results back. The main advantage of using the driver is that you don't have to change your own upper-level code if you need to move from one RDBMS client API to another.

The driver can use two different methods to access the particular RDBMS. If the RDBMS provides a client library for the given computer system (e.g. Sun/Solaris), then the driver uses that library. If no such client library exists, then the driver connects to an RDBMS through a special gateway server which is running on a computer system where such a library does exist.

The driver architecture

There are two major groups of the driver's objects: the RDBMS-independent objects, and the objects which are specific to a particular RDBMS. The only RDBMS-specific object which user should be aware of is a "Driver Context". The "Driver Context" is effectively a "Connection" factory. The only way to make a connection to the server is to call the Connect () method of a "Driver Context" object. So, before doing anything with an RDBMS, you need to create at least one driver context object. All driver contexts implement the same interface defined in I_DriverContext class. If you are working on a library which could be used with more than one RDBMS, the driver context should not be created by the library. Instead, the library API should include a pointer to I_DriverContext so an existing driver context can be passed in.

There is no "real" factory for driver contexts because it's not always possible to statically link the RDBMS libraries from different vendors into the same binary. Most of them are written in C and name collisions do exist. The Driver Manager helps to overcome this problem. It allows creating a mixture of statically linked and dynamically loaded drivers and using them together in one executable.

The driver context creates the connection which is RDBMS-specific, but before returning it to the caller it puts it into an RDBMS-independent "envelope", CDB_Connection. The same is true for the commands and for the results - the user gets the pointer to the RDBMS-independent "envelope object" instead of the real one. It is the caller's responsibility to delete those objects. The life spans of the real object and the envelope object are not necessarily the same.

Once you have the connection object, you could use it as a factory for the different types of commands. The command object in turn serves as a factory for the results. The connection is always single threaded, that means that you have to execute the commands and process their results sequentially one by one. If you need to execute the several commands in parallel, you can do it using multiple connections.

Another important part of the driver is error and message handling. There are two different mechanisms implemented. The first one is exceptions. All exceptions which could be thrown by the driver are inherited from the single base class CDB_Exception. Drivers use the exception mechanism whenever possible, but in many cases the underlying client library uses callbacks or handlers to report error messages rather than throwing exceptions. The driver supplies a handler's stack mechanism to manage these cases.

To send and to receive the data through the driver you have to use the driver provided datatypes. The collection of the datatypes includes: one, two, four and eight byte integers; float and double; numeric; char, varchar, binary, varbinary; datetime and smalldatetime; text and image. All datatypes are derived from a single base class CDB_Object.

Sample program

This program opens one connection to the server and selects the database names and the date when each database was created (assuming that table "sysdatabases" does exist). In this example the string "XXX" should be replaced with the real driver name.

```
#include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
/* Here, XXXlib has to be replaced with the real name, e.g. "ctlib" */
#include <dbapi/driver/XXXlib/interfaces.hpp>
USING_NCBI_SCOPE;
int main()
{
 try { // to be sure that we are catching all driver related exceptions
 // We need to create a driver context first
 // In real program we have to replace CXXXContext with something real
 CXXXContext my_context;
 // connecting to server "MyServer"
 // with user name "my_user_name" and password "my_password"
 CDB_Connection* con = my_context.Connect("MyServer", "my_user_name",
 "my_password", 0);
 // Preparing a SQL query
 CDB_LangCmd* lcmd =
 con->LangCmd("select name, crdate from sysdatabases");
 // Sending this query to a server
 lcmd->Send();
 CDB_Char dbname(64);
 CDB_DateTime crdate;
 // the result loop
 while(lcmd->HasMoreResults()) {
 CDB_Result* r= lcmd->Result();
 // skip all but row result
 if (r == 0 || r->ResultType() != eDB_RowResult) {
 delete r;
 continue;
 }
 // printing the names of selected columns
 NcbiCout << r->ItemName(0) << " \t\t\t"
 << r->ItemName(1) << NcbiEndl;
 // fetching the rows
 while ( r->Fetch() ) {
 r->GetItem(&dbname); // get the database name
 r->GetItem(&crdate); // get the creation date
 NcbiCout << dbname.Value() << ' '
 << crdate.Value().AsString("M/D/Y h:m")
 << NcbiEndl;
 }
 delete r; // we don't need this result anymore
 }
 delete lcmd; // delete the command
 delete con; // delete the connection
 }
 catch (CDB_Exception& e) { // printing the error messages
 CDB_UserHandler_Stream myExHandler(&cerr);
 myExHandler.HandleIt(&e);
 }
}
```

Error handling

Error handling is almost always a pain when you are working with an RDBMS because different systems implement different approaches. Depending on the system, you can get error messages through return codes, callbacks, handlers, and/or exceptions. These messages could have different formats. It could be just an integer (error code), a structure, or a set of callback's arguments. The NCBI DBAPI driver intercepts all those error messages in all different formats and converts them into various types of objects derived from CDB_Exception.

CDB_Exception provides the following methods for all exceptions:

- GetDBErrCode() - returns the integer code for this message (assigned by SQL server).
- SeverityString(void) - returns the severity string of this message (assigned by SQL server).
- GetErrCodeString() - returns the name for this error code (e.g. "eSQL").
- Type() - returns the type value for this exception type (e.g. eSQL).
- TypeString() - returns the type string for this exception type (e.g. "eSQL"). This is a pass-through to CException::GetType().
- ErrCode() - alias for GetDBErrCode().
- Message() - returns the error message itself. This is a pass-through to CException::GetMsg().
- OriginatedFrom() - returns the SQL server name. This is a pass-through to CException::GetModule().
- SetServerName() - sets the SQL server name.
- GetServerName() - returns the SQL server name.
- SetUserName() - sets the SQL user name.
- GetUserName() - returns the SQL user name.
- SetExtraMsg() - sets extra message text to be included in the message output.
- GetExtraMsg() - gets the extra message text.
- SetSybaseSeverity() - sets the severity value for a Sybase exception - N.B. Sybase severity values can be provided for Sybase/FreeTDS ctlib/dblib drivers only.

- GetSybaseSeverity() - gets the severity value for a Sybase exception - N.B. Sybase severity values can be provided by Sybase/FreeTDS ctlib/dblib drivers only.
- ReportExtra() - outputs any extra text to the supplied stream.
- Clone() - creates a new exception based on this one.

N.B. The following CDB_Exception methods are deprecated:

- Severity() - returns the severity value of this message (assigned by SQL server).
- SeverityString(EDB_Severity sev) - returns the severity string of this message (assigned by SQL server).

The DBAPI driver may throw any of the following types derived from CDB_Exception:

- CDB_SQLEx This type is used if an error message has come from a SQL server and indicates an error in a SQL query. It could be a wrong table or column name or a SQL syntax error. This type provides the additional methods:
  - BatchLine() - returns the line number in the SQL batch that generated the error.
  - SqlState() - returns a byte string describing an error (it's not useful most of the time).
- CDB_RPCEx An error message has come while executing an RPC or stored procedure. This type provides the additional methods:
  - ProcName() - returns the procedure name where the exception originated.
  - ProcLine() - returns the line number within the procedure where the exception originated.
- CDB_DeadlockEx An error message has come as a result of a deadlock.
- CDB_DSEx An error has come from an RDBMS and is not related to a SQL query or RPC.
- CDB_TimeoutEx An error message has come due to a timeout.
- CDB_ClientEx An error message has come from the client side.

Drivers use two ways to deliver an error message object to an application. If it is possible to throw an exception, then the driver throws the error message object. If not, then the driver calls the user's error handler with a pointer to the error message object as an argument. It's not always convenient to process all types of error messages in one error handler. Some users may want to use a special error message handler inside some function or block and a default error handler outside. To accommodate these cases the driver provides a handler stack mechanism. The top handler in the stack gets the error message object first. If it knows how to deal with this message, then it processes the message and returns true. If handler wants to pass this message to the other handlers, then it returns false. So, the driver pushes the error message object through the stack until it gets true from the handler. The default driver's error handler, which just prints the error message to stderr, is always on the bottom of the stack.

Another tool which users may want to use for error handling is the CDB_MultiEx object. This tool allows collecting multiple CDB_Exception objects into one container and then throwing the container as one exception object.

Driver context and connections

Every program which is going to work with an NCBI DBAPI driver should create at least one Driver Context object first. The main purpose of this object is to be a Connection factory, but it's a good idea to customize this object prior to opening a connection. The first step is to setup two message handler stacks. The first one is for error messages which are not bound to some particular connection or could occur inside the Connect() method. Use PushCntxMsgHandler

() to populate it. The other stack serves as an initial message handler stack for all connections which will be derived from this context. Use PushDefConnMsgHandler() method to populate this stack. The second step of customization is setting timeouts. The SetLoginTimeout() and SetTimeout() methods do the job. If you are going to work with text or image objects in your program, you need to call SetMaxTextImageSize() to define the maximum size for such objects. Objects which exceed this limit could be truncated.

```
class CMyHandlerForConnectionBoundErrors : public CDB_UserHandler
{
 virtual bool HandleIt(CDB_Exception* ex);
 ...
};
class CMyHandlerForOtherErrors : public CDB_UserHandler
{
 virtual bool HandleIt(CDB_Exception* ex);
 ...
};
...
int main()
{
 CMyHandlerForConnectionBoundErrors conn_handler;
 CMyHandlerForOtherErrors other_handler;
 ...
 try { // to be sure that we are catching all driver related exceptions
 // We need to create a driver context first
 // In real program we have to replace CXXXContext with something real
 CXXXContext my_context;
 my_context.PushCntxMsgHandler(&other_handler);
 my_context.PushDefConnMsgHandler(&conn_handler);
 // set timeouts (in seconds) and size limits (in bytes):
 my_context.SetLoginTimeout(10); // for logins
 my_context.SetTimeout(15); // for client/server communications
 my_context.SetMaxTextImageSize(0x7FFFFFFF); // text/image size limit
 ...
 CDB_Connection* my_con =
 my_context.Connect("MyServer", "my_user_name", "my_password",
 I_DriverContext::fBcpIn);
 ...
 }
 catch (CDB_Exception& e) {
 other_handler.HandleIt(&e);
 }
}
```

The only way to get a connection to a server in an NCBI DBAPI driver is through a Connect () method in driver context. The first three arguments: server name, user name and password are obvious. Values for mode are constructed by a bitwise-inclusive-OR of flags defined in EConnectionMode. If reusable is false, then driver creates a new connection which will be destroyed as soon as user delete the correspondent CDB_Connection (the pool_name is ignored in this case).

Opening a connection to a server is an expensive operation. If program opens and closes connections to the same server multiple times it worth calling the Connect() method with reusable set to true. In this case driver does not close the connection when the correspondent CDB_Connection is deleted, but keeps it around in a "recycle bin". Every time an application calls the Connect() method with reusable set to true, driver tries to satisfy the request from a "recycle bin" first and opens a new connection only if necessary.

The pool_name argument is just an arbitrary string. An application could use this argument to assign a name to one or more connections (to create a connection pool) or to invoke a connection by name from this pool.

```
...
// Create a pool of four connections (two to one server and two to another)
// with the default database "DatabaseA"
CDB_Connection* con[4];
int i;
for (i = 4; i--; ) {
 con[i]= my_context.Connect((i%2 == 0) ? "MyServer1" : "MyServer2",
 "my_user_name", "my_password", 0, true,
 "ConnectionPoolA");
 CDB_LangCmd* lcmd= con[i]->LangCmd("use DatabaseA");
 lcmd->Send();
 while(lcmd->HasMoreResults()) {
 CDB_Result* r = lcmd->Result();
 delete r;
 }
 delete lcmd;
}
// return all connections to a "recycle bin"
for(i= 0; i < 4; delete con_array[i++]);
...
// in some other part of the program
// we want to get a connection from "ConnectionPoolA"
// but we don't want driver to open a new connection if pool is empty
try {
 CDB_Connection* my_con= my_context.Connect("", "", "", 0, true,
 "ConnectionPoolA");
 // Note that server name, user name and password are empty
 ...
}
catch (CDB_Exception& e) {
 // the pool is empty
 ...
}
```

An application could combine in one pool the connections to the different servers. This mechanism could also be used to group together the connections with some particular settings (default database, transaction isolation level, etc.).

Driver Manager

It's not always known which NCBI DBAPI driver will be used by a particular program. Sometimes you want a driver to be a parameter in your program. Sometimes you need to use two different drivers in one binary but can not link them statically because of name collisions. Sometimes you just need the driver contexts factory. The Driver Manager is intended to solve these problems.

Let's rewrite our Sample program using the Driver Manager. The original text was.

```
#include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
/* Here, XXXlib has to be replaced with the real name, e.g. "ctlib" */
#include <dbapi/driver/XXXlib/interfaces.hpp>
USING_NCBI_SCOPE;
int main()
{
 try { // to be sure that we are catching all driver related exceptions
 // We need to create a driver context first
 // In real program we have to replace CXXXContext with something real
 CXXXContext my_context;
 // connecting to server "MyServer"
 // with user name "my_user_name" and password "my_password"
 CDB_Connection* con = my_context.Connect("MyServer", "my_user_name",
 "my_password", 0);
 ...
```

If we use the Driver Manager we could allow the driver name to be a program argument.

```
#include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
#include <dbapi/driver/driver_mgr.hpp> // this is a new header
USING_NCBI_SCOPE;
int main(int argc, const char* argv[])
{
 try { // to be sure that we are catching all driver related exceptions
 C_DriverMgr drv_mgr;
 // We need to create a driver context first
 I_DriverContext* my_context= drv_mgr.GetDriverContext(
 (argc > 1)? argv[1] : "ctlib");
 // connecting to server "MyServer"
 // with user name "my_user_name" and password "my_password"
 CDB_Connection* con = my_context->Connect("MyServer", "my_user_name",
 "my_password", 0);
 ...
```

This fragment creates an instance of the Driver Manager, dynamically loads the driver's library, implicitly registers this driver, creates the driver context and makes a connection to a server. If you don't want to load some drivers dynamically for any reason, but want to use the Driver Manager as a driver contexts factory, then you need to statically link your program with those libraries and explicitly register those using functions from dbapi/driver/drivers.hpp header.

Text and Image Data Handling

text and image are SQL datatypes and can hold up to 2Gb of data. Because they could be huge, the RDBMS keeps these values separately from the other data in the table. In most cases the table itself keeps just a special pointer to a text/image value and the actual value is stored separately. This creates some difficulties for text/image data handling.

When you retrieve a large text/image value, you often prefer to "stream" it into your program and process it chunk by chunk rather than get it as one piece. Some RDBMS clients allow to stream the text/image values only if a corresponding column is the only column in a select statement.

Let's suppose that you have a table T (i_val int, t_val text) and you need to select all i_val, t_val where i_val > 0. The simplest way is to use a query:

```
select i_val, t_val from T where i_val > 0
```

But it could be expensive. Because two columns are selected, some clients will put the whole row in a buffer prior to giving access to it to the user. The better way to do this is to use two selects:

```
select i_val from T where i_val > 0
select t_val from T where i_val > 0
```

Looks ugly, but could save you a lot of memory.

Updating and inserting the text/image data is also not a straightforward process. For small texts and images it is possible to use just SQL insert and update statements, but it will be inefficient (if possible at all) for the large ones. The better way to insert and update text and image columns is to use the SendData() method of the CDB_Connection object or to use the CDB_SendDataCmd object.

The recommended algorithm for inserting text/image data is:

- Use a SQL insert statement to insert a new row into the table. Use a space value (' ') for each text column and a zero value (0x0) for each image column you are going to populate. Use NULL only if the value will remain NULL.
- Use a SQL select statement to select all text/image columns from this row.
- Fetch the row result and get an I_ITDescriptor for each column.
- Finish the results loop.
- Use the SendData() method or CDB_SendDataCmd object to populate the columns.

Example

Let's suppose that we want to insert a new row into table T as described above.

```
CDB_Connection* con;
...
// preparing the query
CDB_LangCmd* lcmd= con->LangCmd("insert T (i_val, t_val) values(100, ' ')
\n");
lcmd->More("select t_val from T where i_val = 100");
// Sending this query to a server
lcmd->Send();
```

```
I_ITDescriptor* my_descr;
// the result loop
while(lcmd->HasMoreResults()) {
 CDB_Result* r= lcmd->Result();
 // skip all but row result
 if (r == 0 || r->ResultType() != eDB_RowResult) {
 delete r;
 continue;
 }
 // fetching the row
 while(r->Fetch()) {
 // read 0 bytes from the text (some clients need this trick)
 r->ReadItem(0, 0);
 my_deskr = r->GetImageOrTextDescriptor();
 }
 delete r; // we don't need this result anymore
}
delete lcmd; // delete the command
CDB_Text my_text;
my_text.Append("This is a text I want to insert");
//sending the text
con->SendData(my_descr, my_text);
delete my_descr; // we don't need this descriptor anymore
...
```

The recommended algorithm for updating the text/image data is:

- Use a SQL update statement to replace the current value with a space value (' ') for a text column and a zero value (0x0) for an image column.
- Use a SQL select statement to select all text/image columns you want to update in this row.
- Fetch the row result and get an I_ITDescriptor for each column.
- Finish the results loop.
- Use the SendData() method or the CDB_SendDataCmd object to populate the columns.

Example

```
CDB_Connection* con;
...
// preparing the query
CDB_LangCmd* lcmd= con->LangCmd("update T set t_val= ' ' where i_val = 100");
lcmd->More("select t_val from T where i_val = 100");
// Sending this query to a server
lcmd->Send();
I_ITDescriptor* my_descr;
// the result loop
while(lcmd->HasMoreResults()) {
 CDB_Result* r= lcmd->Result();
 // skip all but row result
 if (r == 0 || r->ResultType() != eDB_RowResult) {
 delete r;
 continue;
 }
 // fetching the row
 while(r->Fetch()) {
 // read 0 bytes from the text (some clients need this trick)
 r->ReadItem(0, 0);
 my_deskr = r->GetImageOrTextDescriptor();
 }
 delete r; // we don't need this result anymore
}
delete lcmd; // delete the command
CDB_Text my_text;
my_text.Append("This is a text I want to see as an update");
//sending the text
con->SendData(my_descr, my_text);
delete my_descr; // we don't need this descriptor anymore
...
```

Results loop

Each connection in the NCBI DBAPI driver is always single threaded. Therefore, applications have to retrieve all the results from a current command prior to executing a new one. Not all results are meaningful (i.e. an RPC always returns a status result regardless of whether or not a procedure has a return statement), but all results need to be retrieved. The following loop is recommended for retrieving results from all types of commands:

```
CDB_XXXCmd* cmd; // XXX could be Lang, RPC, etc.
...
while (cmd->HasMoreResults()) {
 // HasMoreResults() method returns true // if the Result() method needs to
be called.
 // It doesn't guarantee that Result() will return not NULL result
 CDB_Result* res = cmd->Result();
 if (res == 0)
 continue; // a NULL res doesn't mean that there is no more results
 switch(res->ResultType()) {
 case eDB_RowResult: // row result
 while(res->Fetch()) {
 ...
 }
 break;
 case eDB_ParamResult: // Output parameters
 while(res->Fetch()) {
 ...
 }
 break;
 case eDB_ComputeResult: // Compute result
 while(res->Fetch()) {
 ...
 }
 break;
 case eDB_StatusResult: // Status result
 while(res->Fetch()) {
```

```
 ...
 }
 break;
 case eDB_CursorResult: // Cursor result
 while(res->Fetch()) {
 ...
 }
 break;
 }
 delete res;
}
```

If you don't want to process some particular type of result, just skip the while (res->Fetch()) {...} in the corresponding case.

Supported DBAPI drivers

- FreeTDS (TDS ver. 7.0)
- Sybase CTLIB
- Sybase DBLIB
- ODBC
- MySQL Driver

FreeTDS (TDS ver. 7.0)

This driver is the most recommended, built-in, and portable.

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_FTDS()
- Driver default name (for the run-time loading from a DLL) "ftds".
- Driver library ncbi_xdbapi_ftds
- FreeTDS libraries and headers used by the driver $(FTDS_LIBS) $(FTDS_INCLUDE)
- FreeTDS-specific driver context attributes "version", default = <DBVERSION_UNKNOWN> (also allowed: "42", "46", "70", "100")
- FreeTDS works on UNIX and Windows platforms.
- This driver supports Windows Domain Authentication using protocol NTLMv2, which is a default authentication protocol for Windows at NCBI.
- This driver supports TDS protocol version auto-detection. TDS protocol version cannot be detected when connecting against Sybase Open Server.
- Caveats:
  - Default version of the TDS protocol (<DBVERSION_UNKNOWN>) will work with MS SQL Server and Sybase SQL Server. If you want to work with Sybase Open server you should use TDS protocol version 4.6 or 5.0. This can be done either by using a driver parameter "version" equal either to "46" or to "50" or by setting an environment variable TDSVER either to "46" or to "50".
  - Although a slightly modified version of FreeTDS is now part of the C++ Toolkit, it retains its own license: the GNU Library General Public License.
  - TDS protocol version 4.2 should not be used with MS SQL server.

Sybase CTLIB

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_CTLIB()
- Driver default name (for the run-time loading from a DLL) "ctlib"
- Driver library ncbi_xdbapi_ctlib
- Sybase CTLIB libraries and headers used by the driver (UNIX) $(SYBASE_LIBS) $(SYBASE_INCLUDE)
- Sybase CTLIB libraries and headers used by the driver (MS Windows). You will need the Sybase OpenClient package installed on your PC. In MSVC++, set the "C/C++".General."Additional Include Directories" and Linker.General."Additional Library Directories" properties to the Sybase OpenClient headers and libraries (for example "C:\Sybase\include" and "C:\Sybase\lib" respectively). Also set the Linker.Input."Additional Dependencies" property to include the needed Sybase OpenClient libraries: LIBCT.LIB LIBCS.LIB LIBBLK.LIB. To run the application, you must set environment variable %SYBASE% to the Sybase OpenClient root directory (e.g. "C:\Sybase"), and also to have your "interfaces" file there, in INI/sql.ini. In NCBI, we have the Sybase OpenClient libs installed in \\snowman\win-coremake\Lib\ThirdParty\sybase.
- CTLIB-specific header (contains non-portable extensions) dbapi/driver/ctlib/interfaces.hpp
- CTLIB-specific driver context attributes "reuse_context" (default value is "true"), "version" (default value is "125", also allowed "100" and "110")
- Caveats:
  - Cannot communicate with MS SQL server using any TDS version.

Sybase DBLIB

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_DBLIB()
- Driver default name (for the run-time loading from a DLL) "dblib"
- Driver library dbapi_driver_dblib
- Sybase DBLIB libraries and headers used by the driver (UNIX) $(SYBASE_DBLIBS) $(SYBASE_INCLUDE)
- Sybase DBLIB libraries and headers used by the driver (MS Windows) Libraries: LIBSYBDB.LIB. See Sybase OpenClient installation and usage instructions in the Sybase CTLIB section (just above).
- DBLIB-specific header (contains non-portable extensions) dbapi/driver/dblib/interfaces.hpp
- DBLIB-specific driver context attributes "version" (default value is "46", also allowed "100")
- Caveats:
  - Text/image operations fail when working with MS SQL server, because MS SQL server sends text/image length in the reverse byte order, and this cannot be fixed (as it was fixed for FreeTDS) as we do not have access to the DBLIB source code.

- DB Library version level "100" is recommended for communication with Sybase server 12.5, because the default version level ("46") is not working correctly with this server.

ODBC

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_ODBC()
- Driver default name (for the run-time loading from a DLL) "odbc"
- Driver library dbapi_driver_odbc
- ODBC libraries and headers used by the driver (MS Windows) ODBC32.LIB ODBCCP32.LIB ODBCBCP.LIB
- ODBC libraries and headers used by the driver (UNIX) $(ODBC_LIBS)$(ODBC_INCLUDE)
- ODBC-specific header (contains non-portable extensions) dbapi/driver/odbc/interfaces.hpp
- ODBC-specific driver context attributes "version" (default value is "3", also allowed "2"), "use_dsn" (default value is false, if you have set this attribute to true, you need to define your data source using "Control Panel"/"Administrative Tools"/"Data Sources (ODBC)")
- Caveats:
  - The CDB_Result::GetImageOrTextDescriptor() does not work for ODBC driver. You need to use CDB_ITDescriptor instead. The other way to deal with texts/images in ODBC is through the CDB_CursorCmd methods: UpdateTextImage and SendDataCmd.
  - On most NCBI PCs, there is an old header odbcss.h (from 4/24/1998) installed. The symptom is that although everything compiles just fine, however in the linking stage there are dozens of unresolved symbol errors for ODBC functions. Ask "pc.systems" to fix this for your PC.
  - On UNIX, it's only known to work with Merant's implementation of ODBC, and it has not been thoroughly tested or widely used, so surprises are possible.

MySQL Driver

There is a direct (without ODBC) MySQL driver in the NCBI C++ Toolkit DBAPI. However, the driver implements a very minimal functionality and does not support the following:

- Working with images by chunks (images can be accessed as string fields though)
- RPC
- BCP
- SendData functionality
- Connection pools
- Parameter binding
- Canceling results
- ReadItem
- IsAlive
- Refresh functions
- Setting timeouts

Major Features of the BDB Library

The BDB library provides tools for the development of specialized data storage in applications not having access to a centralized RDBMS.

- **C++ wrapper on top of Berkeley DB.** The BDB library takes care of many of the ultra low-level details for C programmers using the Berkeley DB. The BDB implements B-Tree file access (both keyed and sequential), environments, cursors, and transactions.
- **Error checking.** All error codes coming from the Berkeley DB are analyzed and processed in a manner common to all other components of the C++ Toolkit. When an error situation is detected, the BDB library sends an exception that is reported by the diagnostic services and can be handled by the calling application, similar to any other Toolkit exception.
- **Support for relational table structure and different data types.** The Berkeley DB itself is “type agnostic” and provides no means to manipulate data types. But for many cases, clear data type support can save a lot of work. The Toolkit implements all major scalar data types so it can be used like a regular database.
- **Cross platform compatibility.** The BDB databases can be transferred across platforms without reconverting the data. The BDB tracks the fact that the database was created as big-endian or little-endian and does the conversion transparently when the database migrates.
- **Easy BLOBs.** The BDB library supports keyed BLOB storage. BLOBs can be streamed to and from the database. A set of additional interfaces has been written to simplify the BLOB access in comparison with the original Berkeley DB C library.
- **Disk-based cache interface.** The BDB library implements a cache disk cache service used by other Toolkit components to minimize client-server traffic and to store parts of the data locally. Different cache management and data expiration policies have been put in place.
- **Database maps.** The BDB library includes template classes similar to STL map and multimap but persistently stores the map content in the Berkeley DB files.
- **Simple queries.** The BDB library includes implementation of a simple query language to search records in flat files.

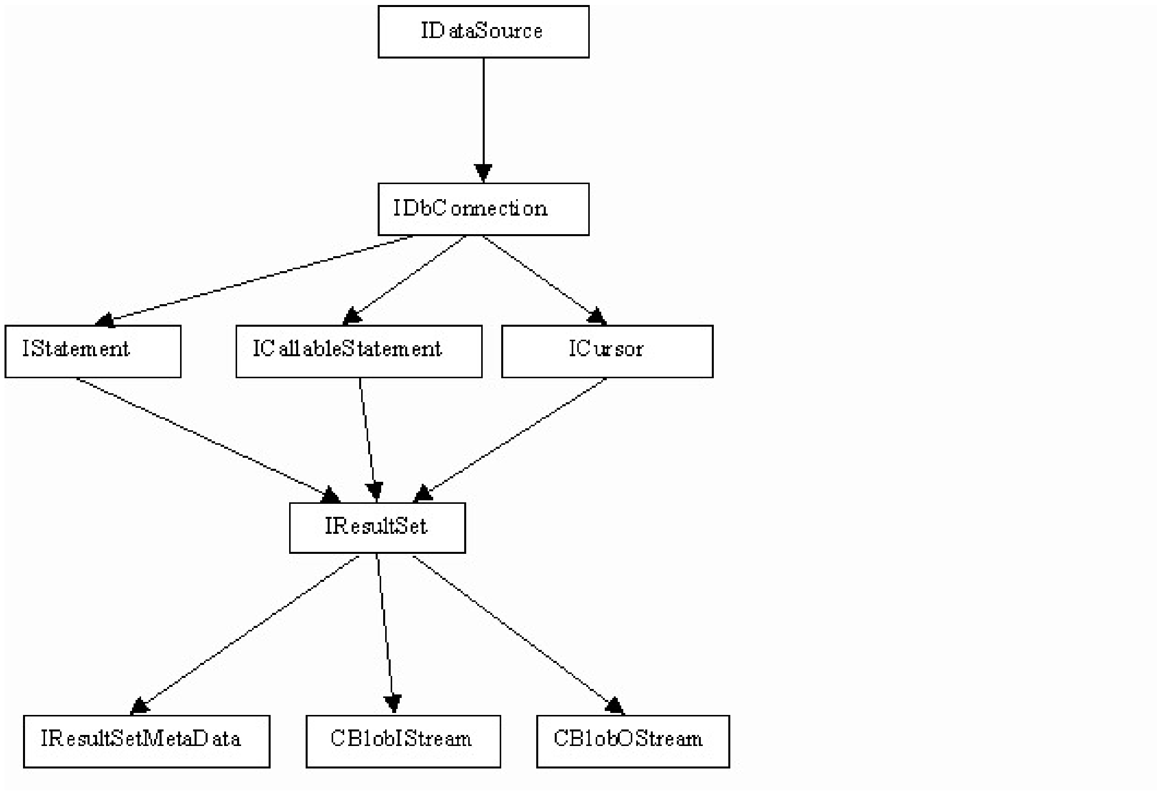


Figure 1. Object Hierarchy

11: CGI and Fast-CGI

Created: January 1, 2005.
Last Update: July 8, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

CGI and Fast-CGI [Libraries xcgi and xfcgi: include | src]

These library classes represent an integrated framework with which to write CGI applications and are designed to help retrieve and parse an HTTP request and then to compose and deliver an HTTP response. (See also this additional class reference documentation). xfcgi is a FastCGI version of xcgi.

Hint: Requires the target executable to be linked with a third-party FastCGI library, as in:

LIBS = $(FASTCGI_LIBS) $(ORIG_LIBS).

Hint: On non-FastCGI capable platforms (or if run as a plain CGI on a FastCGI-capable platform), it works the same as a plain CGI.

CGI Interface

- Basic CGI Application Class (includes CGI Diagnostic Handling) cgiapp[.hpp | .cpp]
- CGI Application Context Classes cgictx[.hpp | .cpp]
- HTTP Request Parser ncbicgi[.hpp | .cpp]
- HTTP Cookies ncbicgi[.hpp | .cpp]
- HTTP Response Generator ncbicgir[.hpp | .cpp]
- Basic CGI Resource Class ncbires[.hpp | .cpp]

FastCGI CGI Interface

- Adapter Between C++ and FastCGI Streams fcgibuf[.hpp | .cpp]
- Fast-CGI Loop Function fcgi_run[.cpp]
- Plain CGI Stub for the Fast-CGI Loop Function cgi_run[.cpp]

Demo Cases [src/cgi/demo | C++/src/sample/app/cgi/]

Test Cases [src/cgi/test]

.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Developing CGI applications

- Overview of the CGI classes
- The CCgiApplication class
- The CNcbiResource and CNcbiCommand classes
- The CCgiRequest class
- The CCgiResponse class
- The CCgiCookie class
- The CCgiCookies class
- The CCgiContext class
- The CCgiUserAgent class
- Example code using the CGI classes
- CGI Registry configuration

- Supplementary Information

Although CGI programs are generally run as web applications with HTML interfaces, this section of the Programming Manual places emphasis on the CGI side of things, omitting HTML details of the implementation where possible. Similarly, the section on Generating web pages focuses largely on the usage of HTML components independent of CGI details. The two branches of the NCBI C++ Toolkit hierarchy are all but independent of one another - with but one explicit hook between them: the constructors for HTML page components accept a CCgiApplication as an optional argument. This CCgiApplication argument provides the HTML page component with access to all of the CGI objects used in the application.

Further discussion of combining a CGI application with the HTML classes can be found in the section on An example web-based CGI application . The focus in this chapter is on the CGI classes only. For additional information about the CGI classes, the reader is also referred to the discussion of NCBI C++ CGI Classes in the Reference Manual.

The CGI classes

Figure 1 illustrates the layered design of the CGI classes.

This design is best described by starting with a consideration of the capabilities one might need to implement a CGI program, including:

- A way to retrieve and store the current values of environment variables
- A means of retrieving and interpreting the client's query request string
- Mechanisms to service and respond to the requested query
- Methods and data structures to obtain, store, modify, and send cookies
- A way to set/reset the context of the application (for Fast-CGI)

The CCgiContext class unifies these diverse capabilities under one aggregate structure. As their names suggest, the CCgiRequest class receives and parses the request, and the CCgiResponse class outputs the response on an output stream. All incoming CCgiCookies are also parsed and stored by the CCgiRequest object, and the outgoing cookies are sent along with the response by the CCgiResponse object. The request is actually processed by the application's CNcbiResource. The list of CNcbiCommands stored with that resource object are scanned to find a matching command, which is then executed.

The CCgiContext object, which is a friend to the CCgiApplication class, orchestrates this sequence of events in coordination with the application object. The same application may be run in many different contexts, but the resource and defined set of commands are invariant. What changes with each context is the request and its associated response.

The CCgiApplication class is a specialization of CNcbiApplication. Figure 2 illustrates the adaptation of the Init() and Run() member functions inherited from the CNcbiApplication class to the requirements of CGI programming. Although the application is contained in the context, it is the application which creates and initializes each context in which it participates. The program arguments and environmental variables are passed along to the context, where they will be stored, thus freeing the application to be restarted in a new context, as in Fast-CGI.

The application's ProcessRequest member function is an abstract function that must be implemented for each application project. In most cases, this function will access the query and the environment variables via the CCgiContext, using ctx.GetRequest() and ctx.GetConfig (). The application may then service the request using its resource's HandleRequest() method. The context's response object can then be used to send an appropriate response.

These classes are described in more detail below, along with abbreviated synopses of the class definitions. These are included here to provide a conceptual framework and are not intended as reference materials. For example, constructor and destructor declarations that operate on void arguments, and const methods that duplicate non-const declarations are generally not included here. Certain virtual functions and data members that have no meaning outside of a web application are also omitted. For complete definitions, refer to the header files via the source browsers.

The CCgiApplication Class (*)

As mentioned, the CCgiApplication class implements its own version of Init() , where it instantiates a CNcbiResource object using LoadResource(). Run() is no longer a pure virtual function in this subclass, and its implementation now calls CreateContext(), ProcessRequest (), and CCgiContext::GetResponse(). The CCgiApplication class does **not** have a CCgiContext data member, because the application object can participate in multiple CCgiContexts. Instead, a local variable in each Run() invocation stores a pointer to the context created there. The LoadServerContext() member function is used in Web applications, such as the query program, where it is necessary to store more complex run-time data with the context object. The CCgiServerContext object returned by this function is stored as a data member of a CCgiContext and is application specific.

```
class CCgiApplication : public CNcbiApplication
{
 friend class CCgiContext;

public:
 void Init(void);
 void Exit(void);
 int Run(void);

 CNcbiResource& GetResource(void);
 virtual int ProcessRequest(CCgiContext&) = 0;
 CNcbiResource* LoadResource(void);
 virtual CCgiServerContext* LoadServerContext(CCgiContext& context);

 bool RunFastCGI(unsigned def_iter=3);

protected:
 CCgiContext* CreateContext(CNcbiArguments*, CNcbiEnvironment*,
 CNcbiIstream*, CNcbiOstream*);

private: auto_ptr<CNcbiResource> m_resource;
};
```

If the program was **not** compiled as a FastCGI application (or the environment does not support FastCGI), then RunFastCGI() will return false. Otherwise, a "FastCGI loop" will be iterated over def_iter times, with the initialization methods and ProcessRequest() function being executed on each iteration. The value returned by RunFastCGI() in this case is true. Run() first calls RunFastCGI(), and if that returns false, the application is run as a plain CGI program.

The CNcbiResource (*) and CNcbiCommand (*) Classes

The resource class is at the heart of the application, and it is here that the program's functionality is defined. The single argument to the resource class's constructor is a CNcbiRegistry object, which defines data paths, resources, and possibly environmental variables for the application. This information is stored in the resource class's data member, m_config. The only other data member is a TCmdList (a list of CNcbiCommands) called m_cmd.

```
class CNcbiResource
{
public:

 CNcbiResource(CNcbiRegistry& config);

 CNcbiRegistry& GetConfig(void);
 const TCmdList& GetCmdList(void) const;
 virtual CNcbiCommand* GetDefaultCommand(void) const = 0;
 virtual const CNcbiResPresentation* GetPresentation(void) const;

 void AddCommand(CNcbiCommand* command);
 virtual void HandleRequest(CCgiContext& ctx);

protected:

 CNcbiRegistry& m_config;
 TCmdList m_cmd;
};
```

The AddCommand() method is used when a resource is being initialized, to add commands to the command list. Given a CCgiRequest object defined in a particular context ctx, HandleRequest(ctx) compares entries in the context's request to commands in m_cmd. The first command in m_cmd that matches an entry in the request is then executed (see below), and the request is considered "handled". If desired, a default command can be installed that will execute when no matching command is found. The default command is defined by implementing the pure virtual function GetDefaultCommand(). The CNcbiResPresentation class is an abstract base class, and the member function, GetPresentation(), returns 0. It is provided as a hook for implementing interfaces between information resources (e.g., databases) and CGI applications.

```
class CNcbiCommand
{
public:
 CNcbiCommand(CNcbiResource& resource);

 virtual CNcbiCommand* Clone(void) const = 0;
 virtual string GetName() const = 0;
 virtual void Execute(CCgiContext& ctx) = 0;
 virtual bool IsRequested(const CCgiContext& ctx) const;

protected:
 virtual string GetEntry() const = 0;
 CNcbiResource& GetResource() const { return m_resource; }
private:
 CNcbiResource& m_resource;
};
```

CNcbiCommand is an abstract base class; its only data member is a reference to the resource it belongs to, and most of its methods - with the exception of GetResource() and IsRequested () - are pure virtual functions. IsRequested() examines the key=value entries stored with the context's request object. When an entry is found where key==GetEntry() and value==GetName (), IsRequested() returns true.

The resource's HandleRequest() method iterates over its command list, calling CNcbiCommand::IsRequested() until the first match between a command and a request entry is found. When IsRequested() returns true, the command is cloned, and the cloned command is then executed. Both the Execute() and Clone() methods are pure virtual functions that must be implemented by the user.

The CCgiRequest Class (*)

The CCgiRequest class serves as an interface between the user's query and the CGI program. Arguments to the constructor include a CNcbiArguments object, a CNcbiEnvironment object, and a CNcbiIstream object. The class constructors do little other than invoke CCgiRequest::x_Init(), where the actual initialization takes place.

x_Init() begins by examining the environment argument, and if it is NULL, m_OwnEnv (an auto_ptr) is reset to a dummy environment. Otherwise, m_OwnEnv is reset to the passed environment, making the request object the effective owner of that environment. The environment is then used to cache network information as "gettable" properties. Cached properties include:

- server properties, such as the server name, gateway interface, and server port
- client properties (the remote host and remote address)
- client data properties (content type and content length of the request)
- request properties, including the request method, query string, and path information
- authentication information, such as the remote user and remote identity
- standard HTTP properties (from the HTTP header)

These properties are keyed to an enumeration named ECgiProp and can be retrieved using the request object's GetProperty() member function. For example, GetProperty(eCgi_HttpCookie) is used to access cookies from the HTTP Header, and GetProperty(eCgi_RequestMethod) is used to determine from where the query string should be read.

NOTE: Setting $QUERY_STRING without also setting $REQUEST_METHOD will result in a failure by x_init() to read the input query. x_init() first looks for the definition of $REQUEST_METHOD, and depending on if it is GET or POST, reads the query from the environment or the input stream, respectively. If the environment does not define $REQUEST_METHOD, then x_Init() will try to read the query string from the command line only.

```
class CCgiRequest {
public:
 CCgiRequest(const CNcbiArguments*, const CNcbiEnvironment*,
 CNcbiIstream*, TFlags);
 static const string& GetPropertyName(ECgiProp prop);
 const string& GetProperty(ECgiProp prop) const;
 size_t GetContentLength(void) const;
 const CCgiCookies& GetCookies(void) const;
 const TCgiEntries& GetEntries(void) const;
 static SIZE_TYPE ParseEntries(const string& str, TCgiEntries& entries);
private:
 void x_Init(const CNcbiArguments*, const CNcbiEnvironment*,
 CNcbiIstream*, TFlags);

 const CNcbiEnvironment* m_Env;
 auto_ptr<CNcbiEnvironment> m_OwnEnv;
 TCgiEntries m_Entries;
 CCgiCookies m_Cookies;
};
```

This abbreviated definition of the CCgiRequest class highlights its primary functions:

To parse and store the <key=value> pairs contained in the query string (stored in m_Entries).

To parse and store the cookies contained in the HTTP header (stored in m_Cookies).

As implied by the "T" prefix, TCgiEntries is a type definition, and defines m_Entries to be an STL multimap of <string,string> pairs. The CCgiCookies class (described below) contains an STL set of CCgiCookie and implements an interface to this set.

The CCgiResponse Class (*)

The CCgiResponse class provides an interface to the program's output stream (usually cout), which is the sole argument to the constructor for CCgiResponse. The output stream can be accessed by the program using CCgiResponse::GetOutput(), which returns a pointer to the output stream, or, by using CCgiResponse::out(), which returns a reference to that stream.

In addition to implementing controlled access to the output stream, the primary function of the response class is to generate appropriate HTML headers that will precede the rest of the response. For example, a typical sequence in the implementation of a particular command's execute function might be:

```
MyCommand::Execute(CCgiContext& ctx)
{
 // ... generate the output and store it in MyOutput

 ctx.GetResponse().WriteHeader();
 ctx.GetResponse().out() << MyOutput;
 ctx.GetResponse.out() << "</body></html>" << endl;
 ctx.GetResponse.Flush();
}
```

Any cookies that are to be sent with the response are included in the headers generated by the response object.

Two member functions are provided for outputting HTML headers: WriteHeader() and WriteHeader(CNcbiOstream&). The second of these is for writing to a specified stream other than the default stream stored with the response object. Thus, WriteHeader(out()) is equivalent to WriteHeader().

The WriteHeader() function begins by invoking IsRawCgi() to see whether the application is a non-parsed header program. If so, then the first header put on the output stream is an HTTP status line, taken from the private static data member, sm_HTTPStatusDefault. Next, unless the content type has been set by the user (using SetContentType()), a default content line is written, using sm_ContentTypeDefault. Any cookies stored in m_Cookies are then written, followed by any additional headers stored with the request in m_HeaderValues. Finally, a new line is written to separate the body from the headers.

```
class CCgiResponse {
public:
 CCgiResponse(CNcbiOstream* out = 0);

 void SetRawCgi(bool raw);
 bool IsRawCgi(void) const;
 void SetHeaderValue(const string& name, const string& value);
 void SetHeaderValue(const string& name, const tm& value);
 void RemoveHeaderValue(const string& name);
 void SetContentType(const string &type);
 string GetHeaderValue(const string& name) const;
 bool HaveHeaderValue(const string& name) const;
 string GetContentType(void) const;

 CCgiCookies& Cookies(void); // Get cookies set
 CNcbiOstream* SetOutput(CNcbiOstream* out); // Set default output stream
 CNcbiOstream* GetOutput(void) const; // Query output stream
 CNcbiOstream& out(void) const; // Conversion to ostream
 // to enable <<

 void Flush() const;

 CNcbiOstream& WriteHeader(void) const; // Write HTTP response header
 CNcbiOstream& WriteHeader(CNcbiOstream& out) const;
protected:
 typedef map<string, string> TMap;
 static const string sm_ContentTypeName;
 static const string sm_ContentTypeDefault;
 static const string sm_HTTPStatusDefault;
 bool m_RawCgi;
 CCgiCookies m_Cookies;
 TMap m_HeaderValues; // Additional header lines in alphabetical order
 CNcbiOstream* m_Output; // Default output stream };
```

The CCgiCookie Class (*)

The traditional means of maintaining state information when servicing a multi-step request has been to include hidden input elements in the query strings passed to subsequent URLs. The newer, preferred method uses HTTP cookies, which provide the server access to client-side state information stored with the client. The cookie is a text string consisting of four key=value pairs:

- name (required)
- expires (optional)
- domain (optional)
- path (optional)

The CCgiCookie class provides a means of creating, modifying, and sending cookies. The constructor requires at least two arguments, specifying the name and value of the cookie, along with the optional domain and path arguments. Format errors in the arguments to the constructor (see Supplementary Information) will cause the invalid argument to be thrown. The CCgiCookie::Write(CNcbiOstream&) member function creates a Set-Cookie directive using its private data members and places the resulting string on the specified output stream:

```
Set-Cookie:
m_Name=
m_Value; expires=
m_Expires; path=
m_Path;
domain=
m_Domain;
m_Secure
```

As with the constructor, and in compliance with the proposed standard (RFC 2109), only the name and value are mandatory in the directive.

```
class CCgiCookie {
public:
 CCgiCookie(const string& name, const string& value,
 const string& domain = NcbiEmptyString,
 const string& path = NcbiEmptyString);
 const string& GetName(void) const;
 CNcbiOstream& Write(CNcbiOstream& os) const;
 void Reset(void);
 void CopyAttributes(const CCgiCookie& cookie);
 void SetValue (const string& str);
 void SetDomain (const string& str);
 void SetPath (const string& str);
 void SetExpDate(const tm& exp_date);
 void SetSecure (bool secure);
 const string& GetValue (void) const;
 const string& GetDomain (void) const;
 const string& GetPath (void) const;
 string GetExpDate(void) const;
 bool GetExpDate(tm* exp_date) const;
 bool GetSecure(void) const;
 bool operator<(const CCgiCookie& cookie) const;
 typedef const CCgiCookie* TCPtr;
 struct PLessCPtr {
 bool operator() (const TCPtr& c1, const TCPtr& c2) const {
 return (*c1 < *c2);
 }
 };
private:
 string m_Name;
 string m_Value;
 string m_Domain;
 string m_Path;
 tm m_Expires;
 bool m_Secure;
};
```

With the exception of m_Name, all of the cookie's data members can be reset using the SetXxx(), Reset(), and CopyAttributes() member functions; m_Name is non-mutable. As with the constructor, format errors in the arguments to these functions will cause the invalid argument to be thrown. By default, m_Secure is false. The GetXxx() methods return the stored value for that attribute or, if no value has been set, a reference to NcbiEmptyString. GetExpDate(tm*) returns false if no expiration date was previously set. Otherwise, tm is reset to m_Expire, and true is returned.

The CCgiCookies Class (*)

The CCgiCookies class provides an interface to an STL set of CCgiCookies (m_Cookies). Each cookie in the set is uniquely identified by its name, domain, and path values and is stored in ascending order using the CCgiCookie::PLessCPtr construct. Two constructors are provided, allowing the user to initialize m_Cookies to either an empty set or to a set of N new cookies created from the string "name1=value1; name2=value2; ...; nameN=valuenN". Many of the operations on a CCgiCookies object involve iterating over the set, and the class's type definitions support these activities by providing built-in iterators and a typedef for the set, TSet.

The Add() methods provide a variety of options for creating and adding new cookies to the set. As with the constructor, a single string of name-value pairs may be used to create and add N cookies to the set at once. Previously created cookies can also be added to the set individually or as sets. Similarly, the Remove() methods allow individual cookies or sets of cookies (in the specified range) to be removed. All of the remove functions destroy the removed cookies when destroy=true. CCgiCookies::Write(CNcbiOstream&) iteratively invokes the CCgiCookie::Write() on each element.

```
class CCgiCookies {
public:
 typedef set<CCgiCookie*, CCgiCookie::PLessCPtr> TSet;
 typedef TSet::iterator TIter;
 typedef TSet::const_iterator TCIter;
 typedef pair<TIter, TIter> TRange;
 typedef pair<TCIter, TCIter> TCRange;
 CCgiCookies(void); // create empty set of cookies
 CCgiCookies(const string& str);
 // str = "name1=value1; name2=value2; ..."
 bool Empty(void) const;
 CCgiCookie* Add(const string& name, const string& value,
 const string& domain = NcbiEmptyString,
 const string& path = NcbiEmptyString);
 CCgiCookie* Add(const CCgiCookie& cookie);
 void Add(const CCgiCookies& cookies);
 void Add(const string& str);
 // "name1=value1; name2=value2; ..."
 CCgiCookie* Find(const string& name, const string& domain,
 const string& path);
 CCgiCookie* Find(const string& name, TRange* range=0);
 bool Remove(CCgiCookie* cookie, bool destroy=true);
 size_t Remove(TRange& range, bool destroy=true);
 size_t Remove(const string& name, bool destroy=true);
 void Clear(void);
 CNcbiOstream& Write(CNcbiOstream& os) const;
private:
 TSet m_Cookies;
};
```

The CCgiContext Class (*)

As depicted in Figure 1 , a CCgiContext object contains an application object, a request object, and a response object, corresponding to its data members m_app, m_request, and m_response. Additional data members include a string encoding the URL for the context (m_selfURL), a message buffer (m_lmsg), and a CCgiServerContext. These last three data members are used only in complex Web applications, such as the query program, where it is necessary to store more complex run-time data with the context object. The message buffer is essentially an STL list of string objects the class definition of which (CCtxMsgString) includes a Write() output function. GetServCtx() returns m_srvCtx if it has been defined and, otherwise, calls the application's CCgiApplication::LoadServerContext() to obtain it.

```
class CCgiContext
{
public:
 CCgiContext(CCgiApplication& app,
 const CNcbiArguments* args = 0,
 const CNcbiEnvironment* env = 0,
 CNcbiIstream* inp = 0,
 CNcbiOstream* out = 0);
 const CCgiApplication& GetApp(void) const;
 CNcbiRegistry& GetConfig(void);
 CCgiRequest& GetRequest(void);
 CCgiResponse& GetResponse(void);
 const string& GetSelfURL(void) const;
 CNcbiResource& GetResource(void);
 CCgiServerContext& GetServCtx(void);
 // output all msgs in m_lmsg to os
 CNcbiOstream& PrintMsg(CNcbiOstream& os);
 void PutMsg(const string& msg); // add message to m_lmsg
 void PutMsg(CCtxMsg* msg); // add message to m_lmsg
 bool EmptyMsg(void); // true iff m_lmsg is empty
 void ClearMsg(void); // delete all messages in m_lmsg
 string GetRequestValue(const string& name) const;
 void AddRequestValue(const string& name, const string& value);
 void RemoveRequestValues(const string& name);
 void ReplaceRequestValue(const string& name, const string& value);
private:
 CCgiApplication& m_app;
 auto_ptr<CCgiRequest> m_request;
 CCgiResponse m_response;
 mutable string m_selfURL;
 list<CCtxMsg*> m_lmsg; // message buffer
 auto_ptr<CCgiServerContext> m_srvCtx;
 // defined by CCgiApplication::LoadServerContext()
 friend class CCgiApplication;
};
```

The CCgiUserAgent class (*)

The CCgiUserAgent class is used to gather information about the client's user agent - i.e. browser type, browser name, browser version, browser engine type, browser engine version, Mozilla version (if applicable), platform, and robot information. The default constructor looks for the user agent string first in the CCgiApplication context using the eCgi_HttpUserAgent request property, then in the CNcbiApplication instance HTTP_USER_AGENT environment variable, and finally in the operating system HTTP_USER_AGENT environment variable.

```
class CCgiUserAgent
{
public:
 CCgiUserAgent(void);
 CCgiUserAgent(const string& user_agent);

 void Reset(const string& user_agent);

 string GetUserAgentStr(void) const;
 EBrowser GetBrowser(void) const;
 const string& GetBrowserName(void) const;
 EBrowserEngine GetEngine(void) const;
 EBrowserPlatform GetPlatform(void) const;

 const TUserAgentVersion& GetBrowserVersion(void) const;
 const TUserAgentVersion& GetEngineVersion(void) const;
 const TUserAgentVersion& GetMozillaVersion(void) const;

 typedef unsigned int TBotFlags;
 bool IsBot(TBotFlags flags = fBotAll, const string& patterns = kEmptyStr)
const;

protected:
 void x_Init(void);
 void x_Parse(const string& user_agent);
 bool x_ParseToken(const string& token, int where);

protected:
 string m_UserAgent;
 EBrowser m_Browser;
 string m_BrowserName;
 TUserAgentVersion m_BrowserVersion;
 EBrowserEngine m_Engine;
 TUserAgentVersion m_EngineVersion;
 TUserAgentVersion m_MozillaVersion;
 EBrowserPlatform m_Platform;
};
```

Example Code Using the CGI Classes

The sample CGI program demonstrates a simple application that combines the NCBI C++ Toolkit's CGI and HTML classes. justcgi.cpp is an adaptation of that program, stripped of all HTML references and with additional request-processing added (see Box 1 and Box 2).

Executing

```
./cgi 'cmd1=init&cmd2=reply'
```

results in execution of only cmd1, as does executing

```
./cgi 'cmd2=reply&cmd1=init'
```

The commands are matched in the order that they are registered with the resource, not according to the order in which they occur in the request. The assumption is that only the first entry (if any) in the query actually specifies a command, and that the remaining entries provide optional arguments to that command. The Makefile (see Box 3) for this example links to both the xncbi and xcgi libraries. Additional examples using the CGI classes can be found in src/cgi/test . (For Makefile.fastcgi.app, see Box 4 .)

CGI Registry Configuration

The application registry defines CGI-related configuration settings in the [CGI] section (see this table).

FastCGI settings. [FastCGI] section (see this table).

CGI load balancing settings. [CGI-LB] section (see this table).

Supplementary Information

Restrictions on arguments to the CCgiCookie constructor.

See Table 1.

CGI Diagnostic Handling

By default, CGI applications support three query parameters affecting diagnostic output : diag-destination , diag-threshold , and diag-format . It is possible to modify this behavior by overriding the virtual function CCgiApplication::ConfigureDiagnostics . (In particular, production applications may wish to disable these parameters by defining ConfigureDiagnostics to be a no-op.)

diag-destination

The parameter diag-destination controls where diagnostics appear. By default, there are two possible values (see Table 2).

However, an application can make other options available by calling RegisterDiagFactory from its Init routine. In particular, calling

```
#include <connect/email_diag_handler.hpp>
...
RegisterDiagFactory("email", new CEmailDiagFactory);
```

and linking against xconnect and connect enables destinations of the form email:user@host, which will cause the application to e-mail diagnostics to the specified address when done.

Similarly, calling

```
#include <html/commentdiag.hpp>
...
RegisterDiagFactory("comments", new CCommentDiagFactory);
```

and linking against xhtml will enable the destination comments. With this destination, diagnostics will take the form of comments in the generated HTML, provided that the application has also used SetDiagNode to indicate where they should go. (Applications may call that function repeatedly; each invocation will affect all diagnostics until the next invocation. Also, SetDiagNode is effectively a no-op for destinations other than comments, so applications may call it unconditionally.)

Those destinations are not available by default because they introduce additional dependencies; however, either may become a standard possibility in future versions of the toolkit.

diag-threshold

The parameter diag-threshold sets the minimum severity level of displayed diagnostics; its value can be either fatal, critical, error, warning, info, or trace. For the most part, setting this parameter is simply akin to calling SetDiagPostLevel . However, setting diag-threshold to trace is **not** equivalent to calling SetDiagPostLevel(eDiag_Trace); the former reports all diagnostics, whereas the latter reports only traces.

diag-format

Finally, the parameter diag-format controls diagnostics' default appearance; setting it is akin to calling {Set,Unset}DiagPostFlag . Its value is a list of flags, delimited by spaces (which appear as "+" signs in URLs); possible flags are file, path, line, prefix, severity, code, subcode, time, omitinfosev, all, trace, log, and default. Every flag but default corresponds to a value in EDiagPostFlag , and can be turned off by preceding its name with an exclamation point ("!"). default corresponds to the four flags which are on by default: line, prefix, code, and subcode, and may not be subtracted.

NCBI C++ CGI Classes

The Common Gateway Interface (CGI) is a method used by web servers to pass information from forms displayed in a web browser to a program running on the server and then allow the program to pass a web page back. The NCBI C++ CGI Classes are used by the program running on the server to decode the CGI input from the server and to send a response. The library also supports cookies, which is a method for storing information on the user's machine. The library supports the http methods GET and POST via application/x-www-form-urlencoded, and supports the POST via multipart/form-data (often used for file upload). In the POST via multipart/form-data, the data gets read into a TCgiEntries; you also can get the filename out of it (the name of the entry is as specified by "name=" of the data-part header). For more information on CGI, see the book **HTML Sourcebook** by Ian Graham or http://hoohoo.ncsa.uiuc.edu/cgi/

There are 5 main classes:

CCgiRequest–what the CGI program is getting from the client.

CCgiResponse–what the CGI program is sending to the client.

CCgiEntry–a single field value, optionally accompanied by a filename.

CCookie–a single cookie

CCookies–a cookie container

Note: In the following libraries you will see references to the following typedefs: CNcbiOstream and CNcbiIstream. On Solaris and NT, these are identical to the standard library output stream (ostream) and input stream (istream) classes. These typedefs are used on older computers to switch between the old stream library and the new standard library stream classes. Further details can be found in an accompanying document (to be written).

A demo program, cgidemo.cpp, can be found in internal/c++/src/corelib/demo.

CCgiRequest

CCgiRequest is the class that reads in the input from the web server and makes it accessible to the CGI program.

CCgiRequest uses the following typedefs to simplify the code:

```
typedef map<string, string> TCgiProperties
typedef multimap<string, CCgiEntry> TCgiEntries
typedef TCgiEntries::iterator TCgiEntriesI
typedef list<string> TCgiIndexes
```

All of the basic types come from the C++ Standard library (http://www.sgi.com/Technology/STL/)

CCgiRequest(int argc, char* argv[], CNcbiIstream* istr=0, bool indexes_as_entries=true)

A CGI program can receive its input from three sources: the command line, environment variables, and an input stream. Some of this input is given to the CCgiRequest class by the following arguments to the constructor:

int argc, char* argv[] : standard command line arguments.

CNcbiIstream* istr=0 : the input stream to read from. If 0, reads from stdin, which is what most web servers use.

bool indexes_as_entries=true : if query has any ISINDEX like terms (i.e. no "=" sign), treat it as a form query (i.e. as if it had an "=" sign).

Example:

```
CCgiRequest * MyRequest = new CCgiRequest(argc, argv);
```

const TCgiEntries& GetEntries(void) const

Get a set of decoded form entries received from the web browser. So if you sent a cgi query of the form ?name=value, the multimap referenced by TCgiEntries& includes "name" as a .first member and <"value", ""> as a .second member.

TCgiEntries& also includes "indexes" if "indexes_as_entries" in the constructor was "true".

const TCgiIndexes& GetIndexes(void) const

This performs a similar task as GetEntries(), but gets a set of decoded entries received from the web browser that are ISINDEX like terms (i.e. no "=" sign),. It will always be empty if "indexes_as_entries" in the constructor was "true"(default).

const string& GetProperty(ECgiProp prop) const

Get the value of a standard property (empty string if not specified). See the "Standard properties" list below.

static const string& GetPropertyName(ECgiProp prop)

The web server sends the CGI program properties of the web server and the http headers received from the web browser (headers are simply additional lines of information sent in a http request and response). This API gets the name(not value!) of standard properties. See the "Standard properties" list below.

Standard properties:

```
eCgi_ServerSoftware ,
eCgi_ServerName,
eCgi_GatewayInterface,
eCgi_ServerProtocol,
eCgi_ServerPort, // see also "GetServerPort()"
// client properties
eCgi_RemoteHost,
eCgi_RemoteAddr, // see also "GetRemoteAddr()"
// client data properties
eCgi_ContentType,
eCgi_ContentLength, // see also "GetContentLength()"
// request properties
eCgi_RequestMethod,
eCgi_PathInfo,
eCgi_PathTranslated,
eCgi_ScriptName,
eCgi_QueryString,
// authentication info
eCgi_AuthType,
eCgi_RemoteUser,
eCgi_RemoteIdent,
// semi-standard properties(from HTTP header)
eCgi_HttpAccept,
eCgi_HttpCookie,
eCgi_HttpIfModifiedSince,
eCgi_HttpReferer,
eCgi_HttpUserAgent
```

const string& GetRandomProperty(const string& key) const

Gets value of any http header that is passed to the CGI program using environment variables of the form "$HTTP_<key>". In general, these are special purpose http headers not included in the list above.

Uint2 GetServerPort(void) const

Gets the server port used by web browser to access the server.

size_t GetContentLength(void) const

Returns the length of the http request.

const CCgiCookies& GetCookies(void) const

Retrieve the cookies that were sent with the request. Cookies are text buffers that are stored in the user's web browsers and can be set and read via http headers. See the CCookie and CCookies classes defined below.

static SIZE_TYPE ParseEntries(const string& str, TCgiEntries& entries)

This is a helper function that isn't normally used by CGI programs. It allows you to decode the URL-encoded string "str" into a set of entries <"name", "value"> and add them to the "entries" multimap. The new entries are added without overriding the original ones, even if they have the same names. If the "str" is in ISINDEX format then the entry "value" will be empty. On success, return zero; otherwise return location(1-base) of error.

static SIZE_TYPE ParseIndexes(const string& str, TCgiIndexes& indexes)

This is also a helper function not usually used by CGI programs. This function decodes the URL-encoded string "str" into a set of ISINDEX-like entries (i.e. no "=" signs in the query) and adds them to the "indexes" set. On success, return zero, otherwise return location(1-base) of error.

CCgiResponse

CCgiResponse is the object that takes output from the CGI program and sends it to the web browser via the web server.

CNcbiOstream& WriteHeader() const

CNcbiOstream& WriteHeader(CNcbiOstream& out) const

This writes the MIME header necessary for all documents sent back to the web browser. By default, this function assumes that the "Content-type" is "text/html". Use the second form of the function if you want to use a stream other that the default.

void SetContentType(const string &type)

Sets the content type. By default this is "text/html". For example, if you were to send plaintext back to the client, you would set type to "text/plain".

string GetContentType(void) const

Retrieves the content type.

CNcbiOstream& out(void) const

This returns a reference to the output stream being used by the CCgiResponse object. Example:

```
CCgiResponse Response;
Response.WriteHeader();
Response.out() << "hello, world" << flush;
```

CNcbiOstream* SetOutput(CNcbiOstream* out)

Sets the default output stream. By default this is stdout, which is what most web servers use.

CNcbiOstream* GetOutput(void) const

Get the default output stream.

void Flush() const

Flushes the output stream.

void SetRawCgi(bool raw)

Turns on non-parsed cgi mode. When this is turned on AND the name of the cgi program begins with "nph-", then the web server does no processing of the data sent back to the client. In this situation, the client must provide all appropriate http headers. This boolean switch causes some of these headers to be sent.

bool IsRawCgi(void) const

Check to see if non-parsed cgi mode is on.

void SetHeaderValue(const string& name, const string& value)

Sets an http header with given name and value. For example, SetHeaderValue("Mime-Version", "1.0"); will create the header "Mime-Version: 1.0".

void SetHeaderValue(const string& name, const tm& value)

Similar to the above, but sets a header value using a date. See time.h for the definition of tm.

void RemoveHeaderValue(const string& name)

Remove the header with name name.

string GetHeaderValue(const string& name) const

Get the value of the header with name name.

bool HaveHeaderValue(const string& name) const

Check to see if the header with name name exists.

void AddCookie(const string& name, const string& value) void AddCookie(const CCgiCookie& cookie)

Add a cookie to the response. This can either be a name, value pair or use the CCookie class described below.

void AddCookies(const CCgiCookies& cookies)

Add a set of cookies to the response. See the CCookies class described below.

const CCgiCookies& Cookies(void) const CCgiCookies& Cookies(void)

Return the set of cookies to be sent in the response.

void RemoveCookie(const string& name)

Remove the cookie with the name name.

void RemoveAllCookies(void)

Remove all cookies.

bool HaveCookies(void) const

Are there cookies?

bool HaveCookie(const string& name) const

Is there a cookie with the given name?

CCgiCookie* FindCookie(const string& name) const

Return a cookie with the given name.

CCgiCookie

A cookie is a name, value string pair that can be stored on the user's web browser. Cookies are allocated per site and have restrictions on size and number. Cookies have attributes, such as the domain they originated from. CCgiCookie is used by the CCgiRequest and CCgiResponse classes.

CCgiCookie(const string& name, const string& value)

Creates a cookie with the given name and value. Throw the "invalid_argument" if "name" or "value" have invalid format:

- the "name" must not be empty; it must not contain '='
- both "name" and "value" must not contain: ";, "

const string& GetName (void) const

Get the cookie name. The cookie name cannot be changed.

CNcbiOstream& Write(CNcbiOstream& os) const

Write the cookie out to ostream os. Normally this is handled by CCgiResponse.

void Reset(void)

Reset everything but the name to the default state

void CopyAttributes(const CCgiCookie& cookie)

Set all attribute values(but name!) to those from "cookie"

void SetValue (const string& str) void SetDomain (const string& str) void SetValidPath (const string& str) void SetExpDate (const tm& exp_date) void SetSecure (bool secure) // "false" by default

These function set the various properties of a cookie. These functions will throw "invalid_argument" if "str" has invalid format. For the definition of tm, see time.h.

bool GetValue (string* str) const bool GetDomain (string* str) const bool GetValidPath (string* str) const bool GetExpDate (string* str) const bool GetExpDate (tm* exp_date) const bool GetSecure (void) const

These functions return true if the property is set. They also return value of the property in the argument. If the property is not set, str is emptied. These functions throw the "invalid_argument" exception if the argument is a zero pointer.

The string version of GetExpDate will return a string of the form "Wed Aug 9 07:49:37 1994"

CCgiCookies

CCgiCookies aggregates a collection of CCgiCookie

CCgiCookies(void) CCgiCookies(const string& str)

Creates a CCgiCookies container. To initialize it with a cookie string, use the format: "name1=value1; name2=value2; ..."

CCgiCookie* Add(const string& name, const string& value)

Add a cookie with the given name, value pair. Note the above requirements on the string format. Overrides any previous cookie with same name.

CCgiCookie* Add(const CCgiCookie& cookie)

Add a CCgiCookie.

void Add(const CCgiCookies& cookies)

Adds a CCgiCookie of cookies.

void Add(const string& str)

Adds cookies using a string of the format "name1=value1; name2=value2; ..." Overrides any previous cookies with same names.

CCgiCookie* Find(const string& name) const

Looks for a cookie with the given name. Returns zero not found.

bool Empty(void) const

"true" if contains no cookies.

bool Remove(const string& name)

Find and remove a cookie with the given name. Returns "false" if one is not found.

void Clear(void)

Remove all stored cookies

CNcbiOstream& Write(CNcbiOstream& os) const

Prints all cookies into the stream "os" (see also CCgiCookie::Write()). Normally this is handled by CCgiResponse.

An example web-based CGI application

Introduction

The previous two chapters described the NCBI C++ Toolkit's CGI and HTML classes, with an emphasis on their independence from one another. In practice however, a real application must employ both types of objects, with a good deal of inter-dependency.

As described in the description of the CGI classes, the CNcbiResource class can be used to implement an application whose functionality varies with the query string. Specifically, the resource class contains a list of CNcbiCommand objects, each of which has a defined GetName () and GetEntry()method. The only command selected for execution on a given query is the one whose GetName() and GetEntry() values match the leading key=value pair in the query string.

The CHelloResource class has different commands which will be executed depending on whether the query string invoked an init or a reply command. For many applications however, this selection mechanism adds unnecessary complexity to the interface, as the application always performs the same function, albeit on different input. In these cases, there is no need to use a CNcbiResource object, or CNcbiCommand objects, as the necessary functionality can be encoded directly in the application's ProcessRequest() method. The example program described in this section uses this simpler approach.

Program description

The car.cgi program presents an HTML form for ordering a custom color car with selected features. The form includes a group of checkboxes (listing individual features) and a set of radio buttons listing possible colors. Initially, no features are selected, and the default color is black. Following the form, a summary stating the currently selected features and color, along with a price quote, is displayed. When the submit button is clicked, the form generates a new query string (which includes the selected features and color), and the program is restarted.

The program uses a CHTMLPage object with a template file (car.html) to create the display. The template file contains three <@tag@> locations, which the program uses to map CNCBINodes to the page, using the AddTagMap() method. Here is an outline of the execution sequence:

Create an instance of class CCar named car.

Load car with the color and features specified in the query string.

Create a CHTMLPage named page.

Generate a CHTML_form object using the features and color currently selected for car, and map that HTML form to the <@FORM@> tag in page.

Generate the summary statement and save it in a CHTMLText node mapped to the <@SUMMARY@> tag.

Generate a price quote and save it in a CHTMLText node mapped to the <@PRICE@> tag.

Output page and exit.

The CCar created in step 1 initially has the default color (black) and no features. Any features or colors specified in the query string with which the program was invoked are added to car in step 2, prior to generating the HTML display elements. In step 4, the form element is created using the set of possible features and the set of possible colors. These sets of attributes are stored as static data members in an external utility class, CCarAttr. Each feature corresponds to a CHTML_checkbox element in the form, and each color corresponds to a CHTML_radio button. The selected color, along with all currently selected features, will be displayed as selected in the form.

The summary statement uses a CHTML_ol list element to itemize the selected features in car. The price is calculated as CCar::m_BasePrice plus an additional $1000 per feature. The submit button generates a fresh page with the new query string, as the action attribute of the form is the URL of car.cgi.

Program design: Distributing the work

The program uses three classes: CCar, CCarAttr, and CCarCgi. The CCar class knows nothing about HTML nodes or CGI objects - its only functions are to store the currently selected color and features, and compute the resulting price:

```
class CCar
{
public:
 CCar(unsigned base_price = 12000) { m_BasePrice = base_price; }
 // Mutating member functions
 void AddFeature(const string& feature_name);
 void SetColor(const string& color_name);
 // Access member functions
 bool HasFeature(const string& feature_name) const;
 string GetColor(void) const;
 string GetPrice(void) const;
 const set<string>& GetFeatures() const;
private:
 set<string> m_Features;
 string m_Color;
 unsigned m_BasePrice;
};
```

Instead, the CCar class provides an interface to all of its data members, thus allowing the application to get/set features of the car as needed. The static utility class, CCarAttr, simply provides the sets of possible features and colors, which will be used by the application in generating the HTML form for submission:

```
class CCarAttr {
public:
 CCarAttr(void);
 static const set<string>& GetFeatures(void) { return sm_Features; }
 static const set<string>& GetColors (void) { return sm_Colors; }
private:
 static set<string> sm_Features;
 static set<string> sm_Colors;
};
```

Both of these classes are defined in a header file which is #include'd in the *.cpp files. Finally, the application class does most of the actual work, and this class must know about CCar, CCarAttr, HTML, and CGI objects. The CCarCgi class has the following interface:

```
class CCarCgi : public CCgiApplication
{
public:
 virtual int ProcessRequest(CCgiContext& ctx);
private:
 CCar* CreateCarByRequest(const CCgiContext& ctx);
 void PopulatePage(CHTMLPage& page, const CCar& car);
 static CNCBINode* ComposeSummary(const CCar& car);
 static CNCBINode* ComposeForm (const CCar& car);
 static CNCBINode* ComposePrice (const CCar& car);
 static const char sm_ColorTag[];
 static const char sm_FeatureTag[];
};
```

The source code is distributed over three files:

car.hpp

car.cpp

car_cgi.cpp

The CCar and CCarAttr classes are defined in car.hpp, and implemented in car.cpp. Both the class definition and implementation for the CGI application class are in car_cgi.cpp. With this design, only the application class will be affected by changes made to either the HTML or CGI class objects. The additional files needed to compile and run the program are:

car.html

Makefile.car_app

CGI Response Codes

Wherever possible the client when encountering errors should return an appropriate response code consisting of the three digits *DDD* . In the case of client error codes, these begin with a "4" (4xx). Table 7 contains a summary of these codes.

> Note that error code 404 should be reserved for situations when the requested file does not exist. It should not be used as a "catch-all" such as when the client simply uses bogus parameters.

Table 7. CGI Client Error Codes (4XX)

Error Code	Description
400	Bad request; the client erred in the request and should not reattempt it without modifications.
401	Unauthorized; the page is password protected but required credentials were not presented.
402	Payment required; reserved.
403	Forbidden; the client is not allowed here.
404	Not found; the requested resource (as indicated in the path) does not exist on the server, temporarily or permanently.
405	Method not allowed; the server must supply allowedrequest methods in "Allow:" HTTP header.
406	Not acceptable; content characteristics are unacceptable to produce the response.
407	Proxy authentication required; similar to 401, but for proxy.
408	Request timeout; the client does not furnish the entire request within the allotted time.
409	Conflict; usually means bad form submission via PUT method.
410	Gone; the resource is and will be no longer available and forwarding address is and will not be known.
411	Length required; the client must use content-length in the request.
412	Precondition failed; request header inquired for a condition that doesn't hold.
413	Request too big; self-explanatory.
414	Request too long; query-line element is exceeding the maximal size(but the body, if any, can be okay).
415	Unsupported media; resource does not support requested format.
416	Bad range; pertains to multi-part messages when the client requested a fragment that is out of allowed range.
417	Expectation failed; "Expect:" from the HTTP/1.1 header is not met.

FCGI Redirection and Debugging C++ Toolkit CGI Programs

Development, testing, and debugging of CGI applications can be greatly facilitated by making them FastCGI-capable and using an FCGI redirector script.

Applications that were written to use the C++ Toolkit CGI framework (see example above) can easily be made to run under your account, on your development machine, and in a number of ways (e.g. standalone, with special configuration, under a debugger, using a memory checker, using strace, etc.). This is accomplished by "tunneling" through a simple FCGI redirector script that forwards HTTP requests to your application and returns the HTTP responses.

The process is described in the following sections:

- Creating and debugging a sample FastCGI application
- Debugging an existing CGI or FCGI application

Creating and debugging a sample FastCGI application

If you are starting from scratch, use the new_project script to create a CGI that is already set up for FCGI redirection:

```
new_project foobar app/cgi
cd foobar
make
```

This creates the following files:

File	Purpose
ffoobar.fcgi	This is the FCGI version of the foobar.cgi application and should be run when debugging.
ffoobar.cgi	This is the FCGI redirector script. It is the CGI called from the browser, and it is the only file that needs to be copied to the web server for FCGI redirection.
ffoobar.ini	This can be edited as desired (e.g. setting [FCGI] Iterations = 1). Note: Unlike “plain” CGIs, FastCGI applications usually handle more than one HTTP request. While they do it sequentially, and there are no multithreading issues, additional care should still be taken to ensure that there is no unwanted interference between different HTTP requests, and that operation is correct under more strict resource (such as heap) accounting. So, to verify how a FastCGI application will work in real life you should test it with the number of iterations greater than 1.
foobar.cgi	This is the CGI application. Use it if you want to directly invoke your application as a CGI. It isn't used in the FCGI redirection process.
foobar.html	This is the web page that your CGI / FCGI applications will load. The name is the same for both CGI and FCGI. Note: By default, the HTML page must be located with your application and it must match your project name (but with the .html extension). The HTML page name can be changed in the ProcessRequest() method.

The steps to debugging your new application using FCGI redirection are:

1. Install the ffoobar.cgi redirector script on the web server.
2. Set up the application:
   a. Configure with ffoobar.ini - set [FastCGI] Iterations = 1.
   b. Set a breakpoint on ProcessRequest() and run ffoobar.fcgi under the debugger (or run under a memory checker or other tool).
3. From your web browser (or using GET/POST command-line utilities), submit a web request to ffoobar.cgi. The request/response will be tunneled to/from ffoobar.fcgi.

Debugging an existing CGI or FCGI application

The steps to debugging an existing CGI or FCGI application using FCGI redirection are (assuming the name is foobar.cgi):

1. If it's a "plain" CGI, then make it FastCGI-capable - change the makefile to build ffoobar.fcgi instead of foobar.cgi and to link with xfcgi.lib instead of xcgi.lib. Note: the application must use the C++ Toolkit's CGI framework (as in the above example).
2. Rebuild the application.
3. Install the ffoobar.cgi redirector script on the web server (in place of the existing CGI).
4. Set up the application:
   a. Copy the application ffoobar.fcgi to a development host.
   b. Configure with ffoobar.ini - set [FastCGI] Iterations = 1.
   c. Set a breakpoint on ProcessRequest() and run ffoobar.fcgi under the debugger (or run under a memory checker or other tool).
5. From your web browser (or using GET/POST command-line utilities), submit a web request to ffoobar.cgi. The request/response will be tunneled to/from ffoobar.fcgi.

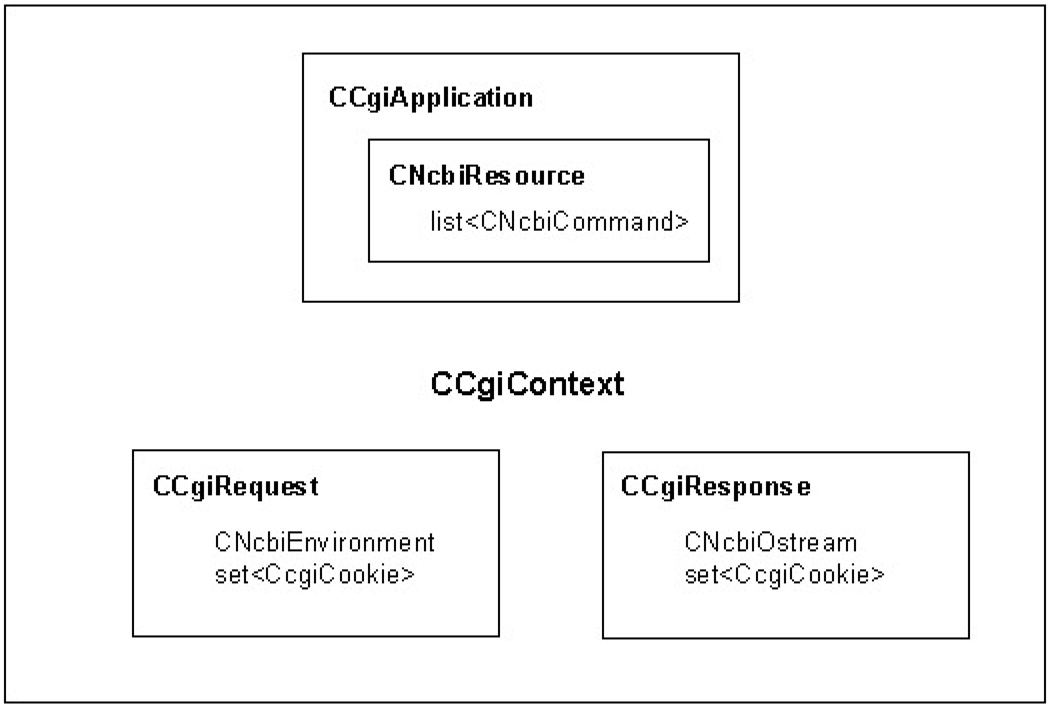


Figure 1. Layered design of the CGI classes

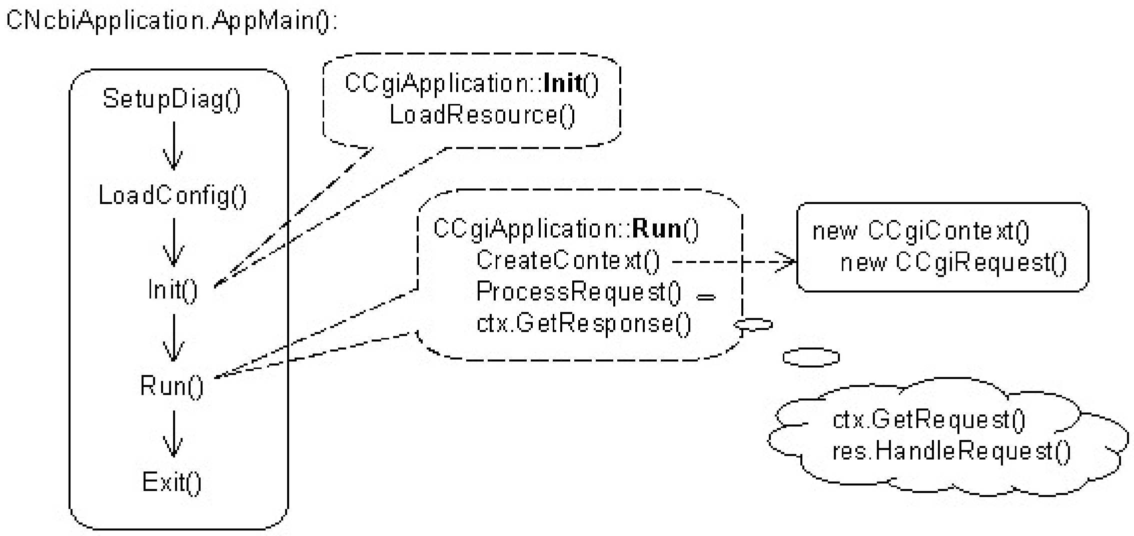


Figure 2. Adapting the init() and run() methods inherited from CNcbiApplication

Table 1. Restrictions on arguments to the CCgiCookie constructor

Field	**Restrictions**
name (required)	No spaces; must be printable ASCII; cannot contain = , or ;
value (required)	No spaces; must be printable ASCII; cannot contain , or ;
domain (optional)	No spaces; must be printable ASCII; cannot contain , or ;
path (optional)	Case sensitive

Table 2. Effect of setting the diag-destination parameter

value	effects
stderr	Send diagnostics to the standard error stream (default behavior)
asbody	Send diagnostics to the client in place of normal output

Box 1

```
// File name: justcgi.cpp
// Description: Demonstrate the basic CGI classes and functions
#include "justcgi.hpp"
#include <cgi/cgictx.hpp>
#include <corelib/ncbistd.hpp>
#include <corelib/ncbireg.hpp>
#include <memory>
USING_NCBI_SCOPE;
/////////////////////////////////////////////////////////////////////////
// Implement the application's LoadResource() and ProcessRequest() methods
CNcbiResource* CCgiApp::LoadResource(void)
{
 auto_ptr<CCgiResource> resource(new CCgiResource(GetConfig()));
 resource->AddCommand(new CCgiBasicCommand(*resource));
 resource->AddCommand(new CCgiReplyCommand(*resource));
 return resource.release();
}
// forward declarations
void ShowCommands (const TCmdList& cmds, CCgiContext& ctx);
void ShowEntries (const TCgiEntries& entries);
int CCgiApp::ProcessRequest(CCgiContext& ctx)
{
 ShowCommands (GetResource().GetCmdList(), ctx);
 ShowEntries (const_cast<TCgiEntries&>(ctx.GetRequest().GetEntries()));
 GetResource().HandleRequest(ctx);
 return 0;
}
//////////////////////////////////////////////////////////////
// Define the resource's default command if none match queryCNcbiCommand*
CCgiResource::GetDefaultCommand(void) const
{
 cerr << " executing CCgiResource::GetDefaultCommand()" << endl;
 return new CCgiBasicCommand(const_cast<CCgiResource&>(*this));
}
////////////////////////////////////////////////////////////////
// Define the Execute() and Clone() methods for the commands
void CCgiCommand::Execute(CCgiContext& ctx)
{
 cerr << " executing CCgiCommand::Execute " << endl;
 const CNcbiRegistry& reg = ctx.GetConfig();
 ctx.GetResponse().WriteHeader();
}
```

```
CNcbiCommand* CCgiBasicCommand::Clone(void) const
{
 cerr << " executing CCgiBasicCommand::Clone()" << endl;
 return new CCgiBasicCommand(GetCgiResource());
}
CNcbiCommand* CCgiReplyCommand::Clone(void) const
{
 cerr << " executing CCgiReplyCommand::Clone" << endl;
 return new CCgiReplyCommand(GetCgiResource());
}
// Show what commands have been installed
void ShowCommands (const TCmdList& cmds, CCgiContext& ctx)
{
 cerr << "Commands defined for this application are: \n";
 ITERATE(TCmdList, it, cmds) {
 cerr << (*it)->GetName();
 if ((*it)->IsRequested(ctx)) {
 cerr << " (requested)" << endl;
 } else {
 cerr << " (not requested)" << endl;
 }
 }
}
// Show the <key=value> pairs in the request string
void ShowEntries (const TCgiEntries& entries)
{
 cerr << "The entries in the request string were: \n";
 ITERATE(TCgiEntries, it, entries) {
 if (! (it->first.empty() && it->second.empty()))
 cerr << it->first << "=" << it->second << endl;
 }
}
static CCgiApp theCgiApp;
int main(int argc, const char* argv[])
{
 SetDiagStream(&cerr);
 return theCgiApp.AppMain(argc, argv);
}
```

Box 2

```
// File name: justcgi.hpp
// Description: Demonstrate the basic CGI classes and functions
#ifndef CGI_HPP
#define CGI_HPP
#include <cgi/cgiapp.hpp>
#include <cgi/ncbires.hpp>
USING_NCBI_SCOPE;
class CCgiApp : public CCgiApplication
{
public:
 virtual CNcbiResource* LoadResource(void);
 virtual int ProcessRequest(CCgiContext& context);
};
class CCgiResource : public CNcbiResource
{
public:
 CCgiResource(CNcbiRegistry& config)
 : CNcbiResource(config) {}
 virtual ~CCgiResource() {};
 // defines the command to be executed when no other command matches
 virtual CNcbiCommand* GetDefaultCommand( void ) const;
};
class CCgiCommand : public CNcbiCommand
{
public:
 CCgiCommand( CNcbiResource& resource ) : CNcbiCommand(resource) {};
 virtual ~CCgiCommand( void ) {};
 virtual void Execute( CCgiContext& ctx );
 virtual string GetLink(CCgiContext&) const { return NcbiEmptyString; }
protected:
 CCgiResource& GetCgiResource() const
 {
 return dynamic_cast<CCgiResource&>( GetResource() );
}
 virtual string GetEntry() const { return string("cmd"); }
};
class CCgiBasicCommand : public CCgiCommand
{
public:
 CCgiBasicCommand(CNcbiResource& resource) : CCgiCommand(resource) {};
 virtual ~CCgiBasicCommand(void) {};
 virtual CNcbiCommand* Clone( void ) const;
 virtual string GetName( void ) const { return string("init"); };
protected:
 virtual string GetEntry() const { return string("cmd1"); }
};
class CCgiReplyCommand : public CCgiBasicCommand
{
public:
 CCgiReplyCommand( CNcbiResource& resource) : CCgiBasicCommand(resource)
{};
 virtual ~CCgiReplyCommand(void) {};
 virtual CNcbiCommand* Clone( void ) const;
 virtual string GetName( void ) const { return string("reply"); };
protected:
 virtual string GetEntry() const { return string("cmd2"); }
};
#endif /* CGI_HPP */
```

Box 3

```
# Author: Diane Zimmerman
# Build CGI application "CGI"
# NOTE: see to build Fast-CGI
#################################
APP = cgi
OBJ = cgiapp
LIB = xcgi xncbi
```

Box 4

```
# Author: Diane Zimmerman
# Build test Fast-CGI application "FASTCGI"
# NOTES: - it will be automagically built as a plain CGI application if
# Fast-CGI libraries are missing on your machine.
# - also, it auto-detects if it is run as a FastCGI or a plain
# CGI, and behave appropriately.
#################################
APP = fastcgi
OBJ = cgiapp
LIB = xfcgi xncbi
LIBS = $(FASTCGI_LIBS) $(ORIG_LIBS)
```

12: HTML

Last Update: May 30, 2013.

The HTML API [Library xhtml: include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This C++ HTML generation API is slowly but surely going out of fashion. Nowadays, it's recommended to use mainstream XML/XSLT approach to prepare HTML pages; in particular, the XmlWrapp API.

NB Don't confuse it with the C++ CGI framework API **-- which is alive and well!**

The HTML module can be used to compose and print out a HTML page by using a static HTML template with embedded dynamic fragments. The HTML module provides a rich set of classes to help build the dynamic fragments using HTML tag nodes together with text nodes arranged into a tree-like structure.

This chapter provides reference material for many of the HTML facilities. You can also see the quick reference guide, a note about using the HTML and CGI classes together and an additional class reference document. For an overview of the HTML module please refer to the HTML section in the introductory chapter on the C++ Toolkit.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Demo Cases [src/html/demo]

Test Cases [src/html/test]

NCBI C++ HTML Classes

The NCBI C++ HTML classes are intended for use in CGI programs that generate HTML. By creating a structured method for creating HTML, these classes allow for reuse of HTML generating code and simplifies laborious tasks, such as creating and maintaining tables.

A good resource for the use of HTML is the **HTML Sourcebook** by Ian Graham.

Using these classes, the in-memory representation of an HTML page is of a graph: each element on the page can have other elements as children. For example, in

```
<HTML><BODY>hello</BODY></HTML>
```

the body tag is a child of the html tag and the text "hello" is a child of the body tag. This graph structure allows for the easy addition of components as well as reuse of code among components since they share the same base classes.

A sample program, htmldemo.cpp, can be found in internal/c++/src/html/demo.

Next, the following topics are discussed:

- Basic Classes
- Specialized Tag Classes used in Forms
- Specialized Tag Classes used in Lists
- Other Specialized Tag Classes

Basic Classes

There are several basic classes for the html library. The most basic class is CNCBINode, which is a node that knows how to contain and manipulate child CNCBINodes. Two main types of classes are derived from CNCBINode, text nodes and tag (or "element") nodes. The text nodes (CHTMLText and CHTMLPlainText) are intended to be used directly by the user, whereas the basic tag nodes (CHTMLNode, CHTMLElement, CHTMLOpenElement, and CHTMLListElement) are base classes for the nodes actually used to construct a page, such as CHTML_form.

CHTMLText and CHTMLPlainText are both used to insert text into the generated html, with the latter class performing HTML encoding before generation.

CHTMLNode is the base class for CHTMLElement (tags with close tags, like FORM), CHTMLOpenElement (tags without end tags, like BR) and CHTMLListElement (tags used in lists, like OL).

The following basic classes are discussed in more detail, next:

- CNCBINode
- CHTMLText
- CHTMLPlainText
- CHTMLNode
- CHTMLElement
- CHTMLOpenElement
- CHTMLListElement

CNCBINode

CNCBINode uses the following typedefs:**typedef list<CNCBINode*> TChildList typedef map<string, string> TAttributes**

CNCBINode* AppendChild(CNCBINode* child) Add a CNCBINode* to the end the list of child nodes. Returns *this so you can repeat the operation on the same line, e.g. Node->AppendChild(new CNCBINode)->AppendChild(new CNCBINode).

CNCBINode* AppendChild(CNodeRef& ref) Add a node by reference to the end the list of child nodes. Returns *this so you can repeat the operation on the same line.

void RemoveAllChildren(void) Removes all child nodes.

TChildList::iterator ChildBegin(void) TChildList::const_iterator ChildBegin(void) const Returns the first child.

TChildList::iterator ChildEnd(void) TChildList::const_iterator ChildEnd(void) const Returns the end of the child list (this is **not** the last child).

TChildList::iterator FindChild(CNCBINode* child) Find a particular child, otherwise return 0.

virtual CNcbiOstream& Print(CNcbiOstream& out) Create HTML from the node and all its children and send it to out. Returns a reference to out.

virtual void CreateSubNodes(void) This function is called during printing when the node has not been initialized. A newly created node is internally marked as not initialized. The intent of this function is for the user to replace it with a function that knows how to create all of the subchildren of the node. The main use of this function is in classes that define whole regions of pages.

const string& GetName(void) const void SetName(const string& namein) Get and set the name of the node.

bool HaveAttribute(const string& name) const Check for an attribute. Attributes are like the href in <a href="http://www.ncbi.nlm.nih.gov">

string GetAttribute(const string& name) const Return a copy of the attribute's value

const string* GetAttributeValue(const string& name) const Return a pointer to the attribute's value

void SetAttribute(const string& name, const string& value) void SetAttribute(const string& name) void SetAttribute(const string& name, int value) void SetOptionalAttribute(const string& name, const string& value) void SetOptionalAttribute(const string& name, bool set) void SetAttribute(const char* name, const string& value) void SetAttribute(const char* name) void SetAttribute(const char* name, int value) void SetOptionalAttribute(const char* name, const string& value) void SetOptionalAttribute(const char* name, bool set) Set an attribute. SetOptionalAttribute() only sets the attribute if value contains a string or is true.

CHTMLText

CHTMLText(const string& text)

This is a text node that can contain html tags, including tags of the form <@...@> which are replaced by CNCBINode's when printing out (this is discussed further in the CHTMLPage documentation).

const string& GetText(void) const void SetText(const string& text) Get and set the text in the node.

CHTMLPlainText

CHTMLPlainText(const string& text)

This node is for text that is to be HTML encoded. For example, characters like "&" are turned into "&amp;"

const string& GetText(void) const void SetText(const string& text)

Get and set text in the node.

CHTMLNode

CHTMLNode inherits from CNCBINode is the base class for html tags.

CHTMLNode* SetWidth(int width) CHTMLNode* SetWidth(const string& width) CHTMLNode* SetHeight(int height) CHTMLNode* SetHeight(const string& width) CHTMLNode* SetAlign(const string& align) CHTMLNode* SetBgColor(const string& color) CHTMLNode* SetColor(const string& color) Sets various attributes that are in common for many tags. Avoid setting these on tags that do not support these attributes. Returns *this so that the functions can be daisy chained:

```
CHTML_table * Table = new CHTML_table;
Table->SetWidth(400)->SetBgColor("#FFFFFF");
```

void AppendPlainText(const string &) Appends a CHTMLPlainText node. A plain text node will be encoded so that it does not contain any html tags (e.g. "<" becomes "<").

void AppendHTMLText(const string &) Appends a CHTMLTextNode. This type of node can contain HTML tags, i.e. it is not html encoded.

CHTMLElement

CHTMLElement is the base class for several tags that have the constructors with the common form:**CHTMLElement() CHTMLElement(CNCBINode* node) CHTMLElement(const string& text)** The second constructor appends node. The third constructor appends CHTMLText(const string& text).

The tags derived from this class include: CHTML_html, CHTML_head, CHTML_body, CHTML_base, CHTML_isindex, CHTML_link, CHTML_meta, CHTML_script, CHTML_style, CHTML_title, CHTML_address, CHTML_blockquote, CHTML_center, CHTML_div, CHTML_h1, CHTML_h2, CHTML_h3, CHTML_h4, CHTML_h5, CHTML_h6, CHTML_hr, CHTML_p, CHTML_pre, CHTML_dt, CHTML_dd, CHTML_li, CHTML_caption, CHTML_col, CHTML_colgroup, CHTML_thead, CHTML_tbody, CHTML_tfoot, CHTML_tr, CHTML_th, CHTML_td, CHTML_applet, CHTML_param, CHTML_cite, CHTML_code, CHTML_dfn, CHTML_em, CHTML_kbd, CHTML_samp, CHTML_strike, CHTML_strong, CHTML_var, CHTML_b, CHTML_big, CHTML_i, CHTML_s, CHTML_small, CHTML_sub, CHTML_sub, CHTML_sup, CHTML_tt, CHTML_u, CHTML_blink, CHTML_map, CHTML_area

CHTMLOpenElement

This is used for tags that do not have a close tag (like img). The constructors are of the same form as CHTMLElement. The tags derived from this class include: CHTML_pnop (paragraph tag without a close tag)

CHTMLListElement

These are elements used in a list.

CHTMLListElement(void) CHTMLListElement(bool compact) CHTMLListElement (const string& type) CHTMLListElement(const string& type, bool compact) Construct the ListElement with the given attibutes: TYPE and COMPACT. Both attributes affect the way the ListElement is displayed.

CHTMLListElement* AppendItem(const string& item) CHTMLListElement* AppendItem(CNCBINode* item) These functions add CHTMLText and CNCBINode items as children of the CHTMLListElement. The tags derived from this class include: CHTML_ul, CHTML_dir, CHTML_menu.

Specialized Tag Classes used in Forms

The rest of the sections deal with tag classes that have additional members or member functions that make the tags easier to use. In addition there are helper classes, such as CHTML_checkbox, that are easier to use instances of HTML tags.

The following specialized tag classes used in forms are discussed, next:

- CHTML_form: derived from CHTMLElement
- CHTML_input: derived from CHTMLOpenElement
- CHTML_checkbox: derived from CHTML_input
- CHTML_hidden: derived from CHTML_input
- CHTML_image: derived from CHTML_input
- CHTML_radio: derived from CHTML_input
- CHTML_reset: derived from CHTML_input
- CHTML_submit: derived from CHTML_input
- CHTML_text: derived from CHTML_input
- CHTML_select: derived from CHTMLElement
- CHTML_option: derived from CHTMLElement
- CHTML_textarea: derived from CHTMLElement

CHTML_form: derived from CHTMLElement

CHTML_form(const string& action = NcbiEmptyString, const string& method = NcbiEmptyString, const string& enctype = NcbiEmptyString) Add an HTML form tag with the given attributes. NCBIEmptyString is simply a null string.

void AddHidden(const string& name, const string& value) Add a hidden value to the form.

CHTML_input: derived from CHTMLOpenElement

CHTML_input(const string& type, const string& name) Create a input tag of the given type and name. Several of the following classes are specialized versions of the input tag, for example, CHTML_checkbox.

CHTML_checkbox: derived from CHTML_input

CHTML_checkbox(const string& name) CHTML_checkbox(const string& name, bool checked, const string& description = NcbiEmptyString) CHTML_checkbox(const

string& name, const string& value) CHTML_checkbox(const string& name, const string& value, bool checked, const string& description = NcbiEmptyString) Create a checkbox with the given attributes. This is an input tag with type = "checkbox".

CHTML_hidden: derived from CHTML_input

CHTML_hidden(const string& name, const string& value) Create a hidden value with the given attributes. This is an input tag with type = "hidden".

CHTML_image: derived from CHTML_input

CHTML_image(const string& name, const string& src) Create an image submit input tag. This is an input tag with type = "image".

CHTML_radio: derived from CHTML_input

CHTML_radio(const string& name, const string& value) CHTML_radio(const string& name, const string& value, bool checked, const string& description = NcbiEmptyString) Creates a radio button. Radio buttons are input tags with type = "radio button".

CHTML_reset: derived from CHTML_input

CHTML_reset(const string& label = NcbiEmptyString) Create a reset button. This is an input tag with type = "reset".

CHTML_submit: derived from CHTML_input

CHTML_submit(const string& name) CHTML_submit(const string& name, const string& label) Create a submit button. This is an input tag with type = "submit".

CHTML_text: derived from CHTML_input

CHTML_text(const string& name, const string& value = NcbiEmptyString) CHTML_text(const string& name, int size, const string& value = NcbiEmptyString) CHTML_text(const string& name, int size, int maxlength, const string& value = NcbiEmptyString) Create a text box. This is an input tag with type = "text".

CHTML_select: derived from CHTMLElement

CHTML_select(const string& name, bool multiple = false) CHTML_select(const string& name, int size, bool multiple = false) Create a selection tag used for drop-downs and selection boxes.

CHTML_select* AppendOption(const string& option, bool selected = false) CHTML_select* AppendOption(const string& option, const string& value, bool selected = false) Add an entry to the selection box by using the option tag. Returns *this to allow you to daisy-chain calls to AppendOption().

CHTML_option: derived from CHTMLElement

CHTML_option(const string& content, bool selected = false) CHTML_option(const string& content, const string& value, bool selected = false) The option tag used inside of select elements. See CHTML_select for an easy way to add option.

CHTML_textarea: derived from CHTMLElement

CHTML_textarea(const string& name, int cols, int rows) CHTML_textarea(const string& name, int cols, int rows, const string& value)

Create a textarea tag inside of a form.

Specialized Tag Classes used in Lists

These are specialized tag classes used in lists. See "Basic Classes" for non-specialized tag classes used in list.

The following specialized tag classes used in lists are discussed, next:

- CHTML_dl: derived from CHTMLElement
- CHTML_ol: derived from CHTMLListElement

CHTML_dl: derived from CHTMLElement

CHTML_dl(bool compact = false) Create a dl tag.

CHTML_dl* AppendTerm(const string& term, CNCBINode* definition = 0) CHTML_dl* AppendTerm(const string& term, const string& definition) CHTML_dl* AppendTerm(CNCBINode* term, CNCBINode* definition = 0) CHTML_dl* AppendTerm(CNCBINode* term, const string& definition) Append a term and definition to the list by using DD and DT tags.

CHTML_ol: derived from CHTMLListElement

CHTML_ol(bool compact = false) CHTML_ol(const string& type, bool compact = false) CHTML_ol(int start, bool compact = false) CHTML_ol(int start, const string& type, bool compact = false) The last two constructors let you specify the starting number for the list.

Other Specialized Tag Classes

These tag classes that have additional members or member functions that make the tags easier to use. The following classes are discussed next:

- CHTML_table: derived from CHTMLElement
- CHTML_a: derived from CHTMLElement
- CHTML_img: derived from CHTMLOpenElement
- CHTML_font: derived from CHTMLElement
- CHTML_color: derived from CHTMLElement
- CHTML_br: derived from CHTMLOpenElement
- CHTML_basefont: derived from CHTMLElement

CHTML_table: derived from CHTMLElement

CNCBINode* Cell(int row, int column) This function can be used to specify the size of the table or return a pointer to a particular cell in the table. Throws a runtime_error exception when the children of the table are not TR or the children of each TR is not TH or TD or there are more columns than should be.

int CalculateNumberOfColumns(void) const int CalculateNumberOfRows(void) const Returns number of columns and number of rows in the table.

CNCBINode* InsertAt(int row, int column, CNCBINode* node) CNCBINode* InsertTextAt(int row, int column, const string& text) Inserts a node or text in the table. Grows the table if the specified cell is outside the table. Uses Cell() so can throw the same exceptions.

void ColumnWidth(CHTML_table*, int column, const string & width) Set the width of a particular column.

CHTML_table* SetCellSpacing(int spacing) CHTML_table* SetCellPadding(int padding) Set the cellspacing or cellpadding attributes.

CHTML_a: derived from CHTMLElement

CHTML_a(const string& href, const string& text) CHTML_a(const string& href, CNCBINode* node) Creates a hyperlink that contains the given text or node.

CHTML_img: derived from CHTMLOpenElement

CHTML_img(const string& url) CHTML_img(const string& url, int width, int height) Creates an image tag with the given attributes.

CHTML_font: derived from CHTMLElement

CHTML_font(void) CHTML_font(int size, CNCBINode* node = 0) CHTML_font(int size, const string& text) CHTML_font(int size, bool absolute, CNCBINode* node = 0) CHTML_font(int size, bool absolute, const string& text) CHTML_font(const string& typeface, CNCBINode* node = 0) CHTML_font(const string& typeface, const string& text) CHTML_font(const string& typeface, int size, CNCBINode* node = 0) CHTML_font(const string& typeface, int size, const string& text) CHTML_font(const string& typeface, int size, bool absolute, CNCBINode* node = 0) CHTML_font(const string& typeface, int size, bool absolute, const string& text) Create a font tag with the given attributes. Appends the given text or node. Note that it is cleaner and more reusable to use a stylesheet than to use the font tag.

void SetRelativeSize(int size) Set the size of the font tag.

CHTML_color: derived from CHTMLElement

CHTML_color(const string& color, CNCBINode* node = 0) CHTML_color(const string& color, const string& text) Create a font tag with the given color and append either node or text.

CHTML_br: derived from CHTMLOpenElement

CHTML_br(void) CHTML_br(int number) The last constructor lets you insert multiple BR tags.

CHTML_basefont: derived from CHTMLElement

CHTML_basefont(int size) CHTML_basefont(const string& typeface) CHTML_basefont(const string& typeface, int size) Set the basefont for the page with the given attributes.

Generating Web Pages with the HTML classes

Web applications involving interactions with a client via a complex HTML interface can be difficult to understand and maintain. The NCBI C++ Toolkit classes decouple the complexity of interacting with a CGI client from the complexity of generating HTML output by defining separate class hierarchies for these activities. In fact, one useful application of the HTML classes is to generate web pages "offline".

The chapter on Developing CGI Applications discussed only the activities involved in processing the client's request and generating a response. This section introduces the C++ Toolkit components that support the creation of HTML pages, and concludes with a brief consideration of how the HTML classes can be used in consort with a running CCgiApplication. Further discussion of combining a CGI application with the HTML classes can be found in the section on An example web-based CGI application. See also NCBI C++ HTML Classes in the Reference Manual.

The following topics are discussed in this section:

The CNCBINode (*) class

All of the HTML classes are derived from the CNCBINode class, which in turn, is derived from the CObject class. Much of the functionality of the many derived subclasses is implemented by the CNCBINode base class. The CNCBINode class has just three data members:

- m_Name - a string, used to identify the type of node or to store text data
- m_Attributes - a map<string, string> of properties for this node
- m_Children - a list of subnodes embedded (at run-time) in this node

The m_Name data member is used differently depending on the type of node. For HTML text nodes, m_Name stores the actual body of text. For CHTMLElement objects, m_Name stores the HTML tagname that will be used in generating HTML formatted output.

The m_Attributes data member provides for the encoding of specific features to be associated with the node, such as background color for a web page. A group of "Get/SetAttribute" member functions are provided for access and modification of the node's attributes. All of the "SetAttribute" methods return this - a pointer to the HTML node being operated on, and so, can be daisy-chained, as in:

```
table->SetCellSpacing(0)->SetBgColor("CCCCCC");
```

Care must be taken however, in the order of invocations, as the object type returned by each operation is determined by the class in which the method is defined. In the above example, table is an instance of CHTML_table, which is a subclass of CNCBINode - where SetBgColor () is defined. The above expression then, effectively executes:

```
table->SetCellSpacing(0);
table->SetBgColor("CCCCCC");
```

In contrast, the expression:

```
table->SetBgColor("CCCCCC")->SetCellSpacing(0);
```

would fail to compile, as it would effectively execute:

```
table->SetBgColor("CCCCCC");
(CNCBINode*)table->SetCellSpacing(0);
```

since the method SetCellSpacing() is undefined for CNCBINode() objects.

The m_Children data member of CNCBINode stores a dynamically allocated list of CNCBINode subcomponents of the node. In general, the in memory representation of each node is a graph of CNCBINode objects (or subclasses thereof), where each object may in turn contain additional CNCBINode children. For example, an unordered list is represented as a CHTML_ul (<ul>) element containing CHTML_li (<li>) subcomponents.

A number of member functions are provided to operate on m_Children. These include methods to access, add, and remove children, along with a pair of begin/end iterators (ChildBegin() and ChildEnd()), and a function to dereference these iterators (Node(i)).

Depending on flags set at compile time, m_Children is represented as either a list of CNodeRef objects, or a list of auto_ptr<CNodeRef>, where CNodeRef is a typedef for CRef<CNCBINode>. This distinction is transparent to the user however, and the important point is that the deallocation of all dynamically embedded child nodes is handled automatically by the containing class.

CNCBINode::Print() recursively generates the HTML text for the node and all of its children, and outputs the result to a specified output stream. The Print() function takes two arguments: (1) an output stream, and (2) a CNCBINode::TMode object, where TMode is an internal class defined inside the CNCBINode class. The TMode object is used by the print function to determine what type of encoding takes place on the output, and in some cases, to locate the containing parent node.

Many of the CNCBINode objects do not actually allocate their embedded subnodes until the Print() method is invoked. Instead, a kind of lazy evaluation is used, and the information required to install these nodes to m_Children is used by the CreateSubNodes() method only when output has been requested (see discussion below).

A slice of the NCBI C++ Toolkit class hierarchy rooted at the CNCBINode class includes the following directly derived subclasses:

- CNCBINode:
  - CSmallPagerBox
  - CSelection
  - CPagerBox
  - CPager
  - CHTMLText
  - CHTMLTagNode
  - CHTMLPlainText
  - CHTMLNode
  - CHTMLDualNode
  - CHTMLBasicPage

   - CButtonList

Many of these subclasses make little sense out of context, as they are designed for use as subcomponents of, for example, a CHTMLPage. Exceptions to this are the text nodes, described next.

HTML Text nodes: CHTMLText (*) and CHTMLPlainText (*)

The CHTMLText class uses the m_Name data member (inherited from CNCBINode) to store a text string of arbitrary length. No new data members are introduced, but two new member functions are defined. SetText() resets m_Name to a new string, and GetText() returns the value currently stored in m_Name. With the exception of specially tagged sections (described below), all text occurring in a CHTMLText node is sent directly to the output without further encoding.

The CHTMLPlainText class is provided for text that may require further encoding. In addition to the SetText() and GetText() member functions described for the CHTMLText class, one new data member is introduced. m_NoEncode is a Boolean variable that designates whether or not the text should be further encoded. NoEncode() and SetNoEncode() allow for access and modification of this private data member. For example:

```
(new CHTMLText("<br> testing BR <br>"))->Print(cout);
```

will generate the output:

```
testing BR
```

whereas:

```
(new CHTMLPlainText("<br> testing BR <br>"))->Print(cout);
```

will generate:

```
<br> testing BR <br>
```

The text in the CHTMLText node is output verbatim, and the web browser interprets the <br> tags as line breaks. In contrast, the CHTMLPlainText node effectively "insulates" its content from the browser's interpretation by encoding the <br> tags as "<br&gt;".

CHTMLText nodes also play a special role in the implementation of page nodes that work with template files. A tagname in the text is delimited by "<@" and "@>", as in: <@tagname@>. This device is used for example, when working with template files, to allow additional nodes to be inserted in a pre-formatted web page. The CHTMLText::PrintBegin() method is specialized to skip over the tag names and their delimiters, outputting only the text generated by the nodes that should be inserted in that tagged section. Further discussion of this feature is deferred until the section on the NCBI page classes, which contain a TTagMap.

The NCBI Page classes

The page classes serve as generalized containers for collections of other HTML components, which are mapped to the page by a tagmap. In general, subcomponents are added to a page using the AddTagMap() method (described below), instead of the AppendChild() method. The page classes define the following subtree in the C++ Toolkit class hierarchy:

- CHTMLBasicPage
   - CHTMLPage

In addition to the data members inherited from CNCBINode, three new private data members are defined in the CHTMLBasicPage class.

- m_CgiApplication - a pointer to the CCgiApplication
- m_Style - an integer flag indicating subcomponents to display/suppress (e.g., Title)
- m_TagMap (see discussion)

In effect, m_TagMap is used to map strings to tagged subcomponents of the page - some of which may not have been instantiated yet. Specifically, m_TagMap is defined as a TTagMap variable, which has the following type definition:

```
typedef map<string, BaseTagMapper*> TTagMap;
```

Here, BaseTagMapper is a base class for a set of functor-like structs. Each of the derived subclasses of BaseTagMapper has a single data member (e.g. m_Node, m_Function or m_Method), which points to either a CNCBINode, or a function that returns a pointer to a CNCBINode. The BaseTagMapper class also has a single member function, MapTag(), which knows how to "invoke" its data member.

The simplest subclass of BaseTagMapper is the ReadyTagMapper class whose sole data member, m_Node, is a CRef pointer to a CNCBINode. In this case the MapTag() function simply returns &*m_Node. Several different types of tagmappers are derived from the BaseTagMapper class in nodemap.hpp. Each of these subclasses specializes a different type of data member, which may be a pointer to a free function, a pointer to a member function, or a pointer to an object, as in the case of the ReadyTagMapper. The action taken by the tagmapper's MapTag() method in order to return a pointer to a CNCBINode is implemented accordingly.

The CHTMLBasicPage class also has a member function named MapTag(), which is used in turn, to invoke a tagmapper's MapTag() method. Specifically, CHTMLBasicPage::MapTag (tagname) first locates the installed tagmapper associated with tagname, m_TagMap[tagname]. If an entry is found, that tagmapper's MapTag() member function is then invoked, which finally returns a pointer to a CNCBINode.

A second member function, CHTMLBasicPage::AddTagMap(str, obj), provides for the insertion of a new tag string and its associated tagmapper struct to m_TagMap. Depending on the object type of the second argument, a type-specific implementation of an overloaded helper function, CreateTagMapper(), can be used to install the desired tagmapper.

In order for a new mapping to have any effect however, the tag must also occur in one of the nodes installed as a child of the page. This is because the Print() methods for the page nodes do virtually nothing except invoke the Print() methods for m_Children. The m_TagMap data member, along with all of its supporting methods, is required for the usage of template files, as described in the next section.

The primary purpose of the CHTMLBasicPage is as a base class whose features are inherited by the CHTMLPage class - it is not intended for direct usage. Important inherited features include its three data members: m_CgiApplication, m_Style, and m_TagMap, and its member functions: Get/SetApplication(), Get/SetStyle(), MapTag(), and AddTagMap(). Several of the more advanced HTML components generate their content via access of the running CGI application. For example, see the description of a CSelection node. It is not strictly necessary to specify a CGI application when instantiating a page object however, and constructors are available that do not require an application argument.

Using the CHTMLPage class with Template Files

The CHTMLPage class is derived from the CHTMLBasicPage. In combination with the appropriate template file, this class can be used to generate the standard NCBI web page, which includes:

- the NCBI logo
- a hook for the application-specific logo
- a top menubar of links to several databases served by the query program
- a links sidebar for application-specific links to relevant sites
- a VIEW tag for the application's web interface
- a bottom menubar for help links, disclaimers, etc.

The template file is a simple HTML text file with one extension -- the use of named tags (<@tagname@>) which allow the insertion of new HTML blocks into a pre-formatted page. The standard NCBI page template file contains one such tag, VIEW.

The CHTMLPage class introduces two new data members: m_Title (string), which specifies the title for the page, and m_TemplateFile (string), which specifies a template file to load. Two constructors are available, and both accept string arguments that initialize these two data members. The first takes just the title name and template file name, with both arguments being optional. The other constructor takes a pointer to a CCgiApplication and a style (type int), along with the title and template_file names. All but the first argument are optional for the second constructor. The member functions, SetTitle() and SetTemplateFile(), allow these data members to be reset after the page has been initialized.

Five additional member functions support the usage of template files and tagnodes as follows:

- CreateTemplate() reads the contents of file m_TemplateFile into a CHTMLText node, and returns a pointer to that node.
- CreateSubNodes() executes AppendChild(CreateTemplate()), and is called at the top of Print() when m_Children is empty. Thus, the contents of the template file are read into the m_Name data member of a CHTMLText node, and that node is then installed as a child in the page's m_Children.
- CreateTitle() returns new CHTMLText(m_Title).
- CreateView() is effectively a virtual function that must be redefined by the application. The CHTMLPage class definition returns a null pointer (0).
- Init() is called by all of the CHTMLPage constructors, and initializes m_TagMap as follows:
  ```
  void CHTMLPage::Init(void)
  {
  AddTagMap("TITLE", CreateTagMapper(this, &CHTMLPage::CreateTitle));
  AddTagMap("VIEW", CreateTagMapper(this, &CHTMLPage::CreateView));
  }
  ```
  As described in the preceding section, CreateTagMapper() is an overloaded function that creates a tagmapper struct. In this case, CreateTitle() and CreateView() will be installed as the m_Method data members in the resulting tagmappers. In general, the type of struct created by CreateTagMapper depends on the argument types to that function. In its usage here, CreateTagMapper is a template function, whose arguments

are a pointer to an object and a pointer to a class method:
template<class C>
BaseTagMapper* CreateTagMapper(const C*, CNCBINode* (C::*method)(void)) {
return new TagMapper<C>(method);
}
The value returned is itself a template object, whose constructor expects a pointer to a method (which will be used as a callback to create an object of type C). Here, AddTagMap() installs CreateTitle() and CreateView() as the data member for the tagmapper associated with tag "TITLE" and tag "VIEW", respectively.

An example using the NCBI standard template file should help make these concepts more concrete. The following code excerpt uses the standard NCBI template and inserts a text node at the VIEW tag position:

```
#include <html/html.hpp>
#include <html/page.hpp>
USING_NCBI_SCOPE;
int main()
{
 try {
 CHTMLPage *Page = new CHTMLPage("A CHTMLPage!", "ncbi_page.html");
 Page->AddTagMap( "VIEW",
 new CHTMLText("Insert this string at VIEW tag"));
 Page->Print(cout);
 cout.flush();
 return 0;
 }
 catch (exception& exc) {
 NcbiCerr << "\n" << exc.what() << NcbiEndl;
 }
 return 1;
}
```

The name of the template file is stored in m_TemplateFile, and no further action on that file will be taken until Page->Print(cout) is executed. The call to AddTagMap() is in a sense then, a forward reference to a tag that we know is contained in the template. Thus, although a new CHTMLText node is instantiated in this statement, it is not appended to the page as a child, but is instead "mapped" to the page's m_TagMap where it is indexed by "VIEW".

The contents of the template file will not be read until Print() is invoked. At that time, the text in the template file will be stored in a CHTMLText node, and when that node is in turn printed, any tag node substitutions will then be made. More generally, nodes are not added to the page's m_Children graph until Print() is executed. At that time, CreateSubNodes() is invoked if m_Children is empty. Finally, the actual mapping of a tag (embedded in the template) to the associated TagMapper in m_TagMap, is executed by CHTMLText::PrintBegin().

The CHTMLPage class, in combination with a template file, provides a very powerful and general method for generating a "boiler-plate" web page which can be adapted to application-specific needs using the CHTMLPage::AddTagMap() method. When needed, The user can edit the template file to insert additional <@tagname@> tags. The AddTagMap() method is defined **only** for page objects however, as they are the only class having a m_TagMap data member.

Before continuing to a general discussion of tagnodes, let's review how the page classes work in combination with a template file:

- A page is first created with a title string and a template file name. These arguments are stored directly in the page's data members, m_Title and m_TemplateFile.
- The page's Init() method is then called to establish tagmap entries for "TITLE" and "VIEW" in m_TagMap.
- Additional HTML nodes which should be added to this page are inserted using the page's AddTagMap(tagname, *node) method, where the string tagname appears in the template as "<@tagname@>". Typically, a CGI application defines a custom implementation of the CreateView() method, and installs it using AddTagMap ("VIEW", CreateView()).
- When the page's Print() method is called, it first checks to see if the page has any child nodes, and if so, assumes there is no template loaded, and simply calls PrintChildren (). If there are no children however, page->CreateSubNodes() is called, which in turn calls the CreateTemplate() method. This method simply reads the contents of the template file and stores it directly in a CHTMLText node, which is installed as the only child of the parent page.
- The page's Print() method then calls PrintChildren(), which (eventually) causes CHTMLText::PrintBegin() to be executed. This method in turn, encodes special handling of "<@tagname@>" strings. In effect, it repeatedly outputs all text up to the first "@" character; extracts the tagname from the text; searches the parent page's m_TagMap to find the TagMapper for that tagname, and finally, calls Print() on the HTML node returned by the TagMapper. CHTMLText::PrintBegin() continues in this fashion until the end of its text is reached.

NOTE: appending any child nodes directly to the page prior to calling the Print() method will make the template effectively inaccessible, since m_Children() will not be empty. For this reason, the user is advised to use AddTagNode() rather than AppendChild() when adding subcomponents.

The CHTMLTagNode (*) class

The objects and methods described to this point provide no mechanisms for dynamically adding tagged nodes. As mentioned, the user is free to edit the template file to contain additional <@tag@> names, and AddTagMap() can then be used to associate tagmappers with these new tags. This however, requires that one know ahead of time how many tagged nodes will be used. The problem specifically arises in the usage of template files, as it is not possible to add child nodes directly to the page without overriding the the template file.

The CHTMLTagNode class addresses this issue. Derived directly from CNCBINode, the class's constructor takes a single (string or char*) argument, tagname, which is stored as m_Name. The CHTMLTagNode::PrintChildren() method is specialized to handle tags, and makes a call to MapTagAll(GetName(), mode). Here, GetName() returns the m_Name of the CHTMLTagNode, and mode is the TMode argument that was passed in to PrintChildren(). In addition to an enumeration variable specifying the mode of output, a TMode object has a pointer to the parent node that invoked PrintChildren(). This pointer is used by MapTagAll(), to locate a parent node whose m_TagMap has an installed tagmapper for the tagname. The TMode object's parent pointer essentially implements a stack which can be used to retrace the dynamic chain of PrintChildren() invocations, until either a match is found or the end of the call stack is reached. When a match is found, the associated tagmapper's MapTag() method is invoked, and Print() is applied to the node returned by this function.

The following example uses an auxillary CNCBINode(tagHolder) to install additional CHTMLTagNode objects. The tags themselves however, are installed in the containing page's m_TagMap, where they will be retrieved by the MapTagAll() function, when PrintChildren() is called for the auxillary node. That node in turn, is mapped to the page's VIEW tag. When the parent page is "printed", CreateSubNodes() will create a CHTMLText node. The text node will hold the contents of the template file and be appended as a child to the page. When PrintBegin() is later invoked for the text node, MapTagAll() associates the VIEW string with the CNCBINode, and in turn, calls Print() on that node.

```
#include <html/html.hpp>
#include <html/page.hpp>
USING_NCBI_SCOPE;
int main()
{
 try {
 CHTMLPage *Page = new CHTMLPage("myTitle", "ncbi_page.html");
 CNCBINode *tagHolder = new CNCBINode();
 Page->AddTagMap( "VIEW", tagHolder);
 tagHolder->AppendChild(new CHTMLTagNode("TAG1"));
 tagHolder->AppendChild(new CHTML_br());
 tagHolder->AppendChild(new CHTMLTagNode("TAG2"));
 Page->AddTagMap( "TAG1",
 new CHTMLText("Insert this string at TAG1"));
 Page->AddTagMap( "TAG2",
 new CHTMLText("Insert another string at TAG2"));
 Page->Print(cout);
 cout.flush();
 return 0;
 }
 catch (exception& exc) {
 NcbiCerr << "\n" << exc.what() << NcbiEndl;
 }
 return 1;
}
```

The CHTMLNode (*) class

CHTMLNode is derived directly from the CNCBINode class, and provides the base class for all elements requiring HTML tags (e.g., <ul>,<br>, <img>, <table>, etc.). The class interface includes several constructors, all of which expect the first argument to specify the HTML tagname for the node. This argument is used by the constructor to set the m_Name data member. The optional second argument may be either a text string, which will be appended to the node using AppendPlainText(), or a CNCBINode, which will be appended using AppendChild().

A uniform system of class names is applied; each subclass derived from the CHTMLNode base class is named CHTML_[tag], where [tag] is the HTML tag in lowercase, and is always preceded by an underscore. The NCBI C++ Toolkit hierarchy defines roughly 40 subclasses of CHTMLNode - all of which are defined in the Quick Reference Guide at the end of this section. The constructors for "empty" elements, such as CHTML_br, which have no assigned values, are simply invoked as CHTML_br(). The Quick Reference Guide provides brief explanations of each class, along with descriptions of the class constructors.

In addition to the subclasses explicitly defined in the hierarchy, a large number of lightweight subclasses of CHTMLNode are defined by the preprocessor macro DECLARE_HTML_ELEMENT(Tag, Parent) defined in html.hpp. All of these elements have the same interface as other CHTMLNode classes however, and the distinction is invisible to the user.

A rich interface of settable attributes is defined in the base class, and is applicable to all of the derived subclasses, including those implemented by the preprocessor macros. Settable attributes include: class, style, id, width, height, size, alignment, color, title, accesskey, and name. All of the SetXxx() functions which set these attributes return a this pointer, cast as CHTMLNode*.

The CHTMLDualNode (*) class

CHTMLDualNode is derived directly from the CNCBINode class, and provides the base class for all elements requiring different means for displaying data in eHTML and ePlainText modes.

This class interface includes several constructors. The second argument in these constructors specifies the alternative text to be displayed in ePlainText mode. The first argument of these constructors expects HTML text or pointer to an object of (or inherited from) CNCBINode class. It will be appended to the node using AppendChild() method, and printed out in eHTML mode. For example:

```
(new CHTMLDualNode(new CHTML_p("text"),"\nTEXT \n"))->Print(cout);
```

will generate the output:

```
<p>text</p>
```

whereas:

```
(new CHTMLDualNode(new CHTML_p("text"),"\n TEXT \n"))
->Print(cout, CNCBINode::ePlainText);
```

will generate:

```
\n TEXT \n
```

Using the HTML classes with a CCgiApplication object

The previous chapter described the NCBI C++ Toolkit's CGI classes, with an emphasis on their independence from the HTML classes. In practice however, a real application must employ both types of objects, and they must communicate with one another. The only explicit connection between the CGI and HTML components is in the HTML page classes, whose constructors accept a CCgiApplication as an input parameter. The open-ended definition of the page's m_TagMap data member also allows the user to install tagmapper functions that are under control of the application, thus providing an "output port" for the application. In particular, an application-specific CreateView() method can easily be installed as the function to be associated with a page's VIEW tag. The CGI sample program provides a simple example of using these classes in coordination with each other.

Supplementary Information

The following topics are discussed in this section:

The CNCBINode::TMode class

TMode is an internal class defined inside the CNCBINode class. The TMode class has three data members defined:

- EMode m_Mode - an enumeration variable specifying eHTML (0) or ePlainText (1) output encoding
- CNCBINode* m_Node - a pointer to the CNCBINode associated with this TMode object
- TMode* m_Previous - a pointer to the TMode associated with the parent of m_Node

Print() is implemented as a recursive function that allows the child node to dynamically "inherit" its mode of output from the parent node which contains it. Print() outputs the current node using PrintBegin(), recursively prints the child nodes using PrintChildren(), and concludes with a call to PrintEnd(). TMode objects are created dynamically as needed, inside the Print() function. The first call to Print() from say, a root Page node, generally specifies the output stream only, and uses a default eHTML enumeration value to initialize a TMode object. The TMode constructor in this case is:

```
TMode(EMode m = eHTML): m_Mode(m), m_Node(0), m_Previous(0) {}
```

The call to Print() with no TMode argument automatically calls this default constructor to create a TMode object which will then be substituted for the formal parameter prev inside tbe Print() method. One way to think of this is that the initial print call - which will ultimately be propagated to all of the child nodes - is initiated with a "null parent" TMode object that only specifies the mode of output.

```
CNcbiOstream& CNCBINode::Print(CNcbiOstream& os, TMode prev)
{
 // ...

 TMode mode(&prev, this);

 PrintBegin(os, mode);
 try {
 PrintChildren(out, mode);
 }
 catch (...) {
 // ...
 }
 PrintEnd(os, mode); }
```

In the first top-level call to Print(), prev is the default TMode object described above, with NULL values for m_Previous and m_Node. In the body of the Print() method however, a new TMode is created for subsequent recursion, with the following constructor used to create the new TMode at that level:

```
TMode(const TMode* M, CNCBINode* N) : m_Mode(M->m_Mode),m_Node(N),
m_Previous(M) {}
```

where M is the TMode input parameter, and N is the current node.

Thus, the output encoding specified at the top level is propagated to the PrintXxx() methods of all the child nodes embedded in the parent. The CNCBINode::PrintXxx() methods essentially do nothing;PrintBegin() and PrintEnd() simply return 0, and PrintChildren() just calls Print() on each child. Thus, the actual printing is implemented by the PrintBegin() and PrintEnd() metwebpgs.html_CHTMLBasicPaghods that are specialized by the child objects.

As the foregoing discussion implies, a generic CNCBINode which has no children explicitly installed will generate no output. For example, a CHTMLPage object which has been initialized by loading a template file has no children until they are explicitly created. In this case, the Print () method will first call CreateSubNodes() before executing PrintChildren(). The use of template files, and the associated set of TagMap functions are discussed in the section on the NCBI Page classes.

Quick Reference Guide

The following is a quick reference guide to the HTML and related classes:

- CNCBINode
  - CButtonList
  - CHTMLBasicPage
    - CHTMLPage
- CHTMLNode
  - CHTMLComment
  - CHTMLOpenElement
    - CHTML_br
    - CHTML_hr
    - CHTML_img
    - CHTML_input
      - CHTML_checkbox
      - CHTML_file
      - CHTML_hidden
      - CHTML_image
      - CHTML_radio
      - CHTML_reset
      - CHTML_submit
      - CHTML_text
- CHTMLElement
  - CHTML_a
  - CHTML_basefont CHTML_button
  - CHTML_dl
  - CHTML_fieldset
  - CHTML_font

        - CHTML_color
- CHTML_form
- CHTML_label
- CHTML_legend
- CHTML_option
- CHTML_select
- CHTML_table
    - CPageList
    - CPagerView
    - CQueryBox
- CHTML_tc
- CHTML_textarea
- CHTML_tr
- CHTMLListElement
    - CHTML_dir
    - CHTML_menu
    - CHTML_ol
    - CHTML_ul
- CHTMLPlainText
- CHTMLTagNode
- CHTMLDualNode
    - CHTMLSpecialChar
- CHTMLText
- CPager
- CPagerBox
- CSelection
- CSmallPagerBox
- CButtonList (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. An HTML select button with a drop down list; used in CPagerBox. The constructor takes no arguments, and child nodes (options) are added using method CbuttonList::CreateSubNodes()
- CHTML_a Derived from CHTMLElement, defined in html.hpp - an HTML anchor element, as used in <*a href="..."*>. The constructor takes the URL string as the argument, and optionally, a CNCBINode to be appended as a child node. The label inserted before the closing tag (</a>) can thus be specified by providing a CHTMLText node to the constructor, or by using the AppendChild() after the anchor has been created.
- CHTML_basefont Derived from CHTMLElement, defined in html.hpp - an HTML basefont element used to define the font size and/or typeface for text embedded in this node by AppendChild(). The constructor expects one to two arguments specifying size, typeface, or both.

- CHTML_br Derived from CHTMLOpenElement, defined in html.hpp - the HTML component used to insert line breaks. The constructor takes no arguments.
- CHTML_checkbox Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a checkbox. The constructor takes up to four arguments specifying the name (string), value (string), state (bool), and description (string) for the node.
- CHTML_color Derived from CHTML_font, defined in html.hpp - an HTML font color element. Two constructors are available, and both expect string color as the first argument. If no other argument is provided, a NULL CNCBINode is assumed for the second argument, and text can be added to the node using AppendChild(). An alternative constructor accepts a simple string text argument.
- CHTML_dir Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert a dir list. The constructor takes zero to two arguments; if no arguments are provided, the compact attribute is by default false, and the type attribute is left to the browser. CHTML_dir("square", true) will create a compact dir element with square icons. Items can be added to the list using AppendChild(new CHTMLText ("<li>...").
- CHTML_dl Derived from CHTMLElement, defined in html.hpp - an HTML glossary list. The constructor takes a single bool argument; if no arguments are provided, the compact attribute is by default false. Terms are added to the list using AppendTerm ().
- CHTML_fieldset Derived from CHTMLElement, defined in html.hpp - an element that groups related form controls (such as checkboxes, radio buttons, etc.) together to define a form control group. The constructors take at most 1 argument, which may be either a string or a CHTML_legend node. If the argument is a string, then it is used to create a CHTML_legend node for the fieldset. The individual form controls to be included in the group are specified using the AppendChild() method.
- CHTML_file Derived from CHTML_input, defined in html.hpp - used only inside a CHTML_form - a form input type to create a file widget for selecting files to be sent to the server. The constructor takes a string name and an optional string value.
- CHTML_font Derived from CHTMLElement, defined in html.hpp - an HTML font element. The constructor takes up to four arguments. The first three arguments specify the font typeface and size, along with a Boolean value indicating whether the given font size is absolute or relative. The last argument is either a string or a CNCBINode containing text. Additional text should be added using the AppendChild() method.
- CHTML_form Derived from CHTMLElement, defined in html.hpp - an HTML form node with two constructors. The first takes the URL string (for submission of form data) and method (CHTML::eGet or CHTML::ePost), and the AppendChild() method is used to add nodes. The second constructor takes three arguments, specifying the URL, an HTML node to append to the form, and the enumereated get/post method.
- CHTML_hidden Derived from CHTML_input, defined in html.hpp - used only inside a CHTML_form - the HTML node for adding hidden key/value pairs to the data that will be submitted by an CHTML_form. The constructor takes a name string and a value, where the latter may be either a string or an int.
- CHTML_hr Derived from CHTMLOpenElement, defined in html.hpp - the HTML component used to insert a horizontal rule. The constructor takes up to three arguments, specifying the size, width and shading to be used in the display.
- CHTML_image Derived from CHTML_input, defined in html.hpp - used only inside a CHTML_form - the HTML component used to add an inline active image to an

HTML form. Clicking on the image submits the form data to the CHTML_form's URL. The constructor takes three arguments, specifying the name of the node, the URL string for the image file, and a Boolean value (optional) indicating whether or not the displayed image should have a border.

- CHTML_img Derived from CHTMLOpenElement, defined in html.hpp - an HTML img component for adding an inline image to a web page. The constructor takes a single URL string argument for the image's src. The alternative constructor also accepts two integer arguments specifying the width and height of the displayed image.
- CHTML_input Derived from CHTMLOpenElement, defined in html.hpp - the base class for all HTML input elements to be added to a CHTML_form. The constructor takes a (char*) input type and a (string) name. The constructor for each of the subclasses has a static member sm_InputType which is passed as the first argument to the CParent's (CHTML_input) constructor.
- CHTML_label Derived from CHTMLElement, defined in html.hpp - associates a label with a form control. The constructors take a string argument which specifies the text for the label, and optionally, a second string argument specifying the FOR attribute. The FOR attribute explicitly identifies the form control to associate with this label.
- CHTML_legend Derived from CHTMLElement, defined in html.hpp - defines a caption for a CHTML_fieldset element. The constructors take a single argument which may be either a string or a CHTMLNode.
- CHTML_menu Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert a menu list. The constructor takes zero to two arguments; if no arguments are provided, the compact attribute is by default false, and the type attribute is left to the browser. CHTML_menu("square", true) will create a compact menu element with square icons. Items can be added to the list using AppendChild (new CHTMLText("<li>...").
- CHTML_ol Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert an enumerated list. The constructor takes up to three arguments, specifying the starting number, the type of enumeration (Arabic, Roman Numeral etc.), and a Boolean argument specifying whether or not the display should be compact. Items can be added to the list using AppendChild(new CHTMLText ("<li>...").
- CHTML_option Derived from CHTMLElement, defined in html.hpp - an HTML option associated with a CHTML_select component. The constructor takes a value (string), a label (string or char*), and a Boolean indicating whether or not the option is by default selected. The last two arguments are optional, and by default the option is not selected.
- CHTML_radio Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a radio button. The constructor takes up to four arguments specifying the name (string), value (string), state (bool), and description (string) for the node.
- CHTML_reset Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a reset button. The constructor takes a single optional argument specifying the button's label.
- CHTML_select Derived from CHTMLElement, defined in html.hpp - an HTML select component. The constructor takes up to three arguments, specifying the name (string) and size (int) of the selection box, along with a Boolean specifying whether or not multiple selections are allowed (default is false). Select options should be added using the AppendOption() method.

- CHTML_submit Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a submit button. The constructor takes two string arguments specifying the button's name and label (optional). When selected, this causes the data selections in the including form to be sent to the form's URL.
- CHTML_table Derived from CHTMLElement, defined in html.hpp - an HTML table element. The constructor takes no arguments, but many member functions are provided to get/set attributes of the table. Because each of the "set attribute" methods returns this, the invocations can be strung together in a single statement.
  Use InsertAt(row, col, contents) to add contents to table cell row, col. To add contents to the next available cell, use AppendChild (new
  <listref rid="webpgs.html_CHTML_tc" RBID="webpgs.html_CHTML_tc">
  CHTML_tc </listref>
  (tag, contents)), where tag is type char* and contents is type char*, string or CNCBINode*.
- CHTML_tc Derived from CHTMLElement, defined in html.hpp - an HTML table cell element. All of the constructors expect the first argument to be a char* tagname. The second argument, if present, may be text (char* or string) or a pointer to a CNCBINode.
- CHTML_text Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a text box inside a form. The constructor takes up to four arguments: name (string), size (int), maxlength (int), and value (string). Only the first argument is required.
- CHTML_textarea Derived from CHTML_input, defined in html.hpp - can only be used inside a CHTML_form; the HTML component for a textarea inside a form. The constructor takes up to four arguments: name (string), cols (int), rows (int), and value (string). Only the last argument is optional.
- CHTML_tr Derived from CHTMLElement, defined in html.hpp - an HTML table row element. The constructors take a single argument, which may be either a string or a pointer to a CNCBINode.
- CHTML_ul Derived from CHTMLListElement, defined in html.hpp - the HTML component used to insert an unordered list. The constructor takes zero to two arguments; if no arguments are provided, the compact attribute is by default false, and the type attribute is left to the browser. CHTML_menu("square", true) will create a compact list element with square icons. Items can be added to the list using AppendChild(new CHTMLText("<li>...").
- CHTMLBasicPage Derived from CNCBINode, defined in page.hpp - The base class for CHTMLPage and its descendants. The HTML page classes serve as generalized containers for collections of other HTML elements, which together define a web page. Each page has a TTagMap, which maps names (strings) to the HTML subcomponents embedded in the page. Two constructors are defined. The first takes no arguments, and the other, takes a pointer to a CCgiApplication and a style (int) argument.
- CHTMLComment Derived from CHTMLNode, defined in html.hpp - used to insert an HTML comment. The constructor takes at most one argument, which may be a char*, a string, or a CNCBINode. The constructor then uses AppendPlainText() or AppendChild(), depending on the type of argument, to append the argument to the comment node.
- CHTMLElement Derived from CHTMLOpenElement, defined in html.hpp - the base class for all tagged elements which require a closing tag of the form </tagname>. CHTMLElement specializes the PrintEnd() method by generating the end tag </m_Name> on the output, where m_Name stores the tagname of the instance's subclass.

Subclasses include CHTML_a, CHTML_basefont, CHTML_dl, CHTML_font, CHTML_form, CHTML_option, CHTML_select, CHTML_table, CHTML_tc, CHTML_textarea, and CHTMLListElement.

- CHTMLListElement Derived from CHTMLElement, defined in html.hpp - the base class for CHTML_ul, CHTML_ol, CHTML_dir, and CHTML_menu lists. Arguments to the constructor include the tagname and type strings for the list, along with a Boolean indicating whether or not the list is compact.
- CHTMLNode Derived from CNCBINode, defined in html.hpp - the base class for CHTMLComment and CHTMLOpenElement. Attributes include style, id, title, accesskey, color, bgcolor, height, width, align, valign, size, name, and class. All of the constructors require a tagname argument, which may be either type char* or string. The optional second argument may be type char*, string, or CNCBINode.
- CHTMLOpenElement Derived from CHTMLNode, defined in html.hpp - the base class for all tag elements, including CHTMLElement, CHTML_br, CHTML_hr, CHTML_img, and CHTML_input. All of the constructors require a tagname argument, which may be either type char* or string. The optional second argument may be type char*, string, or CNCBINode.
- CHTMLPage Derived from CHTMLBasicPage; defined in page.hpp - the basic 3 section NCBI page. There are two constructors. The first takes a title (type string) and the name of a template file (type string). Both arguments are optional. The other constructor takes a pointer to a CCgiApplication, a style (type int), a title and a template_file name. All but the first argument are optional.
- CHTMLPlainText Derived from CNCBINode, defined in html.hpp - A simple text component, which can be used to insert text that will be displayed verbatim by a browser (may require encoding). The constructor takes two arguments: the text to be inserted (char* or string) and a Boolean (default false) indicating that the output **should** be encoded. See also CHTMLText.
- CHTMLTagNode Derived from CNCBINode; defined in html.hpp.
- CHTMLDualNode Derived from CNCBINode, defined in html.hpp - Allows the user to explicitly specify what exactly to print out in eHTML and in ePlainText modes. The constructor takes 2 arguments -- the first one is for eHTML mode output (string or a pointer to a CNCBINode), and the second one is a plain text for ePlainText mode output.
- CHTMLSpecialChar Derived from CHTMLDualNode, defined in html.hpp - A class for HTML special chars like &nbsp, &copy, etc. Elements of this class have two variants for output, for eHTML and ePlainText modes. For example: &nbsp have plain text variant - " ", and &copy - "(c)". html.hpp has several predefined simple classes, based on this class, for any special chars. It is CHTML_nbsp, CHTML_gt, CHTML_lt, CHTML_quot, CHTML_amp, CHTML_copy and CHTML_reg. Each have one optional arqument, which specify the number of symbols to output.
- CHTMLText Derived from CNCBINode, defined in html.hpp - A simple text component which can be used to install a default web page design (stored in a template file) on a CHTMLPage or to simply insert encoded text. The PrintBegin() is specialized to handle tagnodes occurring in the text. The constructor takes a single argument - the text itself - which may be of type char* or string. CHTMLPlainText should be used to insert text that does not embed any tagnodes and requires further encoding.
- CNCBINode Derived from CObject, defined in node.hpp - A base class for all other HTML node classes. Contains data members m_Name, m_Attributes, and m_Children.

The constructor takes at most one argument, name, which defines the internal data member m_Name.

- CPageList (Custom feature not for general use.) Derived from CHTML_table; defined in components.hpp. Used by the pager box components to page between results pages; contains forward and backward URLs, the current page number, and a map<int, string> that associates page numbers with URLs.
- CPager (Custom feature not for general use.) Derived from CNCBINode, defined in html.hpp
- CPagerBox (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. A more elaborate paging component than the CSmallPagerBox; contains pointers to a CPageList and (3) CButtonList components (left, right, and top). Additional properties include width, background color, and number of results.
- CPagerView (Custom feature not for general use.) Derived from CHTML_table; defined in pager.hpp.
- CQueryBox (Custom feature not for general use.) Derived from CHTML_table; defined in components.hpp.
- CSelection (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. A checkbox-like component whose choices are generated (using the CreateSubNodes() method) from the TCgiEntries of a CCgiRequest object.
- CSmallPagerBox (Custom feature not for general use.) Derived from CNCBINode; defined in components.hpp. A minimal paging component that displays the number of results from the query and the current page being viewed. Has background color and width attributes and contains a pointer to a CPageList. See also CPagerBox and CPager.

13: Data Serialization (ASN.1, XML)

Last Update: October 10, 2012.

The SERIAL API [Library xserial:include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The Serial library provides a means for loading, accessing, manipulating, and serialization of data in a formatted way. It supports serialization in ASN.1 (text or BER encoding), XML, and JSON formats. See also the DATATOOL documentation discussion of generating C++ code for serializable objects from the corresponding ASN.1 definition.

The structure of data is described by some sort of formal language. In our case it can be ASN.1, DTD or XML Schema. Based on such specification, DATATOOL application, which is part of NCBI C++ toolkit, generates a collection of data storage classes that can be used to store and serialize data. The design purpose was to make these classes as lightweight as possible, moving all details of serialization into specialized classes - "object streams". Structure of the data is described with the help of "type information". Data objects contain data and type information only. Any such data storage object can be viewed as a node tree that provides random access to its data. The Serial library provides a means of traversing this data tree without knowing its structure in advance – using only type information; C++ code generated by DATATOOL makes it possible to access any child node directly.

"Object streams" are intermediaries between data storage objects and input or output stream. They perform encoding or decoding of data according to format specifications. Guided by the type information embedded into data object, on reading they allocate memory when needed, fill in data, and validate that all mandatory data is present; on writing they guarantee that all relevant data is written and that the resulting document is well-formed. All it takes to read or write a top-level data object is one function call – all the details are handled by an object stream.

Closely related to serialization is the task of converting data from one format into another. One approach could be reading data object completely into memory and then writing it in another format. The only problem is that the size of data can be huge. To simplify this task and to avoid storing data in memory, the serial library provides the "object stream copier" class. It reads data by small chunks and writes it immediately after reading. In addition to small memory footprint, it also works much faster.

Input data can be very large in size; also, reading it completely into memory could not be the goal of processing. Having a large file of data, one might want to investigate information containers only of a particular type. Serial library provides a variety of means for doing this. The list includes read and write hooks, several types of stream iterators, and filter templates. It is worth to note that, when using read hooks to read child nodes, one might end up with an invalid top-level data object; or, when using write hooks, one might begin with an invalid object and fill in missing data on the fly – in hooks.

In essence, "hook" is a callback function that client application provides to serial library. Client application installs the hook, then reads (or writes) data object, and somewhere from the depths of serialization processing, the library calls this hook function at appropriate times, for example, when a data chunk of specified type is about to be read. It is also possible to install context-specific hooks. Such hooks are triggered when serializing a particular object type in a particular structural context; for example, for all objects of class A which are contained in object B.

Chapter Outline

The following is an outline of the topics presented in this chapter:

CObject[IO]Streams

The following topics are discussed in this section:

- Format Specific Streams: The CObject[IO]Stream classes
- The CObjectIStream (*) classes
- The CObjectOStream (*) classes
- The CObjectStreamCopier (*) classes

Format Specific Streams: The CObject[IO]Stream classes

The reading and writing of serialized data objects entails satisfying two independent sets of constraints and specifications: (1) format-specific parsing and encoding schemes, and (2) object-specific internal structures and rules of composition. The NCBI C++ Toolkit implements serial IO processes by combining a set of object stream classes with an independently defined set of data object classes. These classes are implemented in the serial and objects directories respectively.

The base classes for the object stream classes are CObjectIStream and CObjectOStream. Each of these base classes has derived subclasses which specialize in different formats, including XML, binary ASN.1, and text ASN.1. A simple example program, xml2asn.cpp (see Code Sample 1), described in Processing serial data, uses these object stream classes in conjunction with a CBiostruct object to translate a file from XML encoding to ASN.1 formats. In this chapter, we consider in more detail the class definitions for object streams, and how the type information associated with the data is used to implement serial input and output.

Each object stream specializes in a serial data format and a direction (in/out). It is not until the input and output operators are applied to these streams, in conjunction with a specified serializable object, that the object-specific type information comes into play. For example, if instr is a CObjectIStream, the statement: instr >> myObject invokes a Read() method associated with the input stream, whose sole argument is a CObjectInfo for myObject.

Similarly, the output operators, when applied to a CObjectOstream in conjunction with a serializable object, will invoke a Write() method on the output stream which accesses the object's type information. The object's type information defines what tag names and value types should be encountered on the stream, while the CObject[IO]Stream subclasses specialize the data serialization format.

The input and output operators (<< and >>) are declared in serial/serial.hpp header.

The CObjectIStream (*) classes

CObjectIStream is a virtual base class for the CObjectIStreamXml, CObjectIStreamAsn, and CObjectIStreamAsnBinary classes. As such, it has no public constructors, and its user interface includes the following methods:

- Open()
- Close()
- GetDataFormat()
- ReadFileHeader()
- Read()
- ReadObject()
- ReadSeparateObject()
- Skip()
- SkipObject()

There are several Open() methods; most of these are static class methods that return a pointer to a newly created CObjectIStream. Typically, these methods are used with an auto_ptr, as in:

```
auto_ptr<CObjectIStream> xml_in(CObjectIStream::Open(filename, eSerial_Xml));
```

Here, an XML format is specified by the enumerated value eSerial_Xml, defined in ESerialDataFormat. Because these methods are static, they can be used to create a new instance of a CObjectIStream subclass, and open it with one statement. In this example, a CObjectIStreamXml is created and opened on the file filename.

An additional non-static Open() method is provided, which can only be invoked as a member function of a previously instantiated object stream (whose format type is of course, implicit to its class). This method takes a CNcbiIstream and a flag indicating whether or not ownership of the CNcbiIstream should be transferred (so that it can be deleted automatically when the object stream is closed):

```
void Open(CNcbiIstream& inStream, EOwnership deleteInStream = eNoOwnership);
```

The next three methods have the following definitions. Close() closes the stream. GetDataFormat() returns the enumerated ESerialDataFormat for the stream. ReadFileHeader () reads the first line from the file, and returns it in a string. This might be used for example, in the following context:

```
auto_ptr<CObjectIStream> in(CObjectIStream::Open(fname, eSerial_AsnText));
string type = in.ReadFileHeader();
if (type.compare("Seq-entry") == 0) {
 CSeq_entry seqent;
 in->Read(ObjectInfo(seqent), eNoFileHeader);
 // ...
}
else if (type.compare("Bioseq-set") == 0) {
 CBioseq_set seqset;
 in->Read(ObjectInfo(seqset), eNoFileHeader);
 // ...
}
```

The ReadFileHeader() method for the base CObjectIStream class returns an empty string. Only those stream classes which specialize in ASN.1 text or XML formats have actual implementations for this method.

Several Read*() methods are provided for usage in different contexts. CObjectIStream::Read() should be used for reading a top-level "root" object from a data file. For convenience, the input operator >>, as described above, indirectly invokes this method on the input stream, using a CObjectTypeInfo object derived from myObject. By default, the Read() method first calls ReadFileHeader(), and then calls ReadObject(). Accordingly, calls to Read() which follow the usage of ReadFileHeader()**must** include the optional eNoFileHeader argument.

Most data objects also contain embedded objects, and the default behavior of Read() is to load the top-level object, along with all of its contained subobjects into memory. In some cases this may require significant memory allocation, and it may be only the top-level object which is needed by the application. The next two methods, ReadObject() and ReadSeparateObject(), can be used to load subobjects as either persistent data members of the root object or as temporary local objects. In contrast to Read(), these methods assume that there is no file header on the stream.

As a result of executing ReadObject(member), the newly created subobject will be instantiated as a member of its parent object. In contrast, ReadSeparateObject(local), instantiates the subobject in the local temporary variable only, and the corresponding data member in the parent object is set to an appropriate null representation for that data type. In this case, an attempt to reference that subobject after exiting the scope where it was created generates an error.

The Skip() and SkipObject() methods allow entire top-level objects and subobjects to be "skipped". In this case the input is still read from the stream and validated, but no object representation for that data is generated. Instead, the data is stored in a delay buffer associated with the object input stream, where it can be accessed as needed. Skip() should only be applied to top-level objects. As with the Read() method, the optional ENoFileHeader argument can be included if the file header has already been extracted from the data stream. SkipObject(member) may be applied to subobjects of the root object.

All of the Read and Skip methods are like wrapper functions, which define what activities take place immediately before and after the data is actually read. How and when the data is then loaded into memory is determined by the object itself. Each of the above methods ultimately calls objTypeInfo->ReadData() or objTypeInfo->SkipData(), where objTypeInfo is the static type information object associated with the data object. This scheme allows the user to install type-specific read, write, and copy hooks, which are described below. For example, the default behavior of loading all subobjects of the top-level object can be modified by installing appropriate read hooks which use the ReadSeparateObject() and SkipObject() methods where needed.

The CObjectOStream (*) classes

The output object stream classes mirror the CObjectIStream classes. The CObjectOStream base class is used to derive the CObjectOStreamXml, CObjectOStreamAsn, and CObjectOStreamAsnBinary classes. There are no public constructors, and the user interface includes the following methods:

- Open()
- Close()
- GetDataFormat()
- WriteFileHeader()
- Write()
- WriteObject()

- WriteSeparateObject()
- Flush()
- FlushBuffer()

Again, there are several Open() methods, which are static class methods that return a pointer to a newly created CObjectOstream:

```
static CObjectOStream* Open(ESerialDataFormat format,
 CNcbiOstream &outStream,
 EOwnership deleteOutStream=eNoOwnership,
 TSerial_Format_Flags formatFlags=0)

static CObjectOStream* Open(ESerialDataFormat format,
 const string &fileName,
 TSerialOpenFlags openFlags=0,
 TSerial_Format_Flags formatFlags=0)

static CObjectOStream* Open(const string &fileName,
 ESerialDataFormat format,
 TSerial_Format_Flags formatFlags=0)
```

The Write*() methods correspond to the Read*() methods defined for the input streams. Write () first calls WriteFileHeader(), and then calls WriteObject(). WriteSeparateObject() can be used to write a temporary object (and all of its children) to the output stream. It is also possible to install type-specific write hooks. Like the Read() methods, these Write() methods serve as wrapper functions that define what occurs immediately before and after the data is actually written.

The CObjectStreamCopier (*) classes

The CObjectStreamCopier class is neither an input nor an output stream class, but a helper class, which allows one to "pass data through" without storing the intermediate objects in memory. Its sole constructor is:

```
CObjectStreamCopier(CObjectIStream& in, CObjectOStream& out);
```

and its most important method is the Copy(CObjectTypeInfo&) method, which, given an object's description, reads that object from the input stream and writes it to the output stream. The serial formats of both the input and output object streams are implicit, and thus the translation between two different formats is performed automatically.

In keeping with the Read and Write methods of the CObjectIStream and CObjectOStream classes, the Copy method takes an optional ENoFileHeader argument, to indicate that the file header is not present in the input and should not be generated on the output. The CopyObject () method corresponds to the ReadObject() and WriteObject() methods.

As an example, consider how the Run() method in xml2asn.cpp might be implemented differently using the CObjectStreamCopier class:

```
int CTestAsn::Run() {
auto_ptr<CObjectIStream>
xml_in(CObjectIStream::Open("1001.xml", eSerial_Xml));
auto_ptr<CObjectOStream>
txt_out(CObjectOStream::Open("1001.asntxt", eSerial_AsnText));
CObjectStreamCopier txt_copier(*xml_in, *txt_out);
txt_copier.Copy(CBiostruc::GetTypeInfo());
auto_ptr<CObjectOStream>
 bin_out(CObjectOStream::Open("1001.asnbin", eSerial_AsnBinary));
CObjectStreamCopier bin_copier(*xml_in, *bin_out);
bin_copier.Copy(CBiostruc::GetTypeInfo());
return 0;
}
```

It is also possible to install type-specific Copy hooks. Like the Read and Write methods, the Copy methods serve as wrapper functions that define what occurs immediately before and after the data is actually copied.

Type-specific I/O routines – the hook classes

Much of the functionality needed to read and write serializable objects may be type-specific yet application-driven. Because the specializations may vary with the application, it does not make sense to implement fixed methods, yet we would like to achieve a similar kind of object-specific behavior.

To address these needs, the C++ Toolkit provides hook mechanisms, whereby the needed functionality can be installed with the object's static class type information object. Local hooks apply to a selected stream whereas global hooks apply to all streams. Note: global skip hooks are not supported.

For any given object type, stream, and processing mode (e.g. reading), at most one hook is "active". The active hook for the current processing mode will be called when objects of the given type are encountered in the stream. For example, suppose that local and global hooks have been set for a given object type. Then if a read occurs on the stream for which the local hook was set, the local hook will be called, otherwise the global hook will be called. Designating multiple read/write hooks (both local and global) for a selected object does not generate an error. Older or less specific hooks are simply overridden by the more specific or most recently installed hook.

Understanding and creating hooks properly relies on three distinct concepts:

- **Structural Context** – the criteria for deciding which objects in the stream will be hooked.
- **Processing Mode** – what is being done when the hook should be called. Hooks will only be called in the corresponding processing mode. For example, if content is being skipped, only skip hooks will be called. If the mode changes to reading, then only read hooks will be called.
- **Operation** – easily confused with processing mode, the operation is what is done inside the hook, not what is being done when the hook is called.

Note: The difference between processing mode and operation can be very confusing. It is natural to think, for example, "I want to read Bioseq id's" without considering how the stream is being processed. The next natural step is to conclude "I want a read hook" - but that could be incorrect. Instead, one should think "I want to *read* a Bioseq id *inside* a hook". Only then should the processing mode be chosen, and it may not match the operation performed inside the hook. The processing mode should be chosen based on what should be done with the *rest* of the stream and whether or not it's necessary to retain the data *outside* the hook. For example, if you want to read Bioseq id's and don't care about anything else, then you should probably choose the 'skip' processing mode (meaning you would use a skip hook), and *within* the skip hook you would *read* the Bioseq id. Or, if you wanted to read entire Bioseq's for later analysis while automatically building a list of Bioseq id's, you would have to use the 'read' processing mode (and therefore a read hook) to save the data for later analysis. Inside the read hook you would use a read operation (to save the data) and at the same time you would have access to the id for building the list of id's.

There are three main **structural contexts** in which an object might be encountered in a stream:

Context	Description
Object	When the stream object matches a specified type – for example, the Bioseq type.
Class Member	When the stream object matches a specified member of a specified SEQUENCE type – for example, the id member of the Bioseq type.
Choice Variant	When the stream object matches a specified variant of a specified CHOICE type – for example, the std variant of the Date type.

Complex structural contexts can be created by nesting the main structural contexts. For example, a stack path hook can apply to a specific class member, but only when it is nested inside another specified class member.

There are four **processing modes** that can be applied to input/output streams:

Mode	Description
Read	When objects are parsed from an input stream and a deserialized instance is retained.
Skip	When objects are parsed from an input stream but a deserialized instance is not retained
Copy	When objects are parsed from an input stream and written directly to an output stream.
Write	When objects are written to an output stream.

The **operation** is not restricted to a limited set of choices. It can be any application-specific task, as long as that task is compatible with the processing mode. For example, a skip operation can be performed inside a read hook, provided that the skipped content is optional for the object being read. Similarly, a read operation can be performed inside a skip hook. The operation performed inside a hook must preserve the integrity of the hooked object, and must advance the stream all the way through the hooked object and no farther.

Hooks can be installed for all combinations of structural context and processing mode. Each combination has a base class that defines a pure virtual method that must be defined in a derived class to implement the hook – e.g. the CReadObjectHook class defines a pure virtual ReadObject() method. The definition of the overriding method in the derived class is often referred to as "the hook".

	Object	Class Member	Choice Variant
Read	CReadObjectHook	CReadClassMemberHook	CReadChoiceVariantHook
Write	CWriteObjectHook	CWriteClassMemberHook	CWriteChoiceVariantHook
Copy	CCopyObjectHook	CCopyClassMemberHook	CCopyChoiceVariantHook
Skip	CSkipObjectHook	CSkipClassMemberHook	CSkipChoiceVariantHook

In addition, there is a hook guard class, which simplifies creating any of the above hooks. There are also stack path hook methods corresponding to each structural context / processing mode combination above, making it easy to create hooks for virtually any conceivable situation.

Hook Sample

Here is a complete program that illustrates how to create a read hook for class members (other sample programs are available at http://www.ncbi.nlm.nih.gov/viewvc/v1/trunk/c%2B%2B/src/sample/app/serial/):

```
#include <ncbi_pch.hpp>
#include <objects/general/Date_std.hpp>
#include <serial/objistr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

// This class implements a read hook for class members.
//
// A read hook is created by passing a new instance of this class to a
// "set hook" method. Hooks may be created as global or local. Global hooks
// apply to all streams, whereas local hooks are associated with a specific
// stream. Thus, the "set hook" methods for creating class member read hooks
// are:
// SetGlobalReadHook()
// SetLocalReadHook()
//
// This class must override the virtual method ReadClassMember(). See the
// comment for the ReadClassMember() method below for more details.
//
// In principle, multiple instances of this hook class could be used to
provide
// the same hook processing for more than one entity. However, it is probably
// best to create a separate class for each "thing" you want to hook and
// process.
//
// You should adopt a meaningful naming convention for your hook classes.
// In this example, the convention is C<mode><context>Hook_<object>__<member>
// where: <mode>=(Read|Write|Copy|Skip)
// <context>=(Obj|CM|CV) -- object, class member, or choice variant
// and hyphens in ASN.1 object types are replaced with underscores.
//
// Note: Since this is a read hook, ReadClassMember() will only be called
when
// reading from the stream. If the stream is being skipped, ReadClassMember()
// will not be called. If you want to use a hook to read a specific type of
// class member while skipping everything else, use a skip hook and call
// DefaultRead() from within the SkipClassMember() method.
//
// Note: This example is a read hook, which means that the input stream is
// being read when the hook is called. Hooks for other processing modes
// (Write, Skip, and Copy) are similarly created by inheriting from the
// respecitive base classes. It is also a ClassMember hook. Hooks for
// other structural contexts (Object and ChoiceVariant) a similarly derived
// from the appropriate base.
class CDemoHook : public CReadClassMemberHook
{
public:
 // Implement the hook method.
 //
 // Once the read hook has been set, ReadClassMember() will be called
 // whenever the specified class member is encountered while
 // reading a hooked input stream. Without the hook, the encountered
 // class member would have been automatically read. With the hook, it is
 // now the responsibility of the ReadClassMember() method to remove the
 // class member from the input stream and process it as desired. It can
 // either read it or skip it to remove it from the stream. This is
 // easily done by calling DefaultRead() or DefaultSkip() from within
 // ReadClassMember(). Subsequent processing is up to the application.
 virtual void ReadClassMember(CObjectIStream& in,
 const CObjectInfoMI& passed_info)
 {
 // Perform any pre-read processing here.
 //NcbiCout << "In ReadClassMember() hook, before reading." << NcbiEndl;

 // You must call DefaultRead() (or perform an equivalent operation)
 // if you want the object to be read into memory. You could also
 // call DefaultSkip() if you wanted to skip the hooked object while
 // reading everything else.
 DefaultRead(in, passed_info);

 // Perform any post-read processing here. Once the object has been
 // read, its data can be used for processing. For example, here we dump
 // the read object into the standard output.
 NcbiCout << MSerial_AsnText << passed_info.GetClassObject();
 }
};

int main(int argc, char** argv)
{
 // Create some ASN.1 data that can be parsed by this code sample.
 char asn[] = "Date-std ::= { year 1998 }";

 // Setup an input stream, based on the sample ASN.1.
 CNcbiIstrstream iss(asn);
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, iss));

 ////////////////////////////////////////////////////
 // Create a hook for the 'year' class member of Date-std objects.
 // The year class member was aribtrarily chosen to illustrate the
 // use of hooks - many other entities would work equally well.
 // Get data structures that model the type information for Date-std
 // objects and their 'year' class members.
 // The type information will be used to recognize and forward 'year'
 // class members of Date-std objects found in the stream to the hook.
 CObjectTypeInfo typeInfo = CType<CDate_std>();
 CObjectTypeInfoMI memberInfo = typeInfo.FindMember("year");

 // Set a local hook for Date-std 'year' class members. This involves
 // creating an instance of the hook class and passing that hook to the
 // "set hook" method, which registers the hook to be called when a hooked
 // type is encountered in the stream.
 memberInfo.SetLocalReadHook(*in, new CDemoHook);


 // The above three statements could be shortened to:
 //CObjectTypeInfo(CType<CDate_std>()).FindMember("year")
 // .SetLocalReadHook(*in, new CDemoHook);


 // Read from the input stream, storing data in the object. At this point,
 // the hook is in place so simply reading from the input stream will
 // cause the hook to be triggered whenever the 'year' class member is
 // encountered.
 CDate_std my_date;
 *in >> my_date;

 return 0;
}
```

Read mode hooks

All of the different structural contexts in which an object might be encountered on an input stream can be reduced to three cases:

- as a stand-alone object
- as a data member of a containing object
- as a variant of a choice object

Hooks can be installed for each of the above contexts, depending on the desired level of specificity. Corresponding to these contexts, three abstract base classes provide the foundations for deriving new Read hooks:

- CReadObjectHook
- CReadClassMemberHook
- CReadChoiceVariantHook

Each of these base hook classes exists only to define a pure virtual Read method, which can then be implemented (in a derived subclass) to install the desired type of read hook. If the goal is to apply the new Read method in all contexts, then the new hook should be derived from the CReadObjectHook class, and registered with the object's static type information object. For example, to install a new CReadObjectHook for a CBioseq, one might use:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).
 SetLocalReadHook(*in, myReadBioseqHook);
```

Another way of installing hooks of any type (read/write/copy, object/member/variant) is provided by CObjectHookGuard class described below.

Alternatively, if the desired behavior is to trigger the specialized Read method only when the object occurs as a data member of a particular containing class, then the new hook should be derived from the CReadClassMemberHook, and registered with that member's type information object:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).
 FindMember("Seq-inst").SetLocalReadHook(*in, myHook);
```

Similarly, one can install a read hook that will only be triggered when the object occurs as a choice variant:

```
CObjectTypeInfo(CSeq_entry::GetTypeInfo()).
 FindVariant("Bioseq").SetLocalReadHook(*in, myReadBioseqHook);
```

The new hook classes for these examples should be derived from CReadObjectHook, CReadClassMemberHook, and CReadChoiceVariantHook, respectively. In the first case, all occurrences of CBioseq on any input stream will trigger the new Read method. In contrast, the third case installs this new Read method to be triggered only when the CBioseq occurs as a choice variant in a CSeq_entry object.

All of the virtual Read methods take two arguments: a CObjectIStream and a reference to a CObjectInfo. For example, the CReadObjectHook class declares the ReadObject() method as:

```
virtual void ReadObject(CObjectIStream& in,
 const CObjectInfo& object) = 0;
```

The ReadClassMember and ReadChoiceVariant hooks differ from the ReadObject hook class, in that the second argument to the virtual Read method is an iterator, pointing to the object type information for a sequence member or choice variant respectively.

In summary, to install a read hook for an object type:

derive a new class from the appropriate hook class:

- if the hook should be called regardless of the structural context in which the target object occurs, use the CReadObjectHook class.
- if the target object occurs as a sequence member, use the CReadClassMemberHook class.
- if the target object occurs as a choice variant, use the CReadChoiceVariant Hook class.

implement the virtual Read method for the new class.

install the hook, using the SetLocalReadHook() method defined in

- CObjectTypeInfo for a CReadObjectHook
- CMemberInfo for a CReadClassMemberHook
- CVariantInfo for a CReadChoiceVariantHook

or use CObjectHookGuard class to install any of these hooks.

In many cases you will need to read the hooked object and do some special processing, or to skip the entire object. To simplify object reading or skipping all base hook classes have

DefaultRead() and DefaultSkip() methods taking the same arguments as the user provided ReadXXXX() methods. Thus, to read a bioseq object from a hook:

```
void CMyReadObjectHook::ReadObject(CObjectIStream& in,
 const CObjectInfo& object)
{
 DefaultRead(in, object);
 // Do some user-defined processing of the bioseq
}
```

Note that from a choice variant hook you can not skip stream data -- this could leave the choice object in an uninitialized state. For this reason the CReadChoiceVariantHook class has no DefaultSkip() method.

Read Object Hook Sample

A read object hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/general/Date_std.hpp>
#include <serial/objistr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CReadObjectHook
{
public:
 virtual void ReadObject(CObjectIStream& strm,
 const CObjectInfo& passed_info)
 {
 DefaultRead(strm, passed_info);
 }
};

int main(int argc, char** argv)
{
 char asn[] = "Date-std ::= { year 1998 }";
 CNcbiIstrstream iss(asn);
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, iss));

 CObjectTypeInfo(CType<CDate_std>()).SetLocalReadHook(*in, new CDemoHook());

 CDate_std my_date;
 *in >> my_date;

 return 0;
}
```

See the class documentation for more information.

Read Class Member Hook Sample

A read class member hook can be created very much like other hooks. For an example, see the hook sample.

See the class documentation for more information.

Read Choice Variant Hook Sample

A read choice variant hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/general/Date.hpp>
#include <serial/objistr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CReadChoiceVariantHook
{
public:
 virtual void ReadChoiceVariant(CObjectIStream& strm,
 const CObjectInfoCV& passed_info)
 {
 DefaultRead(strm, passed_info);
 }
};

int main(int argc, char** argv)
{
 char asn[] = "Date ::= str \"late-spring\"";
 CNcbiIstrstream iss(asn);
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, iss));

 CObjectTypeInfo(CType<CDate>()).FindVariant("str")
 .SetLocalReadHook(*in, new CDemoHook);

 CDate my_date;
 *in >> my_date;

 return 0;
}
```

See the class documentation for more information.

Write mode hooks

The Write hook classes parallel the Read hook classes, and again, we have three base classes:

- CWriteObjectHook
- CWriteClassMemberHook
- CWriteChoiceVariantHook

These classes define the pure virtual methods:

```
CWriteObjectHook::WriteObject(CObjectOStream&,
 const CConstObjectInfo& object) = 0;

CWriteClassMemberHook::WriteClassMember(CObjectOStream&,
 const CConstObjectInfoMI& member) = 0;

CWriteChoiceVariantHook::WriteChoiceVariant(CObjectOStream&,
 const CConstObjectInfoCV& variant) = 0;
```

Like the read hooks, your derived write hooks can be installed by invoking the SetLocalWriteObjectHook() methods for the appropriate type information objects. Corresponding to the examples for read hooks then, we would have:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).
 SetLocalWriteHook(*in, myWriteBioseqHook);

CObjectTypeInfo(CBioseq::GetTypeInfo()).
 FindMember("Seq-inst").SetLocalWriteHook(*in, myWriteSeqinstHook);

CObjectTypeInfo(CSeq_entry::GetTypeInfo()).
 FindVariant("Bioseq").SetLocalWriteHook(*in, myWriteBioseqHook);
```

CObjectHookGuard class provides is a simple way to install write hooks.

Write Object Hook Sample

A write object hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objectio.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CWriteObjectHook
{
public:
 virtual void WriteObject(CObjectOStream& out,
 const CConstObjectInfo& passed_info)
 {
 DefaultWrite(out, passed_info);
 }
};

int main(int argc, char** argv)
```

```
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 auto_ptr<CObjectOStream> out(CObjectOStream::Open(eSerial_AsnText, "of"));

 CObjectTypeInfo(CType<CCit_art>()).SetLocalWriteHook(*out, new CDemoHook);

 CCit_art article;
 *in >> article;
 *out << article;

 return 0;
}
```

See the class documentation for more information.

Write Class Member Hook Sample

A write class member hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Auth_list.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objectio.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook: public CWriteClassMemberHook
{
public:
 virtual void WriteClassMember(CObjectOStream& out,
 const CConstObjectInfoMI& passed_info)
 {
 DefaultWrite(out, passed_info);
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 auto_ptr<CObjectOStream> out(CObjectOStream::Open(eSerial_AsnText, "of"));

 CObjectTypeInfo(CType<CAuth_list>())
 .FindMember("names")
 .SetLocalWriteHook(*out, new CDemoHook);

 CCit_art article;
 *in >> article;
 *out << article;

 return 0;
}
```

See the class documentation for more information.

Write Choice Variant Hook Sample

A write choice variant hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Auth_list.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objectio.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CWriteChoiceVariantHook
{
public:
 virtual void WriteChoiceVariant(CObjectOStream& out,
 const CConstObjectInfoCV& passed_info)
 {
 DefaultWrite(out, passed_info);
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 auto_ptr<CObjectOStream> out(CObjectOStream::Open(eSerial_AsnText, "of"));

 (*CObjectTypeInfo(CType<CAuth_list>()).FindMember("names"))
 .GetPointedType()
 .FindVariant("std")
 .SetLocalWriteHook(*out, new CDemoHook);

 CCit_art article;
 *in >> article;
 *out << article;

 return 0;
}
```

See the class documentation for more information.

Copy mode hooks

As with the Read and Write hook classes, there are three base classes which define the following Copy methods:

```
CCopyObjectHook::CopyObject(CObjectStreamCopier& copier,
 const CObjectTypeInfo& object) = 0;

CCopyClassMemberHook::CopyClassMember(CObjectStreamCopier& copier,
 const CObjectTypeInfoMI& member) = 0;

CCopyChoiceVariantHook::CopyChoiceVariant(CObjectStreamCopier& copier,
 const CObjectTypeInfoCV& variant) = 0;
```

Newly derived copy hooks can be installed by invoking the SetLocalCopyObjectHook() method for the appropriate type information object. The other way of installing hooks is described below in the CObjectHookGuard section.

To do default copying of an object in the overloaded hook method each of the base copy hook classes has a DefaultCopy() method.

Copy Object Hook Sample

A copy object hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objcopy.hpp>
#include <serial/objectio.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CCopyObjectHook
{
public:
 virtual void CopyObject(CObjectStreamCopier& copier,
 const CObjectTypeInfo& passed_info)
 {
 DefaultCopy(copier, passed_info);
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 auto_ptr<CObjectOStream> out(CObjectOStream::Open(eSerial_AsnText, "of"));
 CObjectStreamCopier copier(*in, *out);
 CObjectTypeInfo(CType<CCit_art>())
 .SetLocalCopyHook(copier, new CDemoHook());

 copier.Copy(CType<CCit_art>());

 return 0;
}
```

See the class documentation for more information.

Copy Class Member Hook Sample

A copy class member hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/seq/Bioseq.hpp>
#include <objects/seqset/Seq_entry.hpp>
#include <serial/objcopy.hpp>
#include <serial/objectio.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CCopyClassMemberHook
{
public:
 virtual void CopyClassMember(CObjectStreamCopier& copier,
 const CObjectTypeInfoMI& passed_info)
 {
 DefaultCopy(copier, passed_info);
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 auto_ptr<CObjectOStream> out(CObjectOStream::Open(eSerial_AsnText, "of"));
 CObjectStreamCopier copier(*in, *out);

 CObjectTypeInfo(CType<CBioseq>())
 .FindMember("annot")
 .SetLocalCopyHook(copier, new CDemoHook());

 copier.Copy(CType<CBioseq>());

 return 0;
}
```

See the class documentation for more information.

Copy Choice Variant Hook Sample

A copy choice variant hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Auth_list.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objcopy.hpp>
#include <serial/objectio.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/serial.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CCopyChoiceVariantHook
{
public:
 virtual void CopyChoiceVariant(CObjectStreamCopier& copier,
 const CObjectTypeInfoCV& passed_info)
 {
 DefaultCopy(copier, passed_info);
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 auto_ptr<CObjectOStream> out(CObjectOStream::Open(eSerial_AsnText, "of"));
 CObjectStreamCopier copier(*in, *out);

 (*CObjectTypeInfo(CType<CAuth_list>()).FindMember("names"))
 .GetPointedType()
 .FindVariant("std")
 .SetLocalCopyHook(copier, new CDemoHook);

 copier.Copy(CType<CCit_art>());

 return 0;
}
```

See the class documentation for more information.

Skip mode hooks

As with the Read and Write hook classes, there are three base classes which define the following Skip methods:

```
CSkipObjectHook::SkipObject(CObjectIStream& in,
 const CObjectTypeInfo& object) = 0;

CSkipClassMemberHook::SkipClassMember(CObjectIStream& in,
 const CObjectTypeInfoMI& member) = 0;

CSkipChoiceVariantHook::SkipChoiceVariant(CObjectIStream& in,
 const CObjectTypeInfoCV& variant) = 0;
```

Newly derived skip hooks can be installed by invoking the SetLocalSkipObjectHook() method for the appropriate type information object. The other way of installing hooks is described below in the CObjectHookGuard section.

The CSkipObjectHook class has a DefaultSkip() method, like the base classes for the other processing modes, but for historical reasons DefaultSkip() methods were not defined for the CSkipClassMemberHook and CSkipChoiceVaraintHook classes. Nevertheless, achieving the same result is easily accomplished – for example:

```
class CMySkipClassMemberHook : public CSkipClassMemberHook
{
public:
 virtual void SkipClassMember(CObjectIStream& in,
 const CObjectTypeInfoMI& member)
 {
 in.SkipObject(*member);
 }
};
```

Skip Object Hook Sample

A skip object hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objistr.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CSkipObjectHook
{
public:
 virtual void SkipObject(CObjectIStream& in,
 const CObjectTypeInfo& passed_info)
 {
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));
 CObjectTypeInfo(CType<CCit_art>()).SetLocalSkipHook(*in, new CDemoHook);

 in->Skip(CType<CCit_art>());

 return 0;
}
```

See the class documentation for more information.

Skip Class Member Hook Sample

A skip class member hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Auth_list.hpp>
#include <objects/biblio/Cit_art.hpp>
#include <serial/objistr.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CSkipClassMemberHook
{
public:
 virtual void SkipClassMember(CObjectIStream& in,
 const CObjectTypeInfoMI& passed_info)
 {
 in.SkipObject(*passed_info);
 }
};

int main(int argc, char** argv)
{
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, "if"));

 CObjectTypeInfo(CType<CAuth_list>())
 .FindMember("names")
 .SetLocalSkipHook(*in, new CDemoHook);

 in->Skip(CType<CCit_art>());

 return 0;
}
```

See the class documentation for more information.

Skip Choice Variant Hook Sample

A skip choice variant hook can be created very much like other hooks. For example, the executable lines in the hook sample, can be replaced with:

```
#include <ncbi_pch.hpp>
#include <objects/biblio/Imprint.hpp>
#include <objects/general/Date.hpp>
#include <serial/objistr.hpp>

USING_NCBI_SCOPE;
USING_SCOPE(ncbi::objects);

class CDemoHook : public CSkipChoiceVariantHook
{
public:
 virtual void SkipChoiceVariant(CObjectIStream& in,
 const CObjectTypeInfoCV& passed_info)
 {
 in.SkipObject(*passed_info);
 }
};

int main(int argc, char** argv)
{
 char asn[] = "Imprint ::= { date std { year 2010 } }";
 CNcbiIstrstream iss(asn);
 auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, iss));

 CObjectTypeInfo(CType<CDate>()).FindVariant("std")
 .SetLocalSkipHook(*in, new CDemoHook());

 in->Skip(CType<CImprint>());

 return 0;
}
```

See the class documentation for more information.

The CObjectHookGuard class

To simplify hooks usage CObjectHookGuard class may be used. It's a template class: the template parameter is the class to be hooked (in case of member or choice variant hooks it's the parent class of the member).

The CObjectHookGuard class has several constructors for installing different hook types. The last argument to all constructors is a stream pointer. By default the pointer is NULL and the hook is intalled as a global one. To make the hook stream-local pass the stream to the guard constructor.

- Object read/write hooks:
  CObjectHookGuard(CReadObjectHook& hook,
  CObjectIStream* in = 0);
  CObjectHookGuard(CWriteObjectHook& hook,
  CObjectOStream* out = 0);
- Class member read/write hooks:
  CObjectHookGuard(string id,
  CReadClassMemberHook& hook,

CObjectIStream* in = 0);
CObjectHookGuard(string id,
CWriteClassMemberHook& hook,
CObjectOStream* out = 0);

The string "id" argument is the name of the member in ASN.1 specification for generated classes.

- Choice variant read/write hooks:
  CObjectHookGuard(string id,
  CReadChoiceVariantHook& hook,
  CObjectIStream* in = 0);
  CObjectHookGuard(string id,
  CWriteChoiceVariantHook& hook,
  CObjectOStream* out = 0);

The string "id" argument is the name of the variant in ASN.1 specification for generated classes.

The guard's destructor will uninstall the hook. Since all hook classes are derived from CObject and stored as CRef<>-s, the hooks are destroyed automatically when uninstalled. For this reason it's recommended to create hook objects on heap.

Stack Path Hooks

When an object is serialized or deserialized, a string called the stack path is created internally to track the structural context of the current location. The stack path starts with the type name of the top-level data object. While each sub-object is processed, a '.' and the sub-object name are "pushed on the stack".

An example of a possible stack path string is:

```
Seq-entry.set.seq-set.seq.annot.data.ftable.data.pub.pub.article
```

Hooks based on the stack path can be created if you need to specify a more complex structural context for when a hook should be called. More complex, that is, than the "object", "class member", and "choice variant" contexts discussed in earlier sections. For example, "I want to hook the reading of objects named 'title' when and only when they are contained by objects named 'book', not all occurrences of 'title' objects", or, "I want to hook the reading of all sequence members named 'title' in all objects, not only in a specific one". The serial library makes it possible to set hooks for such structural contexts by passing a stack path mask to various "SetHook" methods. When the stack path string for the object being processed matches the stack path mask, the hook will be called.

The general form of the stack path mask is:

```
TypeName.Member1.Member2.HookedMember
```

More formally:

```
StackPathMask ::= (TypeName | Wildcard) ('.' (MemberName | Wildcard))+
```

Here TypeName and MemberName are strings; '.' separates path elements; and Wildcard is defined as:

```
Wildcard ::= ('?' | '*')
```

The question mark means "match exactly one path element with any name", while the asterisk means "match one or more path elements with any names".

An example of a possible stack path mask is:

```
*.article.*.authors
```

Note: The first element of the stack path mask must be either a wildcard or the type of the top-level object in the stream. Type names are not permitted anywhere but the first element, which makes stack path masks like "*.Cit-book.*.date" invalid (ASN.1 type names begin with uppercase while member names begin with lowercase).

As with regular serialization hooks, it is possible to install a path hook for a specific object:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).
 SetPathReadHook(in, path, myReadBioseqHook);
```

a member of a sequence object:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).FindMember("inst").
 SetPathReadHook(in, path, myReadSeqinstHook);
```

or a variant of a choice object:

```
CObjectTypeInfo(CSeq_entry::GetTypeInfo()).FindVariant("seq").
 SetPathReadHook(in, path, myReadBioseqHook);
```

Here in is a pointer to an input object stream. If it is equal to zero, the hook will be installed globally, otherwise - for that particular stream.

In addition, it is possible to install path hooks directly in object streams without specifying an ASN.1 type. For example, to install a read hook on all string objects named last-name, one could use either this:

```
CObjectTypeInfo(CStdTypeInfo<string>::GetTypeInfo()).
 SetPathReadHook(in,"*.last-name",myObjHook);
```

or this:

```
in->SetPathReadObjectHook("*.last-name", myObjHook);
```

Setting path hooks directly in streams also makes it possible to differentiate between last-name being a sequence member and choice variant. So, for example:

```
in->SetPathReadMemberHook("*.last-name", myMemHook);
```

will hook sequence members and not choice variants, while:

```
in->SetPathReadVariantHook("*.last-name", myVarHook);
```

will hook choice variants and not sequence members.

Stack path hooks can be removed by passing NULL instead of a hook pointer to the various "SetHook" methods.

Stream Iterators

When working with a stream, it is sometimes convenient to be able to read or write data elements directly, bypassing the standard data storage mechanism. For example, when reading a large container object, the purpose could be to process its elements. It is possible to read everything at once, but this could require a lot of memory to store the data in. An alternative approach, which greatly reduces the amount of required memory, could be to read elements one by one, process them as they arrive, and then discard. Or, when writing a container, one could construct it in memory only partially, and then add missing elements 'on the fly' - where appropriate. To make it possible, the SERIAL library introduces stream iterators. Needless to say, the most convenient way of using this mechanism is in read/write hooks.

SERIAL library defines the following stream iterator classes: CIStreamClassMemberIterator and CIStreamContainerIterator for input streams, and COStreamClassMember and COStreamContainer for output ones.

Reading a container could look like this:

```
for ( CIStreamContainerIterator it(in, containerType); it; ++it ) {
 CElementClass element;
 it >> element;
}
```

Writing - like this:

```
set<CElementClass> container; // your container
............
COStreamContainer osc(out, containerType);
ITERATE(set<CElementClass>, it, container) {
 const CElementClass& element = *it;
 osc << element;
}
```

For more examples of using stream iterators please refer to asn2asn sample application.

The ByteBlock and CharBlock classes

CObject[IO]Stream::ByteBlock class may be used for non-standard processing of an OCTET STRING data, e.g. from a read/write hooks. The CObject[IO]Stream::CharBlock class has almost the same functionality, but may be used for VisibleString data processing.

An example of using ByteBlock or CharBlock classes is generating data on-the-fly in a write hook. To use block classes:

Initialize the block variable with an i/o stream and, in case of output stream, the length of the block.

Use Read()/Write() functions to process block data

Close the block with the End() function

Below is an example of using CObjectOStream::ByteBlock in an object write hook for non-standard data processing. Note, that ByteBlock and CharBlock classes read/write data only. You should also provide some code for writing class' and members' tags.

Since OCTET STRING and VisibleString in the NCBI C++ Toolkit are implemented as vector<char> and string classes, which have no serailization type info, you can not install a read or write hook for these classes. The example also demonstrates how to process members of these types using the containing class hook. Another example of using CharBlock with write hooks can be found in test_serial.cpp application.

```
void CWriteMyObjectHook::WriteObject(CObjectOStream& out,
 const CConstObjectInfo& object)
{
 const CMyObject& obj = *reinterpret_cast<const CMyObject*>
 (object.GetObjectPtr());
 if ( NothingToProcess(obj) ) {
 // No special processing - use default write method
 DefaultWrite(out, object);
 return;
 }
 // Write object open tag
 out.BeginClass(object.GetClassTypeInfo());
 // Iterate object members
 for (CConstObjectInfo::CMemberIterator member =
 object.BeginMembers(); member; ++member) {
 if ( NeedProcessing(member) ) {
 // Write the special member manually
 out.BeginClassMember(member.GetMemberInfo()->GetId());
 // Start byte block, specify output stream and block size
 size_t length = GetRealDataLength(member);
 CObjectOStream::ByteBlock bb(out, length);
 // Processing and output
 for (int i = 0; i < length; ) {
 char* buf;
 int buf_size;
 // Assuming ProcessData() generates the data from "member",
 // starting from position "i" and stores the data to "buf"
 ProcessData(member, i, &buf_size, &buf);
 i += buf_size;
 bb.Write(buf, buf_size);
 }
 }
 // Close the byte block
 bb.End();
 // Close the member
 out.EndClassMember();
 }
 else {
 // Default writer for members without special processing
 if ( member.IsSet() )
 out.WriteClassMember(member);
 }
```

```
     // Close the object
     out.EndClass();
    }
```

NCBI C++ Toolkit Network Service (RPC) Clients

The following topics are discussed in this section:

- Introduction and Use
- Implementation Details

Introduction and Use

The C++ Toolkit now contains datatool-generated classes for certain ASN.1-based network services: at the time of this writing, Entrez2, ID1, and MedArch. (There is also an independently written class for the Taxon1 service, CTaxon1, which this page does not discuss further.) All of these classes, declared in headers named objects/.../client(_).hpp, inherit certain useful properties from the base template CRPCClient<>:

- They normally defer connection until the first actual query, and disconnect automatically when destroyed, but let users request either action explicitly.
- They are designed to be thread-safe (but, at least for now, maintain only a single connection per instance, so forming pools may be appropriate).

The usual interface to these classes is through a family of methods named AskXxx, each of which takes a request of an appropriate type and an optional pointer to an object that will receive the full reply and returns the corresponding reply choice. For example, CEntrez2Client::AskEval_boolean takes a request of type const CEntrez2_eval_boolean& and an optional pointer of type CEntrez2_reply*, and returns a reply of type CRef<CEntrez2_boolean_reply>. All of these methods automatically detect server-reported errors or unexpected reply choices, and throw appropriate exceptions when they occur. There are also lower-level methods simply named Ask, which may come in handy if you do not know what kind of query you will need to make.

In addition to these standard methods, there are certain class-specific methods: CEntrez2Client adds GetDefaultRequest and SetDefaultRequest for dealing with those fields of Entrez2-request besides request itself, and CID1Client adds {Get,Set}AllowDeadEntries (off by default) to control how to handle the result choice gotdeadseqentry.

Implementation Details

In order to get datatool to generate classes for a service, you must add some settings to the corresponding modulename.def file. Specifically, you must set [-]clients to the relevant base file name (typically service_client), and add a correspondingly named section containing the entries listed in Table 1. (If a single specification defines multiple protocols for which you would like datatool to generate classes, you may list multiple client names, separated by spaces.)

Verification of Class Member Initialization

When serializing an object, it is important to verify that all mandatory primitive data members (e.g. strings, integers) are given a value. The NCBI C++ Toolkit implements this through a data initialization verification mechanism. In this mechanism, the value itself is not validated; that is, it still could be semantically incorrect. The purpose of the verification is only to make sure that the member has been assigned some value. The verification also provides for a possibility to check whether the object data member has been initialized or not. This could be useful when constructing such objects in memory.

From this perspective, each data member (XXX) of a serial object generated by DATATOOL from an ASN or XML specification has the IsSetXXX() and CanGetXXX() methods. Also, input and output streams have SetVerifyData() and GetVerifyData() methods. The purpose of CanGetXXX() method is to answer the question whether it is safe or not to call the corresponding GetXXX(). The meaning of IsSetXXX() is whether the data member has been assigned a value explicitly (using assignment function call, or as a result of reading from a stream) or not. The stream's SetVerifyData() method defines a stream behavior in case it comes across an uninitialized data member.

There are three kinds of object data members:

- optional ones,
- mandatory with a default value,
- mandatory with no default value.

Optional members and mandatory ones with no default have "no value" initially. As such, they are "ungetatable"; that is, GetXXX() throws an exception (this is also configurable though). Mandatory members with a default are always getable, but not always set. It is possible to assign a default value to a mandatory member with a default value. In this case it becomes set, and as such will be written into an output stream.

The discussion above refers only to primitive data members, such as strings, or integers. The behavior of containers is somewhat different. All containers are pre-created on the parent object construction, so for container data members CanGetXXX() always returns TRUE. This can be justified by the fact that containers have a sort of "natural default value" - empty. Also, IsSetXXX() will return TRUE if the container is either mandatory, or has been read (even if empty) from the input stream, or SetXXX() was called for it.

The following additional topics are discussed in this section:

- Initialization Verification in CSerialObject Classes
- Initialization Verification in Object Streams

Initialization Verification in CSerialObject Classes

CSerialObject defines two functions to manage how uninitialized data members would be treated:

```
static void SetVerifyDataThread(ESerialVerifyData verify);
static void SetVerifyDataGlobal(ESerialVerifyData verify);
```

The SetVerifyDataThread() defines the behavior of GetXXX() for the current thread, while the SetVerifyDataGlobal() for the current process. Please note, that disabling CUnassignedMember exceptions in GetXXX() function is potentially dangerous because it could silently return garbage.

The behavior of initialization verification has been designed to allow for maximum flexibility. It is possible to define it using environment variables, and then override it in a program, and vice versa. It is also possible to force a specific behavior, no matter what the program sets, or could set later on. The ESerialVerifyData enumerator could have the following values:

- eSerialVerifyData_Default
- eSerialVerifyData_No
- eSerialVerifyData_Never

- eSerialVerifyData_Yes
- eSerialVerifyData_Always

Setting eSerialVerifyData_Never or eSerialVerifyData_Always results in a "forced" behavior: setting eSerialVerifyData_Never prohibits later attempts to enable verification; setting eSerialVerifyData_Always prohibits attempts to disable it. The default behavior could be defined from the outside, using the SET_VERIFY_DATA_GET environment variable:

```
SET_VERIFY_DATA_GET ::= ( 'NO' | 'NEVER' | 'YES' | 'ALWAYS' )
```

Alternatively, the default behavior can also be set from a program code using CSerialObject::SetVerifyDataXXX() functions.

Setting the environment variable to "Never/Always" overrides any attempt to change the verification behavior in the program. Setting "Never/Always" for the process overrides attempts to change it for a thread. "Yes/No" setting is less restrictive: the environment variable, if present, provides the default, which could then be overridden in a program, or thread. Here thread settings supersede the process ones.

Initialization Verification in Object Streams

Data member verification in object streams is a bit more complex.

First, it is possible to set the verification behavior on three different levels:

- for a specific stream (SetVerifyData()),
- for all streams created by a current thread (SetVerifyDataThread()),
- for all stream created by the current process (SetVerifyDataGlobal()).

Second, there are more options in defining what to do in case of an uninitialized data member:

- throw an exception;
- skip it on writing (write nothing), and leave uninitialized (as is) on reading;
- write some default value on writing, and assign it on reading (even though there is no default).

To accommodate these situations, the ESerialVerifyData enumerator has two additional values:

- eSerialVerifyData_DefValue
- eSerialVerifyData_DefValueAlways

In this case, on reading a missing data member, stream initializes it with a "default" (usually 0); on writing the unset data member, it writes it "as is". For comparison: in the "No/Never" case on reading a missing member stream could initialize it with a "garbage", while on writing it writes nothing. The latter case produces semantically incorrect output, but preserves information of what has been set, and what is not set.

The default behavior could be set similarly to CSerialObject. The environment variables are as follows:

```
SET_VERIFY_DATA_READ ::= ( 'NO' | 'NEVER' | 'YES' | 'ALWAYS' |
'DEFVALUE' | 'DEFVALUE_ALWAYS' )
SET_VERIFY_DATA_WRITE ::= ( 'NO' | 'NEVER' | 'YES' | 'ALWAYS' |
'DEFVALUE' | 'DEFVALUE_ALWAYS' )
```

Simplified Serialization Interface

The reading and writing of serial object requires creation of special object streams which encode and decode data. While such streams provide with a greater flexibility in setting the formatting parameters, in some cases it is not needed - the default behavior is quite enough. NCBI C++ toolkit library makes it possible to use the standard I/O streams in this case, thus hiding the creation of object streams. So, the serialization would look like this:

```
cout << MSerial_AsnText << obj;
```

The only information that is always needed is the output format. It is defined by the following stream manipulators:

- MSerial_AsnText
- MSerial_AsnBinary
- MSerial_Json
- MSerial_Xml

Few additional manipulators define the handling of un-initialized object data members:

- MSerial_VerifyDefault
- MSerial_VerifyNo
- MSerial_VerifyYes
- MSerial_VerifyDefValue

Finding in input stream objects of a specific type

When processing serialized data, it is pretty often that one has to find all objects of a specific type, with this type not being a root one. To make it easier, serial library defines a helper template function Serial_FilterObjects. The idea is to be able to define a special hook class with a single virtual function Process with a single parameter: object of the requested type. Input stream is being scanned then, and, when an object of the requested type is encountered, the user-supplied function is being called.

For example, suppose an input stream contains Bioseq objects, and you need to find and process all Seq-inst objects in it. First, you need to define a class that will process them:

```
Class CProcessSeqinstHook : public
CSerial_FilterObjectsHook<CSeq_inst>
{
public:
 virtual void Process(const CSeq_inst& obj);
};
```

Second, you just call filtering function specifying the root object type:

```
Serial_FilterObjects<CBioseq>(input_stream, new
CProcessSeqinstHook());
```

Another variant of this function – Serial_FilterStdObjects – finds objects of standard type, not derived from CSerialObject – strings, for example. The usage is similar. First, define a hook class that will process data:

```
class CProcessStringHook : public CSerial_FilterObjectsHook<string>
{
public:
 virtual void Process(const string& obj);
};
```

Then, call the filtering function:

```
Serial_FilterStdObjects<CBioseq>(input_stream, new CProcessStringHook());
```

An even more sophisticated, yet easier to use mechanism relies on multi-threading. It puts data reading into a separate thread and hides synchronization issues from client application. There are two template classes, which make it possible: CIStreamObjectIterator and CIStreamStdIterator. The former finds objects of CSerialObject type:

```
CIStreamObjectIterator<CBioseq,CSeq_inst> i(input_stream);
for ( ; i.IsValid(); ++i) {
 const CSeq_inst& obj = *i;
 ...
}
```

The latter – objects of standard type:

```
CIStreamStdIterator<CBioseq,string> i(input_stream);
for ( ; i.IsValid(); ++i) {
 const string& obj = *i;
 ...
}
```

The NCBI C++ Toolkit Iterators

The following topics are discussed in this section:

- STL generic iterators
- CTypeIterator (*) and CTypeConstIterator (*)
- Class hierarchies, embedded objects, and the NCBI C++ type iterators
- CObjectIterator (*) and CObjectConstIterator (*)
- CStdTypeIterator (*) and CStdTypeConstIterator (*)
- CTypesIterator (*)
- Context filtering in type iterators
- Additional Information

STL generic iterators

Iterators are an important cornerstone in the generic programming paradigm - they serve as intermediaries between generic containers and generic algorithms. Different containers have different access properties, and the interface to a generic algorithm must account for this.

The vector class allows input, output, bidirectional, and random access iterators. In contrast, the list container class does **not** allow random access to its elements. This is depicted graphically by one less strand in the ribbon connector. In addition to the iterators, the generic algorithms may require function objects such as less<T> to support the template implementations.

The STL standard iterators are designed to iterate through any STL container of homogeneous elements, e.g., vectors, lists, deques, stacks, maps, multimaps, sets, multisets, etc. A prerequisite however, is that the container must have begin() and end() functions defined on it as start and end points for the iteration.

But while these standard iterators are powerful tools for generic programming, they are of no help in iterating over the elements of aggregate objects - e.g., over the heterogeneous data members of a class object. As this is an essential operation in processing serialized data structures, the NCBI C++ Toolkit provides additional types of iterators for just this purpose. In the section on Runtime object type information, we described the CMemberIterator and CVariantIterator classes, which provide access to the instance and type information for **all** of the sequence members and choice variants of a sequence or choice object. In some cases however, we may wish to visit only those data members which are of a certain type, and do not require any type information. The iterators described in this section are of this type.

CTypeIterator (*) and CTypeConstIterator (*)

The CTypeIterator and CTypeConstIterator can be used to traverse a structured object, stopping at all data members of a specified type. For example, it is very common to represent a linked list of objects by encoding a next field that embeds an object of the same type. One way to traverse the linked list then, would be to "iterate" over all objects of that type, beginning at the head of the list. For example, suppose you have a CPersonclass defined as:

```
class CPerson
{
public:
 CPerson(void);
 CPerson(const string& name, const string& address, CPerson* p);
 virtual ~CPerson(void);
 static const CTypeInfo* GetTypeInfo(void);
 string m_Name, m_Addr;
 CPerson *m_NextDoor;
};
```

Given this definition, one might then define a neighborhood using a single CPerson. Assuming a function FullerBrushMan(CPerson&) must now be applied to each person in the neighborhood, this could be implemented using a CTypeIterator as follows:

```
CPerson neighborhood("Moe", "123 Main St",
 new CPerson("Larry", "127 Main St",
 new CPerson("Curly", "131 Main St", 0)));
for (CTypeIterator<CPerson> house(Begin(neighborhood)); house; ++house ) {
 FullerBrushMan(*house);
}
```

In this example, the data members visited by the iterator are of the same type as the top-level aggregate object, since neighbor is an instance of CPerson. Thus, the first "member" visited is the top-level object itself. This is not always the case however. The top-level object is only included in the iteration when it is an instance of the type specified in the template argument (CPerson in this case).

All of the NCBI C++ Toolkit type iterators are recursive. Thus, since neighborhood has CPerson data members, which in turn contain objects of type CPerson, all of the nested data members will also be visited by the above iterator. More generally, given a hierarchically structured object containing data elements of a given type nested several levels deep, the NCBI C++ Toolkit type iterators effectively generate a "flat" list of all these elements.

It is not difficult to imagine situations where recursive iterators such as the CTypeIterator could lead to infinite loops. An obvious example of this would be a doubly-linked list. For example, suppose CPerson had both previous and next data members, where x->next->previous == x. In this case, visiting x followed by x->next would lead back to x with no terminating condition. To address this issue, the Begin() function accepts an optional second argument, eDetectLoops. eDetectLoops is an enum value which, if included, specifies that the iterator should detect and avoid infinite loops. The resulting iterator will be somewhat slower but can be safely used on objects whose references might create loops.

Let's compare the syntax of this new iterator class to the standard iterators:

```
ContainerType<T> x;
for (ContainerType<T>::IteratorType i = x.begin(); i != x.end(); ++i)
for (CTypeIterator<T> i(Begin(ObjectName)); i; ++i)
```

The standard iterator begins by pointing to the first item in the container x.begin(), and with each iteration, visits subsequent items until the end of the container x.end() is reached. Similarly, the CTypeIterator begins by pointing to the first data member of ObjectName that is of type T, and with each iteration, visits subsequent data members of type T until the end of the top-level object is reached.

A lot of code actually uses = Begin(...) instead of (Begin(...)) to initialize iterators; although the alternate syntax is somewhat more readable and often works, some compilers can mis-handle it and give you link errors. As such, direct initialization as shown above generally works better. Also, note that this issue only applies to construction; you should (and must) continue to use = to reset existing iterators.

How are generic iterators such as these implemented? The Begin() expression returns an object containing a pointer to the input object ObjectName, as well as a pointer to a CTypeInfo object containing type information about that object. On each iteration, the ++ operator examines the **current** type information to find the next data member which is of type T. The current object, its type information, and the state of iteration is pushed onto a local stack, and the iterator is then reset with a pointer to the next object found, and in turn, a pointer to its type information. Each data member of type T (or derived from type T) must be capable of providing its own type information as needed. This allows the iterator to recursively visit all data members of the specified type at all levels of nesting.

More specifically, each object included in the iteration, as well as the initial argument to Begin (), must have a statically implemented GetTypeInfo() class member function to provide the needed type information. For example, all of the serializable objects generated by datatool in the src/objects subtrees have GetTypeInfo() member functions. In order to apply type iterators to user-defined classes (as in the above example), these classes must also make their type information explicit. A set of macros described in the section on User-defined Type Information are provided to simplify the implementation of the GetTypeInfo() methods for user-defined classes. The example included at the end of this section (see Additional Information) uses several of the C++ Toolkit type iterators and demonstrates how to apply some of these macros.

The CTypeConstIterator parallels the CTypeIterator, and is intended for use with const objects (i.e. when you want to prohibit modifications to the objects you are iterating over). For const iterators, the ConstBegin() function should be used in place of Begin().

Class hierarchies, embedded objects, and the NCBI C++ type iterators

As emphasized above, all of the objects visited by an iterator must have the GetTypeInfo() member function defined in order for the iterators to work properly. For an iterator that visits objects of type T, the type information provided by GetTypeInfo() is used to identify:

- data members of type T
- data members containing objects of type T
- data members derived from type T
- data members containing objects derived from type T

Explicit encoding of the class hierarchy via the GetTypeInfo() methods allows the user to deploy a type iterator over a single specified type which may in practice include a set of types via inheritance. The section Additional Information includes a simple example of this feature. The preprocessor macros used in this example which support the encoding of hierarchical class relations are described in the User-defined Type Information section. A further generalization of this idea is implemented by the CTypesIterator described later.

CObjectIterator (*) and CObjectConstIterator (*)

Because the CObject class is so central to the Toolkit, a special iterator is also defined, which can automatically distinguish CObjects from other class types. The syntax of a CObjectIterator is:

```
for (CObjectIterator i(Begin(ObjectName)); i; ++i)
```

Note that there is no need to specify the object type to iterate over, as the type CObject is built into the iterator itself. This iterator will recursively visit all CObjects contained or referenced in ObjectName. The CObjectConstIterator is identical to the CObjectIterator but is designed to operate on const elements and uses the ConstBegin() function.

User-defined classes that are derived from CObject can also be iterated over (assuming their GetTypeInfo() methods have been implemented). In general however, care should be used in applying this type of iterator, as not all of the NCBI C++ Toolkit classes derived from CObject have implementations of the GetTypeInfo() method. **All** of the generated serializable objects in include/objects **do** have a defined GetTypeInfo() member function however, and thus can be iterated over using either a CObjectIterator or a CTypeIterator with an appropriate template argument.

CStdTypeIterator (*) and CStdTypeConstIterator (*)

All of the type iterators described thus far require that each object visited must provide its own type information. Hence, none of these can be applied to standard types such as int, float, double or the STL type string. The CStdTypeIterator and CStdTypeConstIterator classes selectively iterate over data members of a specified type. But for these iterators, the type **must** be a simple C type (int, double, char*, etc.) or an STL type string. For example, to iterate over all the string data members in a CPerson object, we could use:

```
for (CStdTypeIterator<string> i(Begin(neighborhood)); i; ++i) {
 cout << *i << ' ';
}
```

The CStdTypeConstIterator is identical to the CStdTypeIterator but is designed to operate on const elements and requires the ConstBegin() function.

For examples using CTypeIterator and CStdTypeIterator, see Code Sample 2 (ctypeiter.cpp) and Code Sample 3 (ctypeiter.hpp).

CTypesIterator (*)

Sometimes it is necessary to iterate over a set of types contained inside an object. The CTypesIterator, as its name suggests, is designed for this purpose. For example, suppose you have loaded a gene sequence into memory as a CBioseq (named seq), and want to iterate over all of its references to genes and organisms. The following sequence of statements defines an iterator that will step through all of seq's data members (recursively), stopping only at references to gene and organism citations:

```
CTypesIterator i;
CType<CGene_ref>::AddTo(i); // define the types to stop at
CType<COrg_ref>::AddTo(i);

for (i = Begin(seq); i; ++i) {

 if (CType<CGene_ref>::Match(i)) {
 CGene_ref* geneRef = CType<CGene_ref>::Get(i);
 ...
 }
 else if (CType<COrg_ref>::Match(i) {
 COrg_ref* orgRef = CType<COrg_ref>::Get(i);
 ...
 }
}
```

Here, CType is a helper template class that simplifies the syntax required to use the multiple types iterator:

- CType<TypeName>::AddTo(i) specifies that iterator i should stop at type TypeName.
- CType<TypeName>::Match(i) returns true if the specified type TypeName is the type currently pointed to by iterator i.
- CType<TypeName>::Get(i) retrieves the object currently pointed to by iterator i**if** there is a type match to TypeName, and otherwise returns 0. In the event there is a type match, the retrieved object is type cast to TypeName before it is returned.

The Begin() expression is as described for the above CTypeIterator and CTypeConstIterator classes. The CTypesConstIterator is the const implementation of this type of iterator, and requires the ConstBegin() function.

Context Filtering In Type Iterators

In addition to traversing objects of a specific type one might want to specify the structural context in which such objects should appear. For example, you might want to iterate over string data members, but only those called title. This could be done using context filtering. Such a filter is a string with the format identical to the one used in Stack Path Hooks and is specified as an additional parameter of a type iterator. So, for example, the declaration of a string data member iterator with context filtering could look like this:

```
CStdTypeIterator<string> i(Begin(my_obj), "*.title")
```

Additional Information

The following example demonstrates how the class hierarchy determines which data members will be included in a type iterator. The example uses five simple classes:

- Class CA contains a single int data member and is used as a target object type for the type iterators demonstrated.
- class CB contains an auto_ptr to a CA object.
- Class CC is derived from CA and is used to demonstrate the usage of class hierarchy information.
- Class CD contains an auto_ptr to a CC object, and, since it is derived from CObject, can be used as the object pointed to by a CRef.
- Class CX contains both pointers-to and instances-of CA, CB, CC, and CD objects, and is used as the argument to Begin() for the demonstrated type iterators.

The preprocessor macros used in this example implement the GetTypeInfo() methods for the classes, and are described in the section on User-defined type information.

```
// Define a simple class to use as iterator's target objects
class CA
{
public:
 CA() : m_Data(0) {};
 CA(int n) : m_Data(n) {};
 static const CTypeInfo* GetTypeInfo(void);
 int m_Data;
};
// Define a class containing an auto_ptr to the target class
class CB
{
public:
 CB() : m_a(0) {};
 static const CTypeInfo* GetTypeInfo(void);
 auto_ptr<CA> m_a;
};
// define a subclass of the target class
class CC : public CA
{
public:
 CC() : CA(0){};
 CC(int n) : CA(n){};
 static const CTypeInfo* GetTypeInfo(void);
};

// define a class derived from CObject to use in a CRef
// this class also contains an auto_ptr to the target class
class CD : public CObject
{
public:
 CD() : m_c(0) {};
 static const CTypeInfo* GetTypeInfo(void);
 auto_ptr<CC> m_c;
};
// This class will be the argument to the iterator. It contains 4
// instances of CA - directly, through pointers, and via inheritance
class CX
{
public:
 CX() : m_a(0), m_b(0), m_d(0) {};
 ~CX(){};
 static const CTypeInfo* GetTypeInfo(void);
 auto_ptr<CA> m_a; // auto_ptr to a CA
 CB *m_b; // pointer to an object containing a CA
 CC m_c; // instance of a subclass of CA
 CRef<CD> m_d; // CRef to an object containing an auto_ptr to CC
};
////////// Implement the GetTypeInfo() methods /////////
////////// (see User-defined type information) /////////
BEGIN_CLASS_INFO(CA)
{
 ADD_STD_MEMBER(m_Data);
 ADD_SUB_CLASS(CC);
}
END_CLASS_INFO


BEGIN_CLASS_INFO(CB)
{
 ADD_MEMBER(m_a, STL_auto_ptr, (CLASS, (CA)));
}
END_CLASS_INFO


BEGIN_DERIVED_CLASS_INFO(CC, CA)
{
}
END_DERIVED_CLASS_INFO


BEGIN_CLASS_INFO(CD)
{
 ADD_MEMBER(m_c, STL_auto_ptr, (CLASS, (CC)));
}
END_CLASS_INFO


BEGIN_CLASS_INFO(CX)
{
 ADD_MEMBER(m_a, STL_auto_ptr, (CLASS, (CA)));
 ADD_MEMBER(m_b, POINTER, (CLASS, (CB)));
 ADD_MEMBER(m_c, CLASS, (CC));
 ADD_MEMBER(m_d, STL_CRef, (CLASS, (CD)));
}
END_CLASS_INFO

int main(int argc, char** argv)
{
 CB b;
 CD d;

 b.m_a.reset(new CA(2));
 d.m_c.reset(new CC(4));
 CX x;

 x.m_a.reset(new CA(1)); // auto_ptr to CA
 x.m_b = &b; // pointer to CB containing auto_ptr to CA
 x.m_c = *(new CC(3)); // instance of subclass of CA
 x.m_d = &d; // CRef to CD containing auto_ptr to CC

 cout << "Iterating over CA objects in x" << endl << endl;

 for (CTypeIterator<CA> i(Begin(x)); i; ++i)
 cout << (*i).m_Data << endl;

 cout << "Iterating over CC objects in x" << endl << endl;

 for (CTypeIterator<CC> i(Begin(x)); i; ++i)
 cout << (*i).m_Data << endl;

 cout << "Iterating over CObjects in x" << endl << endl;
 for (CObjectIterator i(Begin(x)); i; ++i) {
 const CD *tmp = dynamic_cast<const CD*>(&*i);
 cout << tmp->m_c->m_Data << endl;
 }
 return 0;
}
```

Figure 1 illustrates the paths traversed by CTypeIterator<CA> and CTypeIterator<CC>, where both iterators are initialized with Begin(a). The data members visited by the iterator are indicated by enclosing boxes. See Figure 1.

For additional examples of using the type iterators described in this section, see ctypeiter.cpp.

Processing Serial Data

Although this discussion focuses on ASN.1 and XML formatted data, the data structures and tools described here have been designed to (potentially) support any formalized serial data specification. Many of the tools and objects have open-ended abstract or template implementations that can be instantiated differently to fit various specifications.

The following topics are discussed in this section

- Accessing the object header files and serialization libraries
- Reading and writing serial data

Accessing the object header files and serialization libraries

Reading and writing serialized data is implemented by an integrated set of streams, filters, and object types. An application that reads encoded data files will require the object header files and libraries which define how these serial streams of data should be loaded into memory. This entails #include statements in your source files, as well as the associated library specifications in your makefiles. The object header and implementation files are located in the include/objects and src/objects subtrees of the C++ tree, respectively. The header and implementation files for serialized streams and type information are in the include/serial and src/serial directories.

If you have checked out the objects directories, but not explicitly run the datatool code generator, then you will find that your include/objects subdirectories are (almost) empty, and the source subdirectories contain only makefiles and ASN.1 specifications. These makefiles and ASN.1 specifications can be used to build your own copies of the objects' header and implementation files, using make all_r (if you configured using the --with-objects flag), or running datatool explicitly.

However, building your own local copies of these header and implementation files is neither necessary nor recommended, as it is simpler to use the pre-generated header files and prebuilt libraries. The pre-built header and implementation files can be found in $NCBI/c++/include/objects/ and $NCBI/c++/src/objects/, respectively. Assuming your makefile defines an include path to $NCBI/c++/include, selected object header files such as Date.hpp, can be included as:

```
#include <objects/general/Date.hpp>
```

This header file (along with its implementations in the accompanying src directory) was generated by datatool using the specifications from src/objects/general/general.asn. In order to use the classes defined in the objects directories, your source code should begin with the statements:

```
USING_NCBI_SCOPE;
using namespace objects;
```

All of the objects' header and implementation files are generated by datatool, as specified in the ASN.1 specification files. The resulting object definitions however, are not in any way dependent on ASN.1 format, as they simply specify the in-memory representation of the defined data types. Accordingly, the objects themselves can be used to read, interpret, and write any type of serialized data. Format specializations on the input stream are implemented via CObjectIStream objects, which extract the required tags and values from the input data according to the format specified. Similarly, Format specializations on an output stream are implemented via CObjectOStream objects.

Reading and writing serial data

Let's consider a program xml2asn.cpp that translates an XML data file containing an object of type Biostruc, to ASN.1 text and binary formats. In main(), we begin by initializing the diagnostic stream to write errors to a local file called xml2asn.log. (Exception handling, program tracing, and error logging are described in the Diagnostic Streams section).

An instance of the CTestAsn class is then created, and its member function AppMain() is invoked. This function in turn calls CTestAsn::Run(). The first three lines of code there define the XML input and ASN.1 output streams, using auto_ptr, to ensure automatic destruction of these objects.

Each stream is associated with data serialization mechanisms appropriate to the ESerialDataFormat provided to the constructor:

```
enum ESerialDataFormat {
 eSerial_None = 0,
 eSerial_AsnText = 1, /// ASN.1 text
 eSerial_AsnBinary = 2, /// ASN.1 binary
 eSerial_Xml = 3, /// XML
 eSerial_Json = 4 /// JSON
};
```

CObjectIStream and CObjectOStream are base classes which provide generic interfaces between the specific type information of a serializable object and an I/O stream. The object stream classes that will actually be instantiated by this application, CObjectIStreamXml, CObjectOStreamAsn, and CObjectOStreamAsnBinary, are descendants of these base classes.

Finally, a variable for the object type that will be generated from the input stream (in this case a CBiostruc) is defined, and the CObject[I/O]Stream operators "<<" and ">>" are used to read and write the serialized data to and from the object. (Note that it is **not** possible to simply "pass the data through", from the input stream to the output stream, using a construct like: *inObject >> *outObject). The CObject[I/O]Streams know nothing about the structure of the specific object - they have knowledge only of the serialization format (text ASN, binary ASN, XML, etc.). In contrast, the CBiostruc knows nothing about I/O and serialization formats, but it contains explicit type information about itself. Thus, the CObject[I/O]Streams can apply their specialized serialization methods to the data members of CBiostruc using the type information associated with that object's class.

Reading and writing binary JSON data

JSON is a purely text format - that is, all data values are string representations. Therefore, binary data cannot be serialized or deserialized as JSON without specifying an encoding. Furthermore, the encoding choice is not automatically stored with the encoded data, so the (de) serialization process must explicitly select an encoding.

The following code shows how to read binary JSON data:

```
// Create JSON data with a Base64 encoded binary field.
char jsonb[] = "{ \"Seq_data\": { \"ncbi2na\": \"ASNFZ4mrze8=\" } }";
CNcbiIstrstream iss(jsonb);

// Read the JSON data into a Seq-data object, using Base64 encoding.
CObjectIStreamJson ijson;
ijson.Open(iss);
CSeq_data mySeq_data;
ijson.SetBinaryDataFormat(CObjectIStreamJson::eString_Base64);
ijson >> mySeq_data;
```

The following code shows how to write binary JSON data:

```
// Use ASN.1 data to populate a Seq-data object.
char asn[] = "Seq-data ::= ncbi2na '0123456789ABCDEF'H";
CNcbiIstrstream iss(asn);
auto_ptr<CObjectIStream> in(CObjectIStream::Open(eSerial_AsnText, iss));
CSeq_data mySeq_data;
*in >> mySeq_data;

// Write the Seq-data object in JSON format with Base64 binary encoding.
CObjectOStreamJson ojson(cout, false);
ojson.SetBinaryDataFormat(CObjectOStreamJson::eString_Base64);
ojson << mySeq_data;
```

Determining Which Header Files to Include

As always, we include the corelib header files, ncbistd.hpp and ncbiapp.hpp. In addition, the serial header files that define the generic CObject[IO]Stream objects are included, along with serial.hpp, which defines generalized serialization mechanisms including the insertion (<<) and extraction (>>) operators. Finally, we need to include the header file for the object type we will be using.

There are two source browsers that can be used to locate the appropriate header file for a particular object type. Object class names in the NCBI C++ Toolkit begin with the letter "C". Using the class hierarchy browser, we find CBiostruc, derived from CBiostruc_Base, which is in turn derived from CObject. Following the CBiostruc link, we can then use the locate button to move to the LXR source code navigator, and there, find the name of the header file. In this case, we find CBiostruc.hpp is located in include/objects/mmdb1. Alternatively, if we know the name of the C++ class, the source code navigator's identifier search tool can be used directly. In summary, the following #include statements appear at the top of xml2asn.cpp:

```
#include <corelib/ncbiapp.hpp>
#include <serial/serial.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <objects/mmdb1/Biostruc.hpp>
```

Determining Which Libraries to Link To

Determining which libraries must be linked to requires a bit more work and may involve some trial and error. The list of available libraries currently includes:

access biblio cdd featdef general medlars medline mmdb1 mmdb2 mmdb3 ncbimime objprt proj pub pubmed seq seqalign seqblock seqcode seqfeat seqloc seqres seqset submit xcgi xconnect xfcgi xhtml xncbi xser

It should be clear that we will need to link to the core library, xncbi, as well as to the serial library, xser. In addition, we will need to link to whatever object libraries are entailed by using a CBiostruc object. Minimally, one would expect to link to the mmdb libraries. This in itself is insufficient however, as the CBiostruc class embeds other types of objects, including PubMed citations, features, and sequences, which in turn embed additional objects such as Date. The makefile for xml2asn.cpp, Makefile.xml2asn.app lists the libraries required for linking in the make variable LIB.

```
#############################################################################
# This file was originally generated from by shell script "new_project.sh"
```

```
#############################################################################
APP = xml2asn
OBJ = xml2asn
LIB = mmdb1 mmdb2 mmdb3 seqloc seqfeat pub medline biblio general xser xncbi
LIBS = $(NCBI_C_LIBPATH) -lncbi $(ORIG_LIBS)
```

See also the example program, asn2asn.cpp which demonstrates more generalized translation of Seq-entry and Bioseq-set (defined in seqset.asn).

Note: Two online tools are available to help determine which libraries to link with. See the FAQ for details.

User-defined type information

The following topics are discussed in this section:

- Introduction
- Installing a GetTypeInfo() function: the BEGIN_/END_macros
- Specifying internal structure and class inheritance: the ADD_ macros

Introduction

Object type information, as it is used in the NCBI C++ Toolkit, is defined in the section on Runtime Object Type Information. As described there, all of the classes and constructs defined in the serial include and src directories have a static implementation of a GetTypeInfo() function that yields a CTypeInfo for the object of interest. In this section, we describe how type information can also be generated and accessed for user-defined types. We begin with a review of some of the basic notions introduced in the previous discussion.

The type information for a class is stored outside any instances of that class, in a statically created CTypeInfo object. A class's type information includes the class layout, inheritance relations, external alias, and various other attributes that are independent of specific instances. In addition, the type information object provides an interface to the class's data members.

Limited type information is also available for primitive data types, enumerations, containers, and pointers. The type information for a primitive type specifies that it is an int, float, or char, etc., and whether or not that element is signed. Enumerations are a special kind of primitive type, whose type information specifies its enumeration values and named elements. Type information for containers can specify both the type of container and the type of elements. The type information for a pointer provides convenient methods of access to the type information for the type pointed to.

For all types, the type information is encoded in a static CTypeInfo object, which is then accessed by all instances of a given type using a GetTypeInfo() function. For class types, this function is implemented as a static method for the class. For non class types, GetTypeInfoXxx () is implemented as a static global function, where *Xxx* is a unique suffix generated from the type's name. With the first invocation of GetTypeInfo() for a given type, the static CTypeInfo object is created, which then persists (local to the function GetTypeInfo()) throughout execution. Subsequent calls to GetTypeInfo() simply return a pointer to this statically created local object.

In order to make type information about user-defined classes accessible to your application, the user-defined classes must also implement a static GetTypeInfo() method. A set of preprocessor macros is available, which greatly simplifies this effort. A pre-requisite to using these macros however, is that the class definition must include the following line:

```
DECLARE_INTERNAL_TYPE_INFO();
```

This pre-processor macro will generate the following in-line statement in the class definition:

```
static const NCBI_NS_NCBI::CTypeInfo* GetTypeInfo(void);
```

As with class objects, there must be some means of declaring the type information function for an enumeration prior to using the macros which implement that function. Given an enumeration named EMyEnum, DECLARE_ENUM_INFO(EMyEnum) will generate the following declaration:

```
const CEnumeratedTypeValues* GetTypeInfo_enum_EMyEnum(void);
```

The DECLARE_ENUM_INFO() macro should appear in the header file where the enumeration is defined, immediately following the definition. The DECLARE_INTERNAL_ENUM_INFO macro is intended for usage with internal class definitions, as in:

```
class ClassWithEnum {
 enum EMyEnum {
 ...
 };

 DECLARE_INTERNAL_ENUM_INFO(EMyEnum);
 ...
};
```

The C++ Toolkit also allows one to provide type information for legacy C style struct and choice elements defined in the C Toolkit. The mechanisms used to implement this are mentioned but not described in detail here, as it is not likely that newly-defined types will be in these categories.

Installing a GetTypeInfo() function: the BEGIN_/END_macros

Several pre-processor macros are available for the installation of the GetTypeInfo() functions for different types. Table 2 lists six BEGIN_NAMED_*_INFO macros, along with a description of the type of object each can be applied to and its expected arguments. Each macro in Table 2 has a corresponding END_*_INFO macro definition.

The first four macros in Table 2 apply to C++ objects. The DECLARE_INTERNAL_TYPE_INFO() macro **must** appear in the class definition's public section. These macros take two string arguments:

- an external alias for the type, and
- the internal C++ symbolic class name

The next two macros implement global, uniquely named functions which provide access to type information for C++ enumerations; the resulting functions are named GetTypeInfo_enum_[EnumName]. The DECLARE_ENUM_INFO() or DECLARE_ENUM_INFO_IN() macro should be used in these cases to declare the GetTypeInfo*() functions.

The usage of these six macros generally takes the following form:

```
BEGIN_*_INFO(ClassName)
{
 ADD_*(MemberName1);
 ADD_*(MemberName2);
 ...
}
END_*_INFO
```

That is, the BEGIN/END macros are used to generate the function's signature and enclosing block, and various ADD_* macros are applied to add information about internal members and class relations.

List of the BEGIN_/END_ macros

- BEGIN_NAMED_CLASS_INFO (ClassAlias, ClassName)
- BEGIN_CLASS_INFO (ClassName)

These macros should be used on classes that do not contain any pure virtual functions. For example, the GetTypeInfo() method for the CPerson class (used in the chapter on iterators) can be implemented as:

```
BEGIN_NAMED_CLASS_INFO("CPerson", CPerson)
{
 ADD_NAMED_STD_MEMBER("m_Name", m_Name);
 ADD_NAMED_STD_MEMBER("m_Addr", m_Addr);
 ADD_NAMED_MEMBER("m_NextDoor", m_NextDoor, POINTER, (CLASS, (CPerson)));
}
END_CLASS_INFO
```

or, equivalently, as:

```
BEGIN_CLASS_INFO(CPerson)
{
 ADD_STD_MEMBER(m_Name);
 ADD_STD_MEMBER(m_Addr);
 ADD_MEMBER(m_NextDoor, POINTER, (CLASS, (CPerson)));
}
END_CLASS_INFO
```

Here, the CPerson class has two string data members, m_Name and m_Addr, as well as a pointer to an object of the same type (CPerson*). All built-in C++ types such as int, float, string etc., use the ADD_NAMED_STD_MEMBER or ADD_STD_MEMBER macros. These and other macros used to add members are defined in Specifying internal structure and class inheritance: the ADD_ macros and Table 3.

- BEGIN_NAMED_ABSTRACT_CLASS_INFO(ClassAlias, ClassName)
- BEGIN_ABSTRACT_CLASS_INFO(ClassName)

These macros must be used on abstract base classes which contain pure virtual functions. Because these abstract classes cannot be instantiated, special handling is required in order to install their static GetTypeInfo() methods.

- BEGIN_NAMED_DERIVED_CLASS_INFO (ClassAlias, ClassName, BaseClassName)
- BEGIN_DERIVED_CLASS_INFO (ClassName, BaseClassName)

These macros should be used on derived subclasses whose parent classes also have the GetTypeInfo() method implemented. Data members inherited from parent classes should not be included in the derived class type information.

```
BEGIN_DERIVED_CLASS_INFO(CA, CBase)
{
 // ... data members in CA not inherited from CBase
}
END_DERIVED_CLASS_INFO
```

NOTE:The type information for classes derived directly from CObject does **not** however, follow this protocol. In this special case, although the class is derived from CObject, you should **not** use the DERIVED_CLASS macros to implement GetTypeInfo(), but instead use the usual BEGIN_CLASS_INFO macro. CObject's have a slightly different interface to their type information (see CObjectGetTypeInfo), and apply these macros differently.

- BEGIN_NAMED_CHOICE_INFO (ClassAlias, ClassName)
- BEGIN_CHOICE_INFO (ClassName)

These macros install GetTypeInfo() for C++choice objects, which are implemented as C++ classes. See Choice objects in the C++ Toolkit for a description of C++ choice objects. Each of the choice variants occurs as a data member in the class, and the macros used to add choice variants (ADD_NAMED_*_CHOICE_VARIANT) are used similarly to those which add data members to classes (see discussion of the ADD* macros below).

- BEGIN_NAMED_ENUM_INFO (EnumAlias, EnumName, IsInteger)
- BEGIN_ENUM_INFO (EnumName, IsInteger)

In addition to the two arguments used by the BEGIN_*_INFO macros for classes, a Boolean argument (IsInteger) indicates whether or not the enumeration includes arbitrary integer values or only those explicitly specified.

- BEGIN_NAMED_ENUM_IN_INFO (EnumAlias, CppContext, EnumName, IsInteger)
- BEGIN_ENUM_IN_INFO (CppContext, EnumName, IsInteger)

These macros also implement the type information functions for C++ enumerations --but in this case, the enumeration is defined outside the scope where the macro is applied, so a context argument is required. This new argument, CppContext, specifies the C++ class name or external namespace where the enumeration is defined.

Again, when using the above macros to install type information, the corresponding class definitions **must** include a declaration of the static class member function GetTypeInfo() in the class's public section. The DECLARE_INTERNAL_TYPE_INFO() macro is provided to ensure that the declaration of this method is correct. Similarly, the DECLARE_INTERNAL_ENUM_INFO and DECLARE_ENUM_INFO macros should be used in the header files where enumerations are defined. The DECLARE_ASN_TYPE_INFO and DECLARE_ASN_CHOICE_INFO macros can be used to declare the type information functions for C-style structs and choice nodes.

Specifying internal structure and class inheritance: the ADD_ macros

Information about internal class structure and inheritance is specified using the ADD_* macros (see Table 3). Again, each macro has both a "named" and "unnamed" implementation. The arguments to all of the ADD_NAMED_* macros begin with the external alias and C++ name of the item to be added.

The ADD_* macros that take **only** an alias and a name require that the type being added must be either a built-in type or a type defined by the name argument. When adding a CRef data member to a class or choice object however, the class referenced by the CRef must be made explicit with the RefClass argument, which is the C++ class name for the type pointed to.

Similarly, when adding an enumerated data member to a class, the enumeration itself must be explicitly named. For example, if class CMyClass contains a data member m_MyEnumVal of type EMyEnum, then the BEGIN_NAMED_CLASS_INFO macro for CMyClass should contain the statement:

```
ADD_ENUM_MEMBER (m_MyEnumVal, EMyEnum);
```

or, equivalently:

```
ADD_NAMED_ENUM_MEMBER ("m_MyEnumVal", m_MyEnumVal, EMyEnum);
```

or, to define a "custom" (non-default) external alias:

```
ADD_NAMED_ENUM_MEMBER ("m_CustomAlias", m_MyEnumVal, EMyEnum);
```

Here, EMyEnum is defined in the same namespace and scope as CMyClass. Alternatively, if the enumeration is defined in a different class or namespace (and therefore, then the ADD_ENUM_IN_MEMBER macro must be used:

```
ADD_ENUM_IN_MEMBER (m_MyEnumVal, COtherClassName::, EMyEnum);
```

In this example, EMyEnum is defined in a class named COtherClassName. The CppContext argument (defined here as COtherClassName::) acts as a scope operator, and can also be used to specify an alternative namespace. The ADD_NAMED_ENUM_CHOICE_VARIANT and ADD_NAMED_ENUM_IN_CHOICE_VARIANT macros are used similarly to provide information about enumerated choice options. The ADD_ENUM_VALUE macro is used to add enumerated values to the enumeration itself, as demonstrated in the above example of the BEGIN_NAMED_ENUM_INFO macro.

The most complex macros by far are those which use the TypeMacro and TypeMacroArgs arguments: ADD(_NAMED)_MEMBER and ADD(_NAMED)_CHOICE_VARIANT. These macros are more open-ended and allow for more complex specifications. We have already seen one example of using a macro of this type, in the implementation of the GetTypeInfo() method for CPerson:

```
ADD_MEMBER(m_NextDoor, POINTER, (CLASS, (CPerson)));
```

The ADD_MEMBER and ADD_CHOICE_VARIANT macros always take at least two arguments:

the internal member (variant) name

the definition of the member's (variant's) type

Depending on the (second) TypeMacro argument, additional arguments may or may not be needed. In this example, the TypeMacro is *POINTER*, which **does require** additional arguments. The TypeMacroArgs here specify that m_NextDoor is a pointer to a class type whose C++ name is CPerson.

More generally, the remaining arguments depend on the value of TypeMacro, as these parameters complete the type definition. The possible strings which can occur as TypeMacro, along with the additional arguments required for that type, are given in Table 4.

The ADD_MEMBER macro generates a call to the corresponding ADD_NAMED_MEMBER macro as follows:

```
#define ADD_MEMBER(MemberName,TypeMacro,TypeMacroArgs) \
 ADD_NAMED_MEMBER(#MemberName,MemberName,TypeMacro,TypeMacroArgs)
```

Some examples of using the ADD_MEMBER macro are:

```
ADD_MEMBER(m_X);
ADD_MEMBER(m_A, STL_auto_ptr, (CLASS, (ClassName)));
ADD_MEMBER(m_B, STL_CHAR_vector, (char));
ADD_MEMBER(m_C, STL_vector, (STD, (int)));
ADD_MEMBER(m_D, STL_list, (CLASS, (ClassName)));
ADD_MEMBER(m_E, STL_list, (POINTER, (CLASS, (ClassName))));
ADD_MEMBER(m_F, STL_map, (STD, (long), STD, (string)));
```

Similarly, the ADD_CHOICE_VARIANT macro generates a call to the corresponding ADD_NAMED_CHOICE_VARIANT macro. These macros add type information for the choice object's variants.

Runtime Object Type Information

The following topics are discussed in this section:

- Introduction
- Motivation
- Object Information Classes
- Usage of object type information

Introduction

Run-time information about data types is necessary in several contexts, including:

1. When reading, writing, and processing serialized data, where runtime information about a type's internal structure is needed.
2. When reading from an arbitrary data source, where data members' external aliases must be used to locate the corresponding class data members (e.g.*MyXxx* may be aliased as *my-xxx* in the input data file).
3. When using a generalized C++ type iterator to traverse the data members of an object.
4. When accessing the object type information *per se* (without regard to any particular object instance), e.g. to dump it to a file as ASN.1 or DTD specifications (not data).

In the first three cases above, it is necessary to have both the object itself as well as its runtime type information. This is because in these contexts, the object is usually passed inside a generic function, as a pointer to its most base parent type CObject. The runtime type information is needed here, as there is no other way to ascertain the actual object's data members. In addition to providing this information, a runtime type information object provides an interface for accessing and modifying these data members.

In case (4) above, the type information is used independent of any actual object instances.

Type and Object specific info

The NCBI C++ Toolkit uses two classes to support these requirements:

- **Type information classes** (base class CTypeInfo) are intended for internal usage only, and they encode information about a type, devoid of any instances of that type. This information includes the class layout, inheritance relations, external alias, and various other attributes such as size, which are independent of specific instances. Each data member of a class also has its own type information. Thus, in addition to providing information relevant to the member's occurrence in the class (e.g. the member name and offset), the type information for a class must also provide access to the type information for each of its members. Limited type information is also available for types other than classes, such as primitive data types, enumerations, containers, and pointers. For example, the type information for a primitive type specifies that it is an int, float, or char, etc., and whether or not that element is signed. Enumerations are a special kind of primitive type, whose type information specifies its enumeration values and named elements. Type information for containers specifies both the type of container and the type of elements that it holds.
- Object information classes (base class CObjectTypeInfo) include a pointer to the type information as well as a pointer to the object instance, and provide a safe interface to that object. In situations where type information is used independent of any concrete object, the object information class simply serves as a wrapper to a type information object. Where access to an object instance is required, the object pointer provides direct access to the correctly type-cast instance, and the interface provides methods to access and/or modify the object itself or members of that object.

The C++ Toolkit stores the type information outside any instances of that type, in a statically created CTypeInfo object. For class objects, this CTypeInfo object can be accessed by all instances of the class via a static GetTypeInfo() class method. Similarly, for primitive types and other constructs that have no way of associating methods with them per se, a static globally defined GetTypeInfoXxx() function is used to access a static CTypeInfo object. (The *Xxx* suffix is used here to indicate that a globally unique name is generated for the function).

All of the automatically generated classes and constructs defined in the C++ Toolkit's objects/ directory already have static GetTypeInfo() functions implemented for them. In order to make type information about user-defined classes and elements also accessible, you will need to implement static GetTypeInfo() functions for these constructs. A number of pre-processor macros are available to support this activity, and are described in the section on User-defined Type Information.

Type information is often needed when the object itself has been passed anonymously, or as a pointer to its parent class. In this case, it is not possible to invoke the GetTypeInfo() method directly, as the object's exact type is unknown. Using a <static_cast> operator to enable the member function is also unsafe, as it may open the door to incorrectly associating an object's pointer with the wrong type information. For these reasons, the CTypeInfo class is intended for internal usage only, and it is the CObjectTypeInfo classes that provide a more safe and friendly user interface to type information.

Motivation

We use a simple example to help motivate the use of this type and object information model. Let us suppose that we would like to have a generic function LoadObject(), which can populate an object using data read from a flat file. For example, we might like to have:

```
bool LoadObject(Object& myObj, istream& is);
```

where myObj is an instance of some subclass of Object. Assuming that the text in the file is of the form:

```
MemberName1 value1
MemberName5 value5
MemberName2 value2
:
```

we would like to find the corresponding data member in myObj for each MemberName, and set that data member's value accordingly. Unfortunately, myObj cannot directly supply any useful type information, as the member names we seek are for a specific subclass of Object. Now suppose that we have an appropriate type information object available for myObj, and consider how this might be used:

```
bool LoadObject(TypeInfo& info, Object& myObj, istream& is)
{
 string myName, myValue;

 while ( !is.eof() ) {
 is >> myName >> myValue;
 void* member = FindMember(info, myObj, myName);
 AssignValue(member, myValue);
 }
}
```

Here, we assume that our type information object, info, stores information about the memory offset of each data member in myObj, and that such information can be retrieved using some sort of identifying member name such as myName. This is not too difficult to imagine, and indeed, this is exactly the type of information and facility provided by the C++ Toolkit's type information classes. The FindMember() function just needs to return a void pointer to the appropriate location in memory. The AssignValue() function presents a much greater challenge however, as its two sole arguments are a void pointer and a string. This would be fine if the data member was indeed a void pointer, and a string value was acceptable. In general this is not the case, and stronger methods are clearly needed.

In particular, for each data member encountered, we need to retrieve the type of that member as well as its location in memory, so as to process myValue appropriately before assigning it. In addition, we need safer mechanisms for making such "untyped" assignments. Ideally, we would like a FindMember() function that returns a correctly cast pointer to that data member, along with its associated type information. This is what the object information classes provide - a pointer to the object instance as well as a pointer to its static type information. The interface to the object information class also provides a number of methods such as GetClassMember (), GetTypeFamily(), SetPrimitiveValue(), etc., to support the type of activity described above.

Object Information Classes

The following topics are discussed in this section:

- CObjectTypeInfo (*)
- CConstObjectInfo (*)
- CObjectInfo (*)

CObjectTypeInfo (*)

This is the base class for all object information classes. It is intended for usage where there is no concrete object being referenced, and all that is required is access to the type information. A CObjectTypeInfo contains a pointer to a low-level CTypeInfo object, and functions as a user-friendly wrapper class.

The constructor for CObjectTypeInfo takes a pointer to a const CTypeInfo object as its single argument. This is precisely what is returned by all of the static GetTypeInfo() functions. Thus, to create a CObjectTypeInfo for the CBioseq class - without reference to any particular instance of CBioseq - one might use:

CObjectTypeInfo objInfo(CBioseq::GetTypeInfo());

One of the most important methods provided by the CObjectTypeInfo class interface is GetTypeFamily(), which returns an enumerated value indicating the type family for the object of interest. Five type families are defined by the ETypeFamily enumeration:

```
ETypeFamily GetTypeFamily(void) const;
 enum ETypeFamily {
 eTypeFamilyPrimitive,
 eTypeFamilyClass,
 eTypeFamilyChoice,
 eTypeFamilyContainer,
 eTypeFamilyPointer
};
```

Different queries become appropriate depending on the ETypeFamily of the object. For example, if the object is a container, one might need to determine the type of container (e.g. whether it is a list, map etc.), and the type of element. Similarly, if an object is a primitive type (e.g. int, float, string, etc.), an appropriate query becomes what the value type is, and in the case of integer-valued types, whether or not it is signed. Finally, in the case of more complex objects such as class and choice objects, access to the type information for the individual data members and choice variants is needed. The following methods are included in the CObjectTypeInfo interface for these purposes:

For objects with family type eTypeFamilyPrimitive:

```
EPrimitiveValueType GetPrimitiveValueType(void) const;
bool IsPrimitiveValueSigned(void) const;
```

For objects with family type eTypeFamilyClass:

```
CMemberIterator BeginMembers(void) const;
CMemberIterator FindMember(const string& memberName) const;
CMemberIterator FindMemberByTag(int memberTag) const;
```

For objects with family type eTypeFamilyChoice:

```
CVariantIterator BeginVariants(void) const;
CVariantIterator FindVariant(const string& memberName) const;
CVariantIterator FindVariantByTag(int memberTag) const;
```

For objects with family type eTypeFamilyContainer:

```
EContainerType GetContainerType(void) const;
CObjectTypeInfo GetElementType(void) const;
```

For objects with family type eTypeFamilyPointer:

```
CObjectTypeInfo GetPointedType(void) const;
```

The two additional enumerations referred to here, EContainerType and EPrimitiveValueType, are defined, along with ETypeFamily, in include/serial/serialdef.hpp.

Different iterator classes are used for iterating over class data members versus choice variant types. Thus, if the object of interest is a C++ class object, then access to the type information for its members can be gained using a CObjectTypeInfo::CMemberIterator. The BeginMembers() method returns a CMemberIterator pointing to the first data member in the class; the FindMember*() methods return a CMemberIterator pointing to a data member whose name or tag matches the input argument. The CMemberIterator class is a forward iterator whose operators are defined as follows:

- the ++ operator increments the iterator (makes it point to the next class member)
- the () operator tests that the iterator has not exceeded the legitimate range
- the * dereferencing operator returns a CObjectTypeInfo for the data member the iterator currently points to

Similarly, the BeginVariants() and FindVariant() methods allow iteration over the choice variant data types for a choice class, and the dereferencing operation yields a CObjectTypeInfo object for the choice variant currently pointed to by the iterator.

CConstObjectInfo (*)

The CConstObjectInfo (derived from CObjectTypeInfo) adds an interface to access the particular instance of an object (in addition to the interface inherited from CObjectTypeInfo, which provides access to type information only). It is intended for usage with const instances of the object of interest, and therefore the interface does not permit any modifications to the object. The constructor for CConstObjectInfo takes two arguments:

```
CConstObjectInfo(const void* instancePtr, const CTypeInfo* typeinfoPtr);
```

(Alternatively, the constructor can be invoked with a single STL pair containing these two objects.)

Each CConstObjectInfo contains a pointer to the object's type information as well as a pointer to an instance of the object. The existence or validity of this instance can be checked using any of the following CConstObjectInfo methods and operators:

- bool Valid(void) const;
- operator bool(void) const;
- bool operator!(void) const;

For primitive type objects, the CConstObjectInfo interface provides access to the currently assigned value using GetPrimitiveValueXxx(). Here, Xxx may be Bool, Char, Long, ULong, Double, String, ValueString, or OctetString. In general, to get a primitive value, one first applies a switch statement to the value returned by GetPrimitiveValueType(), and then calls the appropriate GetPrimitiveValueXxx() method depending on the branch followed, e.g.:

```
switch ( obj.GetPrimitiveValueType() ) {
case ePrimitiveValueBool:
 bool b = obj.GetPrimitiveValueBool();
 break;

case ePrimitiveValueInteger:
 if ( obj.IsPrimitiveValueSigned() ) {
 long l = obj.GetPrimitiveValueLong();
 } else {
 unsigned long ul = obj.GetPrimitiveValueULong();
 }
 break;
 //... etc.
}
```

Member iterator methods are also defined in the CConstObjectInfo class, with a similar interface to that found in the CObjectTypeInfo class. In this case however, the dereferencing operators return a CConstObjectInfo object - not a CObjectTypeInfo object - for the current member. For C++class objects, these member functions are:

- CMemberIterator BeginMembers(void) const;
- CMemberIterator FindClassMember(const string& memberName) const;
- CMemberIterator FindClassMemberByTag(int memberTag) const;

For C++ choice objects, only one variant is ever selected, and only that choice variant is instantiated. As it does not make sense to define a CConstObjectInfo iterator for uninstantiated variants, the method GetCurrentChoiceVariant() is provided instead. The dereferencing operator (*) can be applied to the object returned by this method to obtain a CConstObjectInfo for the variant. Of course, type information for unselected variants can still be accessed using the CObjectTypeInfo methods.

The CConstObjectInfo class also defines an element iterator for container type objects. CConstObjectInfo::CElementIterator is a forward iterator whose interface includes increment and testing operators. Dereferencing is implemented by the iterator's GetElement() method, which returns a CConstObjectInfo for the element currently pointed to by the iterator.

Finally, for pointer type objects, the type returned by the method GetPointedObject() is also a CConstObjectInfo for the object - not just a CObjectTypeInfo.

CObjectInfo (*)

The CObjectInfo class is in turn derived from CConstObjectInfo, and is intended for usage with mutable instances of the object of interest. In addition to all of the methods inherited from the parent class, the interface to this class also provides methods that allow modification of the object itself or its data members.

For primitive type objects, a set of SetPrimitiveValueXxx() methods are available, complimentary to the GetPrimitiveValueXxx() methods described above. Methods that return member iterator objects are again reimplemented, and the de-referencing operators now return a CObjectInfo object for that data member. As the CObjectInfo now points to a mutable object, these iterators can be used to set values for the data member. Similarly, GetCurrentChoiceVariant() now returns a CObjectInfo, as does CObjectInfo::CElementIterator::GetElement().

Usage of object type information

We can now reconsider how our LoadObject() function might be implemented using the CObjectInfo class:

```
bool LoadObject(CObjectInfo& info, CNcbiIStream& is)
{
 string alias, myValue;

 while ( !is.eof() ) {
 is >> alias >> myValue;

 CObjectInfo dataMember(*info.FindClassMember(alias));
 if (!dataMember) {
 ERR_POST_X(1, "Couldn't find member named:" << alias);
 }
 SetValue(dataMember, myValue);
 }
}
```

Here, info contains pointers to the CObject itself as well as to its associated CTypeInfo object. For each member alias read from the file, we apply FindClassMember(alias), and dereference the returned iterator to retrieve a CObjectInfo object for that member. We then use the operator () to verify that the member was located, and if so, use the member's CObjectInfo to set a value in the function SetValue():

```
void SetValue(const CObjectInfo& obj, const string value)
{
 if (obj.GetTypeFamily() == eTypeFamilyPrimitive) {

 switch ( obj.GetPrimitiveValueType() ) {

 case ePrimitiveValueBool:
 obj.SetPrimitiveValueBool (atoi (value.c_str()));
 break;

 case ePrimitiveValueChar:
 obj.SetPrimitiveValueChar (value.c_str()[0]);
 break;

 //... etc
 }
 } else {
 ERR_POST_X(2, "Attempt to assign non-primitive from string:" << value);
 }
}
```

In this example, SetValue() can only assign primitive types. More generally however, the CObjectInfo class allows the assignment of more complex types that are simply not implemented here. Note also that the arguments to SetValue() are const, even though the function **does** modify the value of the data instance pointed to. In particular, the type const CObjectInfo should not be confused with the type CConstObjectInfo. The former specifies that object information construct is non-mutable, although the instance it points to can be modified. The latter specifies that the instance itself is non-mutable.

In addition to user-specific applications of the type demonstrated in this example, the generic implementations of the C++ type iterators and the CObject[IO]Streamclass methods provide excellent examples of how runtime object type information can be deployed.

As a final example of how type information might be used, we consider an application whose simple task is to translate a data file on an input stream to a different format on an output stream. One important use of the object classes defined in include/objects is the hooks and parsing mechanisms available to applications utilizing CObject[IO]Streams. The stream objects specialize in different formats (such as XML or ASN.1), and must work in concert with these type-specific object classes to interpret or generate serialized data. In some cases however, the dynamic memory allocation required for large objects may be substantial, and it is preferable to avoid actually instantiating a whole object all at once.

Instead, it is possible to use the CObjectStreamCopier class, described in CObject[IO] Streams. Briefly, this class holds two CObject[IO]Stream data members pointing to the input and output streams, and a set of Copy methods which take a CTypeInfo argument. Using this class, it is easy to translate files between different formats; for example:

```
auto_ptr<CObjectIStream> in(CObjectIStream::Open("mydata.xml",eSerial_Xml));
auto_ptr<CObjectOStream> out(CObjectOStream::Open
("mydata.asn",eSerial_AsnBinary));
CObjectStreamCopier copier(*in, *out);
copier.Copy (CBioseq_set::GetTypeInfo());
```

copies a CBioseq_set encoded in XML to a new file, reformatted in ASN.1 binary format.

Choice objects in the NCBI C++ Toolkit

The following topics are discussed in this section:

- Introduction
- C++ choice objects

Introduction

The datatool program processes the ASN.1 specification files (*.asn) in the src/objects/ directories to generate the associated C++ class definitions. The corresponding program implemented in the C Toolkit, asntool, used the ASN.1 specifications to generate C enums, structs, and functions. In contrast, datatool must generate C++ enums, classes and methods. In addition, for each defined object type, datatool must also generate the associated type information method or function.

There is a significant difference in how these two tools implement ASN.1 choice elements. As an example, consider the following ASN.1 specification:

```
Object-id ::= CHOICE {
 id INTEGER,
 str VisibleString
}
```

The ASN.1 choice element specifies that the corresponding object may be any one of the listed types. In this case, the possible types are an integer and a string. The approach used in asntool was to implement all choice objects as ValNodes, which were in turn defined as:

```
typedef struct valnode {
 unsigned choice;
 DataVal data;
 struct valnode *next;
} ValNode;
```

The DataVal field is a union, which may directly store numerical values, or alternatively, hold a void pointer to a character string or C struct. Thus, to process a choice element in the C Toolkit, one could first retrieve the choice field to determine how the data should be interpreted, and subsequently, retrieve the data via the DataVal field. In particular, no explicit implementation of individual choice objects was used, and it was left to functions which manipulate these elements to enforce logical consistency and error checking for legitimate values. A C struct which included a choice element as one of its fields merely had to declare that element as type *ValNode*. This design was further complicated by the use of a void pointer to store non-primitive types such as structs or character strings.

In contrast, the C++ datatool implementation of choice elements defines a class with built-in, automatic error checking for each choice object. The usage of CObject class hierarchy (and the associated type information methods) solves many of the problems associated with working with void pointers.

C++ choice objects

The classes generated by datatool for choice elements all have the following general structure:

```
class C[AsnChoiceName] : public CObject
{
public:
 ... // constructors and destructors
 DECLARE_INTERNAL_TYPE_INFO(); // declare GetTypeInfo() method
 enum E_Choice { // enumerate the class names
 e_not_set, // for the choice variants
 e_Xxx,
 ...
 };
 typedef CXxx TXxx; // typedef each variant class
 ...
 virtual void Reset(void); // reset selection to none
 E_Choice Which(void) const; // return m_choice
 void Select(E_Choice index, // change the current selection
 EResetVariant reset);
 static string SelectionName(E_Choice index);
 bool IsXxx(void) const; // true if m_choice == eXxx
 CXxx& GetXxx(void);
 const CXxx& GetXxx(void) const;
 CXxx& SetXxx(void);
 void SetXxx(const CRef<CXxx>& ref);
 ...
private:
 E_Choice m_choice; // choice state
 union {
 TXxx m_Xxx;
 ...
 };
 CObject *m_object; // variant's data
 ...
};
```

For the above ASN.1 specification, datatool generates a class named CObject_id, which is derived from CObject. For each choice variant in the specification, an enumerated value (in E_Choice), and an internal typedef are defined, and a declaration in the union data member is made. For this example then, we would have:

```
enum E_Choice {
 e_not_set,
 e_Id,
 e_Str
};
...
typedef int TId;
typedef string TStr;
...
union {
 TId m_Id;
 string *m_string;
};
```

In this case both of the choice variants are C++ built-in types. More generally however, the choice variant types may refer to any type of object. For convenience, we refer to their C++ type names here as "CXxx",

Two private data members store information about the currently selected choice variant: m_choice holds the enum value, and m_Xxx holds (or points to a CObject containing) the variant's data. The choice object's member functions provide access to these two data members. Which() returns the currently selected variant's E_Choice enum value. Each choice variant has its own Get() and Set() methods. Each GetXxx() method throws an exception if the variant type for that method does not correspond to the current selection type. Thus, it is not possible to unknowingly retrieve the incorrect type of choice variant.

Select(e_Xxx) uses a switch(e_Xxx) statement to initialize m_Xxx appropriately, sets m_choice to e_Xxx, and returns. Two SetXxx() methods are defined, and both use this Select () method. SetXxx() with no arguments calls Select(e_Xxx) and returns m_Xxx (as initialized by Select()). SetXxx(TXxx& value) also calls Select(e_Xxx) but resets m_Xxx to value before returning.

Some example choice objects in the C++ Toolkit are:

- CDate
- CInt_fuzz
- CObject_id
- CPerson_id
- CAnnotdesc
- CSeq_annot

Traversing a Data Structure

The following topics are discussed in this section:

- Locating the Class Definitions
- Accessing and Referencing Data Members
- Traversing a Biostruc
- Iterating Over Containers

Locating the Class Definitions

In general, traversing through a class object requires that you first become familiar with the internal class structure and member access functions for that object. In this section we consider how you can access this information in the source files, and apply it. The example provided here involves a Biostruc type which is implemented by class CBiostruc, and its base (parent) class, CBiostruc_Base.

The first question is: how do I locate the class definitions implementing the object to be traversed? There are now two source browsers which you can use. To obtain a synopsis of the class, you can search the index or the class hierarchy of the *Doc++* browser and follow a link to the class. For example, a synopsis of the CBiostruc class is readily available. From this page, you can also access the relevant source files archived by the*LXR* browser, by following the Locate CBiostruc link. Alternatively, you may want to access the *LXR* engine directly by using the Identifier search tool.

Because we wish to determine which headers to include, the synopsis displayed by the Identifier search tool is most useful. There we find a single header file, Biostruc.hpp, listed as defining the class. Accordingly, this is the header file we must include. The CBiostruc class inherits from the CBiostruc_Base class however, and we will need to consult that file as well to understand the internal structure of the CBiostruc class. Following a link to the parent class from the class hierarchy browser, we find the definition of the CBiostruc_Base class.

This is where we must look for the definitions and access functions we will be using. However, it is the derived user class (CBiostruc) whose header should be included in your source files, and which should be instantiated by your local program variable. For a more general discussion of the relationship between the base parent objects and their derived user classes, see Working with the serializable object classes.

Accessing and Referencing Data Members

Omitting some of the low-level details of the base class, we find the CBiostruc_Base class has essentially the following structure:

```
class CBiostruc_Base : public CObject
{
public:
 // type definitions
 typedef list< CRef<CBiostruc_id> > TId;
 typedef list< CRef<CBiostruc_descr> > TDescr;
 typedef list< CRef<CBiostruc_feature_set> > TFeatures;
 typedef list< CRef<CBiostruc_model> > TModel;
 typedef CBiostruc_graph TChemical_graph;
 // Get() members
 const TId& GetId(void) const;
 const TDescr& GetDescr(void) const;
 const TChemical_graph& GetChemical_graph(void) const;
 const TFeatures& GetFeatures(void) const;
 const TModel& GetModel(void) const;
 // Set() members
 TId& SetId(void);
 TDescr& SetDescr(void);
 TChemical_graph& SetChemical_graph(void);
 TFeatures& SetFeatures(void);
 TModel& SetModel(void);
private:
 TId m_Id;
 TDescr m_Descr;
 TChemical_graph m_Chemical_graph;
 TFeatures m_Features;
 TModel m_Model;
};
```

With the exception of the structure's chemical graph, each of the class's private data members is actually a list of references (pointers), as specified by the type definitions. For example, TId is a list of CRef objects, where each CRef object points to a CBiostruc_id. The CRef class is a type of smart pointer used to hold a pointer to a reference-counted object. The dereferencing operator, when applied to a (dereferenced) iterator pointing to an element of CBiostruc::TId, e.g. **CRef_i, will return a CBiostruc_id. Thus, the call to GetId() returns a list which must then be iterated over and dereferenced to get the individual CBiostruc_id objects. In contrast, the function GetChemicalGraph() returns the object directly, as it does not involve a list or a CRef.

NOTE: It is strongly recommended that you use type names defined in the generated classes (e.g. TId, TDescr) rather than generic container names (list< CRef<CBiostruc_id> > etc.). The real container class may change occasionally and you will have to modify the code using generic container types every time it happens. When iterating over a container it's recommended to use ITERATE and NON_CONST_ITERATE macros.

The GetXxx() and SetXxx() member functions define the user interface to the class, providing methods to access and modify ("mutate") private data. In addition, most classes, including

CBiostruc, have IsSetXxx() and ResetXxx() methods to validate and clear the data members, respectively.

Traversing a Biostruc

The program traverseBS.cpp (see Code Sample 4) demonstrates how one might load a serial data file and iterate over the components of the resulting object. This example reads from a text ASN.1 Biostruc file and stores the information into a CBiostruc object in memory. The overloaded Visit() function is then used to recursively examine the object CBiostruc bs and its components.

Visit(bs) simply calls Visit() on each of the CBiostruc data members, which are accessed using bs.GetXxx(). The information needed to write each of these functions - the data member types and member function signatures - is contained in the respective header files. For example, looking at Biostruc_.hpp, we learn that the structure's descriptor list can be accessed using GetDescr(), and that the type returned is a list of pointers to descriptors:

```
typedef list< CRef<CBiostruc_descr> > TDescr;
const TDescr& GetDescr(void) const;
```

Consulting the base class for CBiostruc_desc in turn, we learn that this class has a choice state defining the type of value stored there as well as the method that should be used to access that value. This leads to an implementation of Visit(CBiostruc::TDescr DescrList) that uses an iterator over its list argument and a switch statement over the current descriptor's choice state.

Iterating Over Containers

Most of the Visit() functions implemented here rely on standard STL iterators to walk through a list of objects. The general syntax for using an iterator is:

```
ContainerType ContainerName;
ITERATE(ContainerType, it, ContainerName) {
 ObjectType ObjectName = *it;
 // ...
}
```

Dereferencing the iterator is required, as the iterator behaves like a pointer that traverses consecutive elements of the container. For example, to iterate over the list of descriptors in the *Biostruc*, we use a container of type CBiostruc::TDescr, and the constant version of the ITERATE macro to ensure that the data is not mutated in the body of the loop. Because the descriptor list contains pointers (CRefs) to objects, we will actually need to dereference **twice** to get to the objects themselves.

```
ITERATE(CBiostruc::TDescr, it, descList) {
 const CBiostruc_descr& thisDescr = **it;
 // ...
}
```

In traversing the descriptor list in this example, we handled each type of descriptor with an explicit case statement. In fact, however, we really only visit those descriptors whose types have string representations: TName, TPdb_comment, and TOther_comment. The other two descriptor types, THistory and TAttribute, are objects that are "visited" recursively, but the associated visit functions are not actually implemented (see Code Sample 5, traverseBS.hpp).

The NCBI C++ Toolkit provides a rich and powerful set of iterators for various application needs. An alternative to using the above switch statement to visit elements of the descriptor list would have been to use an NCBI CStdTypeIterator that only visits strings. For example, we could implement the Visit function on a CBiostruc::TDescr as follows:

```
void Visit(const CBiostruc::TDescr& descList)
{
 ITERATE(CBiostruc::TDescr, it1, descList) {
 for (CStdTypeConstIterator<string> it2(ConstBegin(**it1)); it2; ++it2) {
 cout << *it2 << endl;
 }
 }
}
```

In this example, the iterator will skip over all but the string data members.

The CStdTypeIterator is one of several iterators which makes use of an object's type information to implement the desired functionality. We began this section by positing that the traversal of an object requires an a priori knowledge of that object's internal structure. This is not strictly true however, if type information for the object is also available. An object's type information specifies the class layout, inheritance relations, data member names, and various other attributes such as size, which are independent of specific instances. All of the C++ type iterators described in The NCBI C++ Toolkit Iterators section utilize type information, which is the topic of a previous section: Runtime Object Type Information.

SOAP support

The NCBI C++ Toolkit SOAP server and client provide a limited level of support of SOAP 1.1 over HTTP, and use the document binding style with a literal schema definition. Document/literal is the style that most Web services platforms were focusing on when this feature was introduced. Parsing of WSDL (Web services description language) specification and automatic C++ code generation are not supported. Still, since the WSDL message types section uses XML schema, and since the application is capable of parsing Schema, the majority of the C++ code generation can be done automatically.

SOAP message

The core section of the SOAP specification is the messaging framework. The client sends a request and receives a response in the form of a SOAP message. A SOAP message is a one-way transmission between SOAP nodes: from a SOAP sender to a SOAP receiver. The root element of a SOAP message is the Envelope. The Envelope contains an optional Header element followed by a mandatory Body element. The Body element represents the message payload - it is a generic container that can contain any number of elements from any namespace.

In the Toolkit, the CSoapMessage class defines Header and Body containers. Serializable objects (derived from the CSerialObject class) can be added into these containers using AddObject() method. Such a message object can then be sent to a message receiver. The response will also come in the form of an object derived from CSoapMessage. At this time, it is possible to investigate its contents using GetContent() method; or ask directly for an object of a specific type using the SOAP_GetKnownObject() template function.

SOAP client (CSoapHttpClient)

The SOAP client is the initial SOAP sender - a node that originates a SOAP message. Knowing the SOAP receiver's URL, it sends a SOAP request and receives a response using the Invoke () method.

Internally, data objects in the Toolkit SOAP library are serialized and de-serialized using serializable objects streams. Since each serial data object also provides access to its type information, writing such objects is a straightforward operation. Reading the response is not that transparent. Before actually parsing incoming data, the SOAP processor should decide which object type information to use. Hence, a client application should tell the SOAP processor what types of data objects it might encounter in the incoming data. If the processor recognizes the object's type, it will parse the incoming data and store it as an instance of the recognized type. Otherwise, the processor will parse the data into an instance of the CAnyContentObject class.

So, a SOAP client must:

- Define the server's URL.
- Register the object types that might be present in the incoming data (using the RegisterObjectType() method).

The CSoapHttpClient class also has methods for getting and setting the server URL and the default namespace.

SOAP server (CSoapServerApplication)

The SOAP server receives SOAP mesages from a client and processes the contents of the SOAP Body and SOAP Header.

The processing of incoming requests is done with the help of "message listeners" - the server methods which analyze requests (in the form of objects derived from CSoapMessage) and create responses. It is possible to have more than one listener for each message. When such a listener returns TRUE, the SOAP server base class object passes the request to the next listener, if it exists, and so on.

The server can return a WSDL specification if the specification file name is passed to the server's constructor and the file is located with the server.

Sample SOAP server and client

The Toolkit contains a simple example of SOAP client and server in its src/sample/app/soap folder.

The sample SOAP server supports the following operations:

GetDescription() - server receives an empty object of type Description, and it sends back a single string;

GetVersion() - server receives a string, and it sends back two integer numbers and a string;

DoMath() - server receives a list of Operand objects (two integers and an enumerated value), and it sends back a list of integers

The starting point is the WSDL specification - src\sample\app\soap\server \soap_server_sample.wsdl

Both client and server use data objects whose types are described in the message types section of WSDL specification. So, we extract the XML schema part of the specification into a separate file, and create a static library - soap_dataobj. All code in this library is generated automatically by .

Sample server

Server is a CGI application. In its constructor we define the name of WSDL specification file and the default namespace for the data objects. Since server's ability to return a WSDL specification upon request from a client is optional, it is possible to give an empty file name here. Once the name is not empty, the WSDL file should be deployed alongside the server.

During initialization server should register incoming object types and message listeners:

// Register incoming object types, so the SOAP message parser can

// recognize these objects in incoming data and parse them correctly.

RegisterObjectType(CVersion::GetTypeInfo);

RegisterObjectType(CMath::GetTypeInfo);

// Register SOAP message processors.

// It is possible to set more than one listeners for a particular message;

// such listeners will be called in the order of registration.

AddMessageListener((TWebMethod)(&CSampleSoapServerApplication::GetDescription), "Description"); AddMessageListener((TWebMethod)(&CSampleSoapServerApplication::GetDescription2), "Description");

AddMessageListener((TWebMethod)(&CSampleSoapServerApplication::GetVersion), "Version");

AddMessageListener((TWebMethod)(&CSampleSoapServerApplication::DoMath), "Math");

Note that while it is possible to register the Description type, it does not make much sense: the object has no content, so there is no difference whether it will be parsed correctly or not.

Message listeners are user-defined functions that process incoming messages. They analyze the content of SOAP message request and populate the response object.

Sample client

Unlike SOAP server, SOAP client object has nothing to do with CCgiApplication class. It is "just" an object. As such, it can be created and destroyed when appropriate. Sample SOAP client constructor defines the server URL and the default namespace for the data objects. Its constructor is the proper place to register incoming object types:

// Register incoming object types, so the SOAP message parser can

// recognize these objects in incoming data and parse them correctly.

RegisterObjectType(CDescriptionText::GetTypeInfo);

RegisterObjectType(CVersionResponse::GetTypeInfo);

RegisterObjectType(CMathResponse::GetTypeInfo);

Other methods encapsulate operations supported by the SOAP server, which the client talks to. Common schema is to create two SOAP message object - request and response, populate request object, call Invoke() method of the base class, and extract the meaningful data from the response.

Test Cases [src/serial/test]

Available Serializable Classes (as per NCBI ASN.1 Specifications) [Library xobjects: include | src]

The ASN.1 data objects are automatically built from their corresponding specifications in the NCBI ASN.1 data model, using DATATOOL to generate all of the required source code. This set of serializable classes defines an interface to many important sequence and sequence-aware objects that users may directly employ, or extend with their own code. An Object Manager (see below) coordinates and simplifies the use of these ASN.1-derived objects.

Serializable Classes

- access [include | src]
- biblio [include | src]
- cdd [include | src]
- cn3d [include | src]
- docsum [include | src]
- entrez2 [include | src]
- featdef [include | src]
- general [include | src]
- id1 [include | src]
- medlars [include | src]
- medline [include | src]
- mim [include | src]
- mla [include | src]
- mmdb1 [include | src]
- mmdb2 [include | src]
- mmdb3 [include | src]
- ncbimime [include | src]
- objprt [include | src]
- proj [include | src]
- pub [include | src]
- pubmed [include | src]
- seq [include | src]
- seqalign [include | src]
- seqblock [include | src]

- seqcode [include | src]
- seqfeat [include | src]
- seqloc [include | src]
- seqres [include | src]
- seqset [include | src]
- submit [include | src]
- taxon1 [include | src]

A Test Application Using the Serializable ASN.1 Classes

- asn2asn [src]

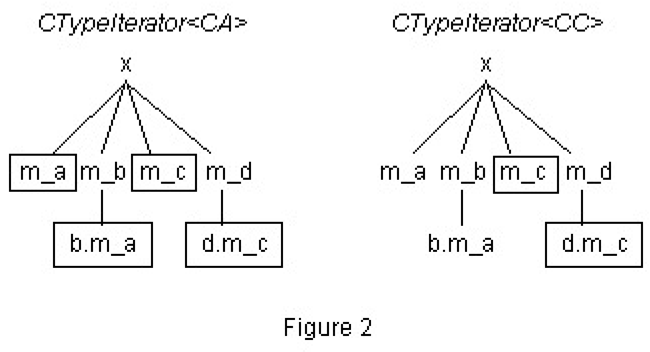


Figure 1. Traversal path of the CTypeIterator

Table 1. Network Service Client Generation Parameters

Name	Value
class (REQUIRED)	C++ class name to use.
service	Named service to connect to; if you do not define this, you will need to override x_Connect in the user class.
serialformat	Serialization format: normally AsnBinary, but AsnText and Xml are also legal.
request (REQUIRED)	ASN.1 type for requests; may include a module name, a field name (as with Entrez2), or both. Must be a CHOICE.
reply (REQUIRED)	ASN.1 type for replies, as above.
reply.choice_name	Reply choice appropriate for requests of type choice_name; defaults to choice_name as well, and determines the return type of AskChoice_name. May be set to special to suppress automatic method generation and let the user class handle the whole thing.

Table 2. BEGIN_NAMED_* Macro names and their usage

Macro name	**Used for**	**Arguments**
BEGIN_NAMED_CLASS_INFO	Non-abstract class object	ClassAlias, ClassName
BEGIN_NAMED_ABSTRACT_CLASS_INFO	Abstract class object	ClassAlias, ClassName
BEGIN_NAMED_DERIVED_CLASS_INFO	Derived subclass object	ClassAlias, ClassName, BaseClassName
BEGIN_NAMED_CHOICE_INFO	C++ class choice object	ClassAlias, ClassName
BEGIN_NAMED_ENUM_INF	Enum object	EnumAlias, EnumName, IsInteger
BEGIN_NAMED_ENUM_IN_INFO	internal Enum object	EnumAlias, CppContext, EnumName, IsInteger

Table 3. ADD_* Macros and their usage

Macro name	**Usage**	**Arguments**
ADD_NAMED_STD_MEMBER	Add a standard data member to a class	MemberAlias, MemberName
ADD_NAMED_CLASS_MEMBER	Add an internal class member to a class	MemberAlias, MemberName
ADD_NAMED_SUB_CLASS	Add a derived subclass to a class	SubClassAlias, SubClassName
ADD_NAMED_REF_MEMBER	Add a CRef data member to a class	MemberAlias, MemberName, RefClass
ADD_NAMED_ENUM_MEMBER	Add an enumerated data member to a class	MemberAlias, MemberName, EnumName
ADD_NAMED_ENUM_IN_MEMBER	Add an externally defined enumerated data member to a class	MemberAlias, MemberName, CppContext, EnumName
ADD_NAMED_MEMBER	Add a data member of the type specified by TypeMacro to a class	MemberAlias, MemberName, TypeMacro, TypeMacroArgs
ADD_NAMED_STD_CHOICE_VARIANT	Add a standard variant type to a C++ choice object	VariantAlias, VariantName
ADD_NAMED_REF_CHOICE_VARIANT	Add a CRef variant to a C++ choice object	VariantAlias, VariantName, RefClass
ADD_NAMED_ENUM_CHOICE_VARIANT	Add an enumeration variant to a C++ choice object	VariantAlias, VariantName, EnumName
ADD_NAMED_ENUM_IN_CHOICE_VARIANT	Add an enumeration variant to a C++ choice object	VariantAlias, VariantName, CppContext, EnumName
ADD_NAMED_CHOICE_VARIANT	Add a variant of the type specified by TypeMacro to a C++ choice object	VariantAlias, VariantName, TypeMacro, TypeMacroArgs
ADD_ENUM_VALUE	Add a named enumeration value to an enum	EnumValName, Value

Table 4. Type macros and their arguments

TypeMacro	TypeMacroArgs
CLASS	(ClassName)
STD	(C++ type)
StringStore	()
null	()
ENUM	(EnumType, EnumName)
POINTER	(Type,Args)
STL_multiset	(Type,Args)
STL_set	(Type,Args)
STL_multimap	(KeyType,KeyArgs,ValueType,ValueArgs)
STL_map	(KeyType,KeyArgs,ValueType,ValueArgs)
STL_list	(Type,Args)
STL_list_set	(Type,Args)
STL_vector	(Type,Args)
STL_CHAR_vector	(C++ Char type)
STL_auto_ptr	(Type,Args)
CHOICE	(Type,Args)

Code Sample 1. xml2asn.cpp

```
// File name: xml2asn.cpp
// Description: Reads an XML Biostruc file into memory
// and saves it in ASN.1 text and binary formats.

#include <corelib/ncbistd.hpp>
#include <corelib/ncbiapp.hpp>
#include <serial/serial.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <objects/mmdb1/Biostruc.hpp>

USING_NCBI_SCOPE;

class CTestAsn : public CNcbiApplication {
public:
 virtual int Run ();
};

using namespace objects;

int CTestAsn::Run() {
 auto_ptr<CObjectIStream>
 xml_in(CObjectIStream::Open("1001.xml", eSerial_Xml));
 auto_ptr<CObjectOStream>
 txt_out(CObjectOStream::Open("1001.asntxt", eSerial_AsnText));
 auto_ptr<CObjectOStream>
 bin_out(CObjectOStream::Open("1001.asnbin", eSerial_AsnBinary));
 CBiostruc bs;
 *xml_in >> bs;
 *txt_out << bs;
 *bin_out << bs;
 return 0;
}

int main(int argc, const char* argv[])
{
 CNcbiOfstream diag("asntrans.log");
 SetDiagStream(&diag);
 CTestAsn theTestApp;
 return theTestApp.AppMain(argc, argv);
}
```

Code Sample 2. ctypeiter.cpp

```
// File name: ctypeiter.cpp
// Description: Demonstrate using a CTypeIterator
// Notes: build with xncbi and xser libraries

#include "ctypeiter.hpp"
#include <serial/serial.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/iterator.hpp>
#include <serial/serialimpl.hpp>

// type information for class CPerson
BEGIN_CLASS_INFO(CPerson){
 ADD_STD_MEMBER(m_Name);
 ADD_STD_MEMBER(m_Addr);
 ADD_MEMBER(m_NextDoor, POINTER, (CLASS, (CPerson)))->SetOptional();
}END_CLASS_INFO

// type information for class CDistrict
BEGIN_CLASS_INFO(CDistrict){
 ADD_STD_MEMBER(m_Number);
 ADD_MEMBER(m_Blocks, STL_list, (CLASS, (CPerson)));
}END_CLASS_INFO

// main and other functions
USING_NCBI_SCOPE;

static void FullerBrushMan (const CPerson& p) {
 cout << "knock-knock! is " << p.m_Name << " home?" << endl;
}

int main(int argc, char** argv)
{
 // Instantiate a few CPerson objects
 CPerson neighborhood("Moe", "1 Main St",
 new CPerson("Larry", "2 Main St",
 new CPerson("Curly", "3 Main St", 0)));
 CPerson another("Harpo", "2 River Rd",
 new CPerson("Chico", "4 River Rd",
 new CPerson("Groucho", "6 River Rd", 0)));

 // Create a CDistrict and install some CPerson objects
 CDistrict district1(1);
 district1.AddBlock(neighborhood);
 district1.AddBlock(another);
 // Send the FullerBrushMan to all CPersons in district1
 for (CTypeConstIterator<CPerson> house = ConstBegin(district1);
 house; ++house ) {
 FullerBrushMan(*house);
 }
 // Iterate over all strings for the CPersons in district1
 list<CPerson> blocks(district1.GetBlocks());
 ITERATE(list<CPerson>, b, blocks) {
 for (CStdTypeIterator<string> it(Begin(*b)); it; ++it) {
 cout << *it << ' ';
 }
 cout << endl;
 }
 return 0;
}
```

Code Sample 3. ctypeiter.hpp

```
// File name: ctypeiter.hpp

#ifndef CTYPEITER_HPP
#define CTYPEITER_HPP

#include <corelib/ncbistd.hpp>
#include <corelib/ncbiobj.hpp>
#include <serial/typeinfo.hpp>
#include <string>
#include <list>

USING_NCBI_SCOPE;

class CPerson
{
public:
 CPerson(void)
 : m_Name(0), m_Addr(0), m_NextDoor(0) {}
 CPerson(string n, string s, CPerson* p)
 : m_Name(n), m_Addr(s), m_NextDoor(p) {}
 virtual ~CPerson(void) {}
 static const CTypeInfo* GetTypeInfo(void);
private:
 string m_Name, m_Addr;
 CPerson *m_NextDoor;
};

class CDistrict
{
public:
 CDistrict(void)
 : m_Number(0) {}
 CDistrict(int n) : m_Number(n) {}
 virtual ~CDistrict(void) {}
 static const CTypeInfo* GetTypeInfo(void);
 int m_Number;
 void AddBlock (const CPerson& p) { m_Blocks.push_back(p); }
 list<CPerson>& GetBlocks() { return m_Blocks; }
private:
 list<CPerson> m_Blocks;
};
#endif /* CTYPEITER_HPP */
```

Code Sample 4. traverseBS.cpp

```
// File name: traverseBS.cpp
// Description: Reads an ASN.1 Biostruc text file into memory
// and visits its components

#include <serial/serial.hpp>
#include <serial/iterator.hpp>
#include <serial/objistr.hpp>
#include <serial/serial.hpp>
#include <objects/general/Dbtag.hpp>
#include <objects/general/Object_id.hpp>
#include <objects/seq/Numbering.hpp>
#include <objects/seq/Pubdesc.hpp>
#include <objects/seq/Heterogen.hpp>
#include <objects/mmdb1/Biostruc.hpp>
#include <objects/mmdb1/Biostruc_id.hpp>
#include <objects/mmdb1/Biostruc_history.hpp>
#include <objects/mmdb1/Mmdb_id.hpp>
#include <objects/mmdb1/Biostruc_descr.hpp>
#include <objects/mmdb1/Biomol_descr.hpp>
#include <objects/mmdb1/Molecule_graph.hpp>
#include <objects/mmdb1/Inter_residue_bond.hpp>
#include <objects/mmdb1/Residue_graph.hpp>
#include <objects/mmdb3/Biostruc_feature_set.hpp>
```

```
#include <objects/mmdb2/Biostruc_model.hpp>
#include <objects/pub/Pub.hpp>
#include <corelib/ncbistre.hpp>

#include "traverseBS.hpp"

USING_NCBI_SCOPE;
using namespace objects;

int CTestAsn::Run()
{
 // initialize ASN input stream
 auto_ptr<CObjectIStream>
 inObject(CObjectIStream::Open("1001.val", eSerial_AsnBinary));
 // initialize, read into, and traverse CBiostruc object
 CBiostruc bs;
 *inObject >> bs;
 Visit (bs);
 return 0;
}

/*****************************************************************
*
* The overloaded free "visit" functions are used to explore the
* Biostruc and all its component members - most of which are also
* class objects. Each class has a public interface that provides
* access to its private data via "get" functions.
*
******************************************************************/
void Visit (const CBiostruc& bs)
{
 cout << "Biostruc:\n" << endl;
 Visit (bs.GetId());
 Visit (bs.GetDescr());
 Visit (bs.GetChemical_graph());
 Visit (bs.GetFeatures());
 Visit (bs.GetModel());
}

/************************************************************************
*
* TId is a type defined in the CBiostruc class as a list of CBiostruc_id,
* where each id has a choice state and a value. Depending on the choice
* state, a different get() function is used.
*
*************************************************************************/
void Visit (const CBiostruc::TId& idList)
{
 cout << "\n Visiting Ids of Biostruc:\n";

 for (CBiostruc::TId::const_iterator i = idList.begin();
 i != idList.end(); ++i) {

 // dereference the iterator to get to the id object
 const CBiostruc_id& thisId = **i;
 CBiostruc_id::E_Choice choice = thisId.Which();
 cout << "choice = " << choice;

 // select id's get member function depending on choice
 switch (choice) {
 case CBiostruc_id::e_Mmdb_id:
 cout << " mmdbId: " << thisId.GetMmdb_id().Get() << endl;
 break;
 case CBiostruc_id::e_Local_id:
 cout << " Local Id: " << thisId.GetLocal_id().GetId() << endl;
 break;
 case CBiostruc_id::e_Other_database:
 cout << " Other DB Id: "
 << thisId.GetOther_database().GetDb() << endl;
 break;
 default:
 cout << "Choice not set or unrecognized" << endl;
 }
 }
}

/***************************************************************************
*
* TDescr is also a type defined in the Biostruc class as a list of
* CBiostruc_descr, where each descriptor has a choice state and a value.
*
***************************************************************************/
void Visit (const CBiostruc::TDescr& descList)
{
 cout << "\n Visiting Descriptors of Biostruc:\n";

 for (CBiostruc::TDescr::const_iterator i = descList.begin();
 i != descList.end(); ++i) {

 // dereference the iterator to get the descriptor
 const CBiostruc_descr& thisDescr = **i;
 CBiostruc_descr::E_Choice choice = thisDescr.Which();
 cout << "choice = " << choice;

 // select the get function depending on choice
 switch (choice) {
 case CBiostruc_descr::e_Name:
 cout << " Name: " << thisDescr.GetName() << endl;
 break;
 case CBiostruc_descr::e_Pdb_comment:
 cout << " Pdb comment: " << thisDescr.GetPdb_comment() << endl;
 break;
 case CBiostruc_descr::e_Other_comment:
 cout << " Other comment: " << thisDescr.GetOther_comment() << endl;
 break;
 case CBiostruc_descr::e_History:
 cout << " History: " << endl;
 Visit (thisDescr.GetHistory());
 break;
 case CBiostruc_descr::e_Attribution:
 cout << " Attribute: " << endl;
 Visit (thisDescr.GetAttribution());
 break;
 default:
 cout << "Choice not set or unrecognized" << endl;
 }
 }
 VisitWithIterator (descList);
}

/
***************************************************************************
**
*
* An alternate way to visit the descriptor nodes using a CStdTypeIterator
*
***************************************************************************
**/
void VisitWithIterator (const CBiostruc::TDescr& descList)
{
 cout << "\n Revisiting descriptor list with string iterator...:\n";

 for (CBiostruc::TDescr::const_iterator i1 = descList.begin();
 i1 != descList.end(); ++i1) {

 const CBiostruc_descr& thisDescr = **i1;

 for (CStdTypeConstIterator<NCBI_NS_STD::string>
 i = ConstBegin(thisDescr); i; ++i) {
 cout << "next descriptor" << *i << endl;
 }
 }
}

/
***************************************************************************
**
*
* Chemical graphs contain lists of descriptors, molecule_graphs, bonds,
and
* residue graphs. Here we just visit some of the descriptors.
*
***************************************************************************
```

```
**/
void Visit (const CBiostruc::TChemical_graph& G)
{
 cout << "\n\n Visiting Chemical Graph of Biostruc\n";

 const CBiostruc_graph::TDescr& descList = G.GetDescr();
 for (CBiostruc_graph::TDescr::const_iterator i = descList.begin();
 i != descList.end(); ++i) {

 // dereference the iterator to get the descriptor
 const CBiomol_descr& thisDescr = **i;
 CBiomol_descr::E_Choice choice = thisDescr.Which();
 cout << "choice = " << choice;


 // select the get function depending on choice
 switch (choice) {
 case CBiomol_descr::e_Name:
 cout << " Name: " << thisDescr.GetName() << endl;
 break;
 case CBiomol_descr::e_Pdb_class:
 cout << " Pdb class: " << thisDescr.GetPdb_class() << endl;
 break;
 case CBiomol_descr::e_Pdb_source:
 cout << " Pdb Source: " << thisDescr.GetPdb_source() << endl;
 break;
 case CBiomol_descr::e_Pdb_comment:
 cout << " Pdb comment: " << thisDescr.GetPdb_comment() << endl;
 break;
 case CBiomol_descr::e_Other_comment:
 cout << " Other comment: " << thisDescr.GetOther_comment() << endl;
 break;
 case CBiomol_descr::e_Organism: // skipped
 case CBiomol_descr::e_Attribution:
 break;
 case CBiomol_descr::e_Assembly_type:
 cout << " Assembly Type: " << thisDescr.GetAssembly_type() << endl;
 break;
 case CBiomol_descr::e_Molecule_type:
 cout << " Molecule Type: " << thisDescr.GetMolecule_type() << endl;
 break;
 default:
 cout << "Choice not set or unrecognized" << endl;
 }
 }
}

void Visit (const CBiostruc::TFeatures&)
{
 cout << "\n\n Visiting Features of Biostruc\n";
}
```

```
void Visit (const CBiostruc::TModel&)
{
 cout << "\n\n Visiting Models of Biostruc\n";
}

int main(int argc, const char* argv[])
{
 // initialize diagnostic stream
 CNcbiOfstream diag("traverseBS.log");
 SetDiagStream(&diag);

 CTestAsn theTestApp;
 return theTestApp.AppMain(argc, argv);
}
```

Code Sample 5. traverseBS.hpp

```
// File name traverseBS.hpp

#ifndef NCBI_TRAVERSEBS__HPP
#define NCBI_TRAVERSEBS__HPP

#include <corelib/ncbistd.hpp>
#include <corelib/ncbiapp.hpp>

USING_NCBI_SCOPE;
using namespace objects;

// class CTestAsn
class CTestAsn : public CNcbiApplication {
public:
 virtual int Run ();
};

void Visit(const CBiostruc&);
void Visit(const CBiostruc::TId&);
void Visit(const CBiostruc::TDescr&);
void Visit(const CBiostruc::TChemical_graph&);
void Visit(const CBiostruc::TFeatures&);
void Visit(const CBiostruc::TModel&);
void Visit(const CBiostruc_history&) {
 cout << "visiting history" << endl;
};

// Not implemented
void Visit(const CBiostruc_descr::TAttribution&) {};
void VisitWithIterator (const CBiostruc::TDescr& descList);
```

```
#endif /* NCBI_TRAVERSEBS__HPP */
```

14: Biological Sequence Data Model

Last Update: February 8, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter describes the NCBI Biological Sequence Data Model, with emphasis on the ASN.1 files and C++ API. ASN.1 type names and the corresponding C++ class or data member names are used almost interchangeably throughout the chapter. Another good source of information about the NCBI data model is:

Bioinformatics
A Practical Guide to the Analysis of Genes and Proteins
Second Edition (2001)
Edited by Andreas D. Baxevanis, B. F. Francis Ouellette
ISBN 0-471-38391-0

Chapter 2 - The NCBI Data Model

Chapter Outline

The following is an outline of the topics presented in this chapter:

Data Model

The Data Model section outlines the NCBI model for biotechnology information, which is centered on the concept of a biological sequence as a simple, linear coordinate system.

Introduction

The NCBI sequence databases and software tools are designed around a particular model of biological sequence data. It is designed to provide a few unifying concepts which cross a wide range of domains, providing a path between the domains. Specialized objects are defined which are appropriate within a domain. In the following sections we will present the unifying ideas, then examine each area of the model in more detail.

Since we expect that computer technologies will continue to develop at a rapid rate, NCBI has made considerable investment of time and energy to ensure that our data and software tools are not too tightly bound to any particular computer platform or database technology. However, we also wish to embrace the intellectual rigor imposed by describing our data within a formal system and in a machine readable and checkable way. For this reason we have chosen to describe our data in Abstract Syntax Notation 1 (ASN.1; ISO 8824, 8825). Enough explanation will be given here to allow the reader to examine the data definitions. A much fuller description of ASN.1 and the NCBI software tools which use it appears in later chapters.

The data specification sections are arranged by ASN.1 module with detailed discussions of data objects defined in each and the software functions available to operate on those objects. Each ASN.1 defined object has a matching C++ language class. Each C++ class has at a minimum, functions to: create it, write it to an ASN.1 stream, read it from an ASN.1 stream, and destroy it. Many objects have additional functions. Some of these are described in the section on the module and some with more extensive interfaces are described in additional sections. Each module section begins with a description of the elements, followed by the full ASN.1 definition of the module. The C++ API is referenced with links.

This section provides an overview of all modules. Selected ASN.1 definitions are inserted into the body of the text as necessary. They are also described in the section on the appropriate module.

There are two major areas for which data objects have been defined. One is bibliographic data. It is clear that this class of information is central to all scientific fields within and outside of molecular biology so we expect these definitions to be widely useful. We have followed the American National Standard for Bibliographic References (ANSI Z39.29-1977) and consulted with the US Patent Office and professional librarians to ensure complete and accurate representation of citation information. Unlike biological data, this data is relatively well understood, so we hope that the bibliographic specification can be quite complete and stable. Despite its importance, the bibliographic specification will not be discussed further here, since it does not present ideas which may be novel to the reader.

The other major area of the specification is biological sequence data and its associated information. Here the data model attempts to achieve a number of goals. Biomedical information is a vast interconnected web of data which crosses many domains of discourse with very different ways of viewing the world. Biological science is very much like the parable of the blind men and elephant. To some of the blind men the elephant feels like a column, to some like a snake, to others like a wall. The excitement of modern biological research is that we all agree that, at least at some level, we are all exploring aspects of the same thing. But it is early enough in the development of the science that we cannot agree on what that thing is.

The power of molecular biology is that DNA and protein sequence data cut across most fields of biology from evolution to development, from enzymology to agriculture, from statistical mechanics to medicine. Sequence data can be viewed as a simple, relatively well defined armature on which data from various disciplines can be hung. By associating diverse data with the sequence, connections can be made between fields of research with no other common ground, and often with little or no idea of what the other field is doing.

This data model establishes a biological sequence as a simple integer coordinate system with which diverse data can be associated. It is reasonable to hope that such a simple core can be very stable and compatible with a very wide range of data. Additional information closely linked to the coordinate system, such as the sequence of amino acids or bases, or genes on a genetic map are layered onto it. With stable identifiers for specific coordinate systems, a greater diversity of information about the coordinate system can be specifically attached to it in a very flexible yet rigorous way. The essential differences between different biological forms are preserved, yet they can viewed as aspects of the same thing around the core, and thus move us toward our goal of understanding the totality.

Biological Sequences

A Bioseq is a single continuous biological sequence. It can be nucleic acid or protein. It can be fully instantiated (i.e. we have data for every residue) or only partially instantiated (e.g. we know a fragment is 10 kilobases long, but we only have sequence data over 1 kilobase). A Bioseq is defined in ASN.1 as follows:

```
Bioseq ::= SEQUENCE {
 id SET OF Seq-id , -- equivalent identifiers
 descr Seq-descr OPTIONAL , -- descriptors
 inst Seq-inst , -- the sequence data
 annot SET OF Seq-annot OPTIONAL }
```

In ASN.1 a named datatype begins with a capital letter (e.g. Bioseq). The symbol "::=" means "is defined as". A primitive type is all capitals (e.g. SEQUENCE). A field within a named datatype begins with a lower case letter (e.g. descr). A structured datatype is bounded by curly brackets ({}). We can now read the definition above: a Bioseq is defined as a SEQUENCE (i.e. a structure where the elements must come in order; the mathematical notion of a sequence, not the biological one). The first element of Bioseq is called "id" and is a SET OF (i.e. an unordered collection of elements of the same type) a named datatype called "Seq-id". Seq-id would have its own definition elsewhere. The second element is called "descr" and is a named type called "Seq-descr", which is optional. In this text, when we wish to refer to the id element of the named type Bioseq, we will use the notation "Bioseq.id".

A Bioseq has two optional elements, which both have descriptive information about the sequence. Seq-descr is a collection of types of information about the context of the sequence. It may set biological context (e.g. define the organism sequenced), or bibliographic context (e.g. the paper it was published in), among other things. Seq-annot is information that is explicitly tied to locations on the sequence. This could be feature tables, alignments, or graphs, at the present time. A Bioseq can have more than one feature table, perhaps coming from different sources, or a feature table and a graph, etc.

A Bioseq is only required to have two elements, id and inst. Bioseq.id is a set of one or more identifiers for this Bioseq. An identifier is a key which allows us to retrieve this object from a database or identify it uniquely. It is not a name, which is a human compatible description, but not necessarily a unique identifier. The name "Jane Doe" does not uniquely identify a person in the United States, while the identifier, social security number, does. Each Seq-id is a CHOICE of one of a number of identifier types from different databases, which may have different structures. All Bioseqs *must* have at least one identifier.

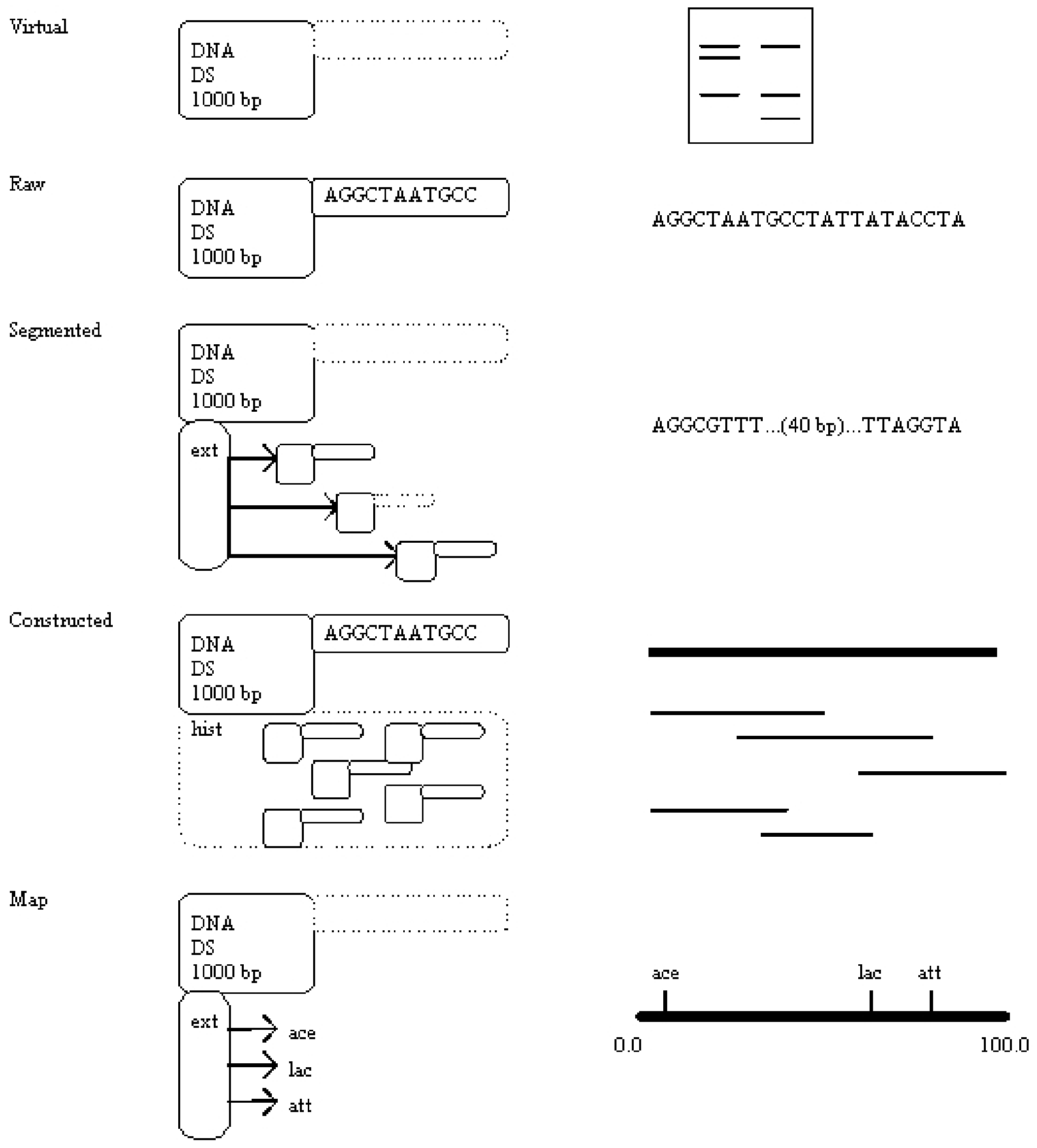
Classes of Biological Sequences

The other required element of a Bioseq is a Seq-inst. This element instantiates the sequence itself. It represents things like: Is it DNA, RNA, or protein? Is it circular or linear? Is it double-stranded or single-stranded? How long is it?

```
Seq-inst ::= SEQUENCE { -- the sequence data itself
 repr ENUMERATED { -- representation class
 not-set (0) , -- empty
 virtual (1) , -- no seq data
 raw (2) , -- continuous sequence
 seg (3) , -- segmented sequence
 const (4) , -- constructed sequence
 ref (5) , -- reference to another sequence
 consen (6) , -- consensus sequence or pattern
 map (7) , -- ordered map of any kind
 delta (8) , -- sequence made by changes (delta) to others
 other (255) } ,
 mol ENUMERATED { -- molecule class in living organism
 not-set (0) , -- > cdna = rna
 dna (1) ,
 rna (2) ,
 aa (3) ,
 na (4) , -- just a nucleic acid
 other (255) } ,
 length INTEGER OPTIONAL , -- length of sequence in residues
 fuzz Int-fuzz OPTIONAL , -- length uncertainty
 topology ENUMERATED { -- topology of molecule
 not-set (0) ,
 linear (1) ,
 circular (2) ,
 tandem (3) , -- some part of tandem repeat
 other (255) } DEFAULT linear ,
 strand ENUMERATED { -- strandedness in living organism
 not-set (0) ,
 ss (1) , -- single strand
 ds (2) , -- double strand
 mixed (3) ,
 other (255) } OPTIONAL , -- default ds for DNA, ss for RNA, pept
 seq-data Seq-data OPTIONAL , -- the sequence
 ext Seq-ext OPTIONAL , -- extensions for special types
 hist Seq-hist OPTIONAL } -- sequence history
```

Seq-inst is the parent class of a sequence representation class hierarchy. There are two major branches to the hierarchy. The molecule type branch is indicated by Seq-inst.mol. This could be a nucleic acid, or further sub classified as RNA or DNA. The nucleic acid may be circular, linear, or one repeat of a tandem repeat structure. It can be double, single, or of a mixed strandedness. It could also be a protein, in which case topology and strandedness are not relevant.

There is also a representation branch, which is independent of the molecule type branch. This class hierarchy involves the particular data structure used to represent the knowledge we have about the molecule, no matter which part of the molecule type branch it may be in. The repr element indicates the type of representation used. The aim of such a set of representation classes is to support the information to express different views of sequence based objects, from chromosomes to restriction fragments, from genetic maps to proteins, within a single overall model. The ability to do this confers profound advantages for software tools, data storage and retrieval, and traversal of related sequence and map data from different scientific domains.



A **virtual** representation is used to describe a sequence about which we may know things like it is DNA, it is double stranded, we may even know its length, but we do not have the actual sequence itself yet. Most fields of the Seq-inst are filled in, but Seq-inst.seq-data is empty. An example would be a band on a restriction map.

A **raw** representation is used for what we traditionally consider a sequence. We know it is DNA, it is double stranded, we know its length exactly, and we have the sequence data itself. In this case, Seq-inst.seq-data contains the sequence data.

A **segmented** representation is very analogous to a virtual representation. We posit that a continuous double stranded DNA sequence of a certain length exists, and pieces of it exist in other Bioseqs, but there is no data in Seq-inst.seq-data. Such a case would be when we have cloned and mapped a DNA fragment containing a large protein coding region, but have only actually sequenced the regions immediately around the exons. The sequence of each exon is an individual raw Bioseq in its own right. The regions between exons are virtual Bioseqs. The segmented Bioseq uses Seq-inst.ext to hold a SEQUENCE OF Seq-loc. That is, the extension is an ordered series of locations on *other* Bioseqs, in this case the raw and virtual Bioseqs representing the exons and introns. The segmented Bioseq contains data only by reference to other Bioseqs. In order to retrieve the base at the first position in the segmented Bioseq, one would go to the first Seq-loc in the extension, and return the appropriate base from the Bioseq it points to.

A **constructed** Bioseq is used to describe an assembly or merge of other Bioseqs. It is analogous to the raw representation. In fact, most raw Bioseqs were actually constructed from an assembly of gel readings. However, the constructed representation class is really meant for tracking higher level merging, such as when an expert in a particular organism or gene region may construct a "typical" sequence from that region by merging available sequence data, often published by different groups, using domain knowledge to resolve discrepancies between reports or to select a typical allele. Seq-inst contains an optional Seq-hist object. Seq-hist contains a field called "assembly" which is a SET OF Seq-align, or sequence alignments. The alignments are used to record the history of how the various component Bioseqs used for the merge are related to the final product. A constructed sequence DOES contain sequence data in Seq-inst.seq-data, unlike a segmented sequence, because the component sequences may overlap, or expert knowledge may have been used to determine the "correct" residue at any position that is not captured in the original components. So Seq-hist.assembly is used to simply record the relationship of the merge to the old Bioseqs, but does NOT describe how to generate it from them.

A **map** is akin to a virtual Bioseq. For example, for a genetic map of E.coli, we might posit that the E.coli chromosome is about 5 million base pairs long, DNA, double stranded, circular, but we do not have the sequence data for it. However, we do know the positions of some genes on this putative sequence. In this case, the Seq-inst.ext is a SEQUENCE OF Seq-feat, that is, a feature table. For a genetic map, the feature table contains Gene-ref features. An ordered restriction map would have a feature table containing Rsite-ref features. The feature table is part of Seq-inst because, for a map, it is an essential part of instantiating the map Bioseq, not merely annotation on a known sequence. In a sense, for a map, the annotation IS part of the sequence. As an aside, note that we have given gene positions on the E.coli genetic map in base pairs, while the standard E.coli map is numbered from 0.0 to 100.0 map units. Numbering systems can be applied to a Bioseq as a descriptor or a feature. For E.coli, we would simply apply the 0.0 - 100.0 floating point numbering system to the map Bioseq. Gene positions can then be shown to the scientists in familiar map units, while the underlying software still treats positions as large integers, just the same as with any other Bioseq.

Coordinates on ANY class of Bioseq are ALWAYS integer offsets. So the first residue in any Bioseq is at position 0. The last residue of any Bioseq is in position (length - 1).

The consequence of this design is that one uses EXACTLY the same data object to describe the location of a gene on an unsequenced restriction fragment, a fully sequenced piece of DNA, a partially sequenced piece of DNA, a putative overview of a large genetic region, or a genetic or physical map. Software to display, manipulate, or compare gene locations can work without change on the full range of possible representations. Sequence and physical map data can be easily integrated into a single, dynamically assembled view by creating a segmented sequence which points alternatively to raw or constructed Bioseqs and parts of a map Bioseq. The relationship between a genetic and physical map is simply an alignment between two Bioseqs of representation class map, no different than the alignment between two sequences of class raw generated by a database search program like BLAST or FASTA.

Locations on Biological Sequences

A Seq-loc is an object which defines a location on a Bioseq. The smooth class hierarchy for Seq-inst makes it possible to use the same Seq-loc to describe an interval on a genetic map as that used to describe an interval on a sequenced molecule.

Seq-loc is itself a class hierarchy. A valid Seq-loc can be an interval, a point, a whole sequence, a series of intervals, and so on.

```
Seq-loc ::= CHOICE {
 null NULL , -- not placed
 empty Seq-id , -- to NULL one Seq-id in a collection
 whole Seq-id , -- whole sequence
 int Seq-interval , -- from to
 packed-int Packed-seqint ,
 pnt Seq-point ,
 packed-pnt Packed-seqpnt ,
 mix Seq-loc-mix ,
 equiv Seq-loc-equiv , -- equivalent sets of locations
 bond Seq-bond ,
 feat Feat-id } -- indirect, through a Seq-feat
```

Seq-loc.null indicates a region of unknown length for which no data exists. Such a location may be used in a segmented sequence for the region between two sequenced fragments about which nothing, not even length, is known.

All other Seq-loc types, except Seq-loc.feat, contain a Seq-id. This means they are independent of context. This means that data objects describing information ABOUT Bioseqs can be created and exchanged independently from the Bioseq itself. This encourages the development and exchange of structured knowledge about sequence data from many directions and is an essential goal of the data model.

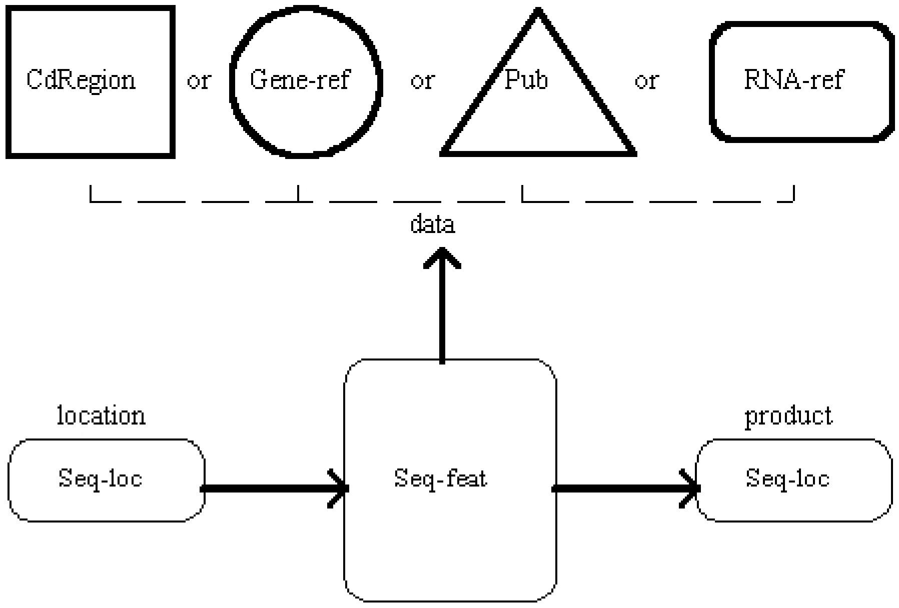
Associating Annotations With Locations On Biological Sequences

Seq-annot, or sequence annotation, is a collection of information ABOUT a sequence, tied to specific regions of Bioseqs through the use of Seq-loc's. A Bioseq can have many Seq-annot's associated with it. This allows knowledge from a variety of sources to be collected in a single place but still be attributed to the original sources. Currently there are three kinds of Seq-annot, feature tables, alignments, and graphs.

Feature Tables

A feature table is a collection of Seq-feat, or sequence features. A Seq-feat is designed to tie a Seq-loc together with a datablock, a block of specific data. Datablocks are defined objects themselves, many of which are objects used in their own right in some other context, such as publications (CPub_Base) or references to organisms (Org-ref) or genes (Gene-ref). Some datablocks, such as coding regions (CdRegion) make sense only in the context of a Seq-loc. However, since by design there is no intention that one datablock need to have anything in common with any other datablock, each can be tailored exactly to do a particular job. If a change or addition is required to one datablock, no others are affected. In those cases where a pre-existing object from another context is used as a datablock, any software that can use that object can now operate on the feature as well. For example, a piece of code to display a publication can operate on a publication from a bibliographic database or one used as a sequence feature with no change.

Since the Seq-feat data structure itself and the Seq-loc used to attach it to the sequence are common to all features, it is also possible to support a class of operations over all features without regard to the different types of datablocks attached to them. So a function to determine all features in a particular region of a Bioseq need not care what type of features they are.



A Seq-feat is bipolar in that it contains up to two Seq-loc's. Seq-feat.location indicates the "source" and is the location similar to the single location in common feature table implementations. Seq-feat.product is the "sink". A CdRegion feature would have its Seq-feat.location on the DNA and its Seq-feat.product on the protein sequence produced. Used this way it defines the process of translating a DNA sequence to a protein sequence. This establishes in an explicit way the important relationship between nucleic acid and protein sequence databases.

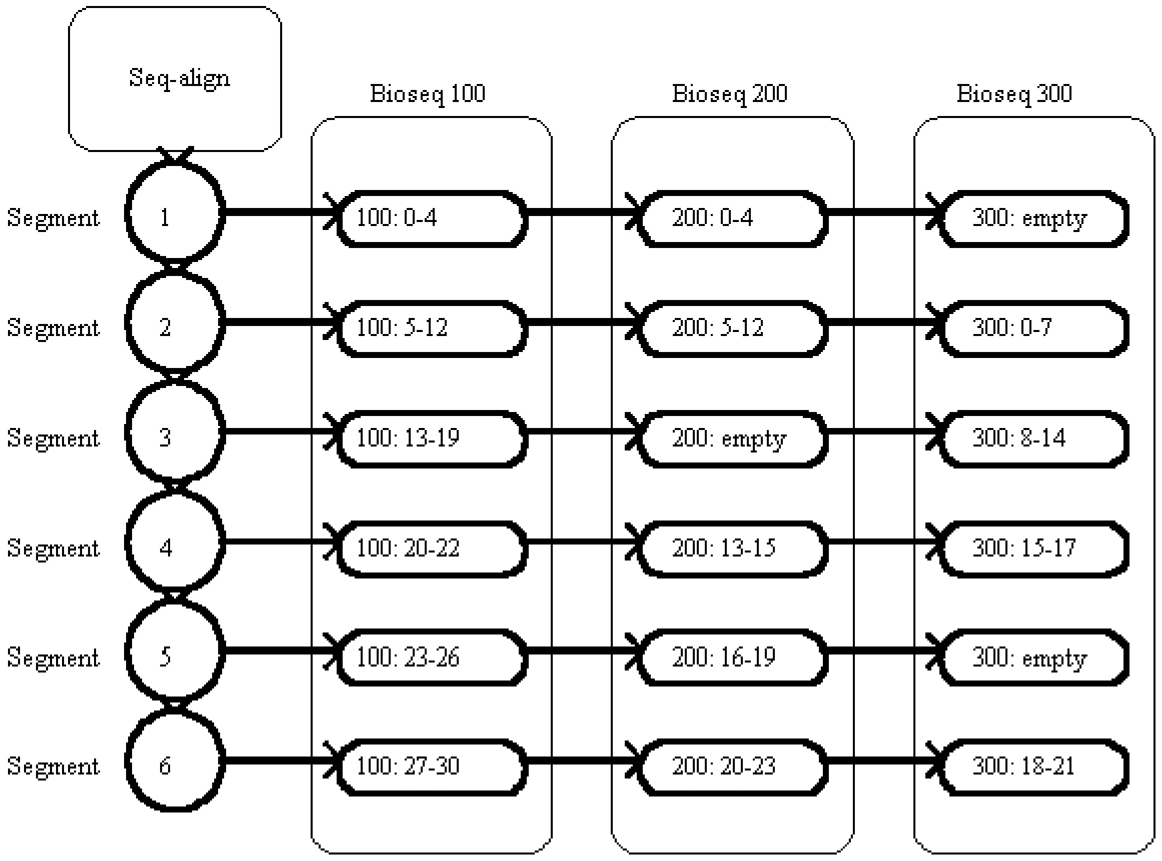
The presence of two Seq-loc's also allows a more complete representation of data conflicts or exceptional biological circumstances. If an author presents a DNA sequence and its protein product in a figure in a paper, it is possible to enter the DNA and protein sequences independently, then confirm through the CdRegion feature that the DNA in fact translates to that protein sequence. In an unfortunate number of published papers, the DNA presented does not translate to the protein presented. This may be a signal that the database has made an error of some sort, which can be caught early and corrected. Or the original paper may be in error. In this case, the "conflict" flag can be set in CdRegion, but the protein sequence is not lost, and retroactive work can be done to determine the source of the problem. It may also be the case that a genomic sequence cannot be translated to a protein for a known biological reason, such as RNA editing or suppressor tRNAs. In this case the "exception" flag can be set in Seq-feat to indicate that the data are correct, but will not behave in the expected way.

Sequence Alignments

A sequence alignment is essentially a correlation between Seq-locs, often associated with some score. An alignment is most commonly between two sequences, but it may be among many at once. In an alignment between two raw Bioseqs, a certain amount of optimization can be done in the data structure based on the knowledge that there is a one to one mapping between the residues of the sequences. So instead of recording the start and stop in Bioseq A and the start and stop in Bioseq B, it is enough to record the start in A and the start in B and the length of the aligned region. However if one is aligning a genetic map Bioseq with a physical map Bioseq, then one will wish to allow the aligned regions to distort relative one another to account for the differences from the different mapping techniques. To accommodate this most general case, there is a Seq-align type which is purely correlations between Seq-locs of any type, with no constraint that they cover exactly the same number of residues.

A Seq-align is considered to be a SEQUENCE OF segments. Each segment is an unbroken interval on a defined Bioseq, or a gap in that Bioseq. For example, let us look at the following three dimensional alignment with 6 segments:

```
Seq-ids
id=100 AAGGCCTTTTAGAGATGATGATGATGATGA
id=200 AAGGCCTaTTAG.......GATGATGATGA
id=300 ....CCTTTTAGAGATGATGAT....ATGA
| 1 | 2 | 3 | 4| 5 | 6 | Segments
```



The example above is a global alignment that is each segment sequentially maps a region of each Bioseq to a region of the others. An alignment can also be of type "diags", which is just a collection of segments with no implication about the logic of joining one segment to the next. This is equivalent to the diagonal lines that are shown on a dot-matrix plot.

The example above illustrates the most general form of a Seq-align, Std-seg, where each segment is purely a correlated set of Seq-loc. Two other forms of Seq-align allow denser packing of data for when only raw Bioseqs are aligned. These are Dense-seg, for global alignments, and Dense-diag for "diag" collections. The basic underlying model for these denser types is very similar to that shown above, but the data structure itself is somewhat different.

Sequence Graph

The third annotation type is a graph on a sequence, Seq-graph. It is basically a Seq-loc, over which to apply the graph, and a series of numbers representing values of the graph along the sequence. A software tool which calculates base composition or hydrophobic tendency might generate a Seq-graph. Additional fields in Seq-graph allow specification of axis labels, setting of ranges covered, compression of the data relative to the sequence, and so on.

Collections of Related Biological Sequences

It is often useful, even "natural", to package a group of sequences together. Some examples are a segmented Bioseq and the Bioseqs that make up its parts, a DNA sequence and its translated proteins, the separate chains of a multi-chain molecule, and so on. A Bioseq-set is such a collection of Bioseqs.

```
Bioseq-set ::= SEQUENCE { -- just a collection
 id Object-id OPTIONAL ,
 coll Dbtag OPTIONAL , -- to identify a collection
 level INTEGER OPTIONAL , -- nesting level
 class ENUMERATED {
 not-set (0) ,
 nuc-prot (1) , -- nuc acid and coded proteins
 segset (2) , -- segmented sequence + parts
 conset (3) , -- constructed sequence + parts
 parts (4) , -- parts for 2 or 3
 gibb (5) , -- geninfo backbone
 gi (6) , -- geninfo
 genbank (7) , -- converted genbank
 pir (8) , -- converted pir
 pub-set (9) , -- all the seqs from a single publication
 equiv (10) , -- a set of equivalent maps or seqs
 swissprot (11) , -- converted SWISSPROT
 pdb-entry (12) , -- a complete PDB entry
 mut-set (13) , -- set of mutations
 pop-set (14) , -- population study
 phy-set (15) , -- phylogenetic study
 eco-set (16) , -- ecological sample study
 gen-prod-set (17) , -- genomic products, chrom+mRNA+protein
 wgs-set (18) , -- whole genome shotgun project
 named-annot (19) , -- named annotation set
 named-annot-prod (20) , -- with instantiated mRNA+protein
 read-set (21) , -- set from a single read
 paired-end-reads (22) , -- paired sequences within a read-set
 other (255) } DEFAULT not-set ,
 release VisibleString OPTIONAL ,
 date Date OPTIONAL ,
 descr Seq-descr OPTIONAL ,
 seq-set SEQUENCE OF Seq-entry ,
 annot SET OF Seq-annot OPTIONAL }
```

The basic structure of a Bioseq-set is very similar to that of a Bioseq. Instead of Bioseq.id, there is a series of identifier and descriptive fields for the set. A Bioseq-set is only a convenient way of packaging sequences so controlled, stable identifiers are less important for them than they are for Bioseqs. After the first few fields the structure is exactly parallel to a Bioseq.

There are descriptors which describe aspects of the collection and the Bioseqs within the collection. The general rule for descriptors in a Bioseq-set is that they apply to "all of everything below". That is, a Bioseq-set of human sequences need have only one Org-ref descriptor for "human" at the top level of the set, and it is applied to all Bioseqs within the set.

Then follows the equivalent of Seq-inst, that is the instantiation of the data. In this case, the data is the chain of contained Bioseqs or Bioseq-sets. A Seq-entry is either a Bioseq or Bioseq-set. Seq-entry's are very often used as arguments to display and analysis functions, since one can move around either a single Bioseq or a collection of related Bioseqs in context just as easily. This also makes a Bioseq-set recursive. That is, it may consist of collections of collections.

```
Seq-entry ::= CHOICE {
 seq Bioseq ,
 set Bioseq-set }
```

Finally, a Bioseq-set may contain Seq-annot's. Generally one would put the Seq-annot's which apply to more than one Bioseq in the Bioseq-set at this level. Examples would be CdRegion features that point to DNA and protein Bioseqs, or Seq-align which align more than one Bioseq with each other. However, since Seq-annot's always explicitly cite a Seq-id, it does not matter, in terms of meaning, at what level they are put. This is in contrast to descriptors, where context does matter.

Consequences of the Data Model

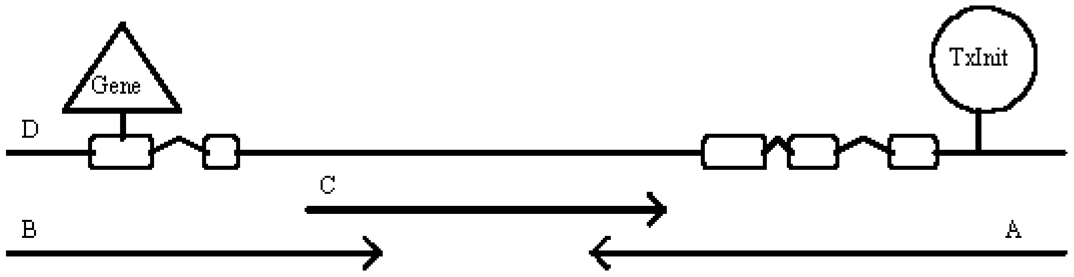
This data model has profound consequences for building sequence databases and for researchers and software tools interacting with them. Assuming that Seq-ids point to stable coordinate systems, it is easily possible to consider the whole set of data conforming to the model as a distributed, active heterogeneous database. For example, let us suppose that two raw Bioseqs with Seq-ids "A" and "B" are published in the scientific literature and appear in the large public sequence databases. They are both genomic nucleic acid sequences from human, each coding for a single protein.

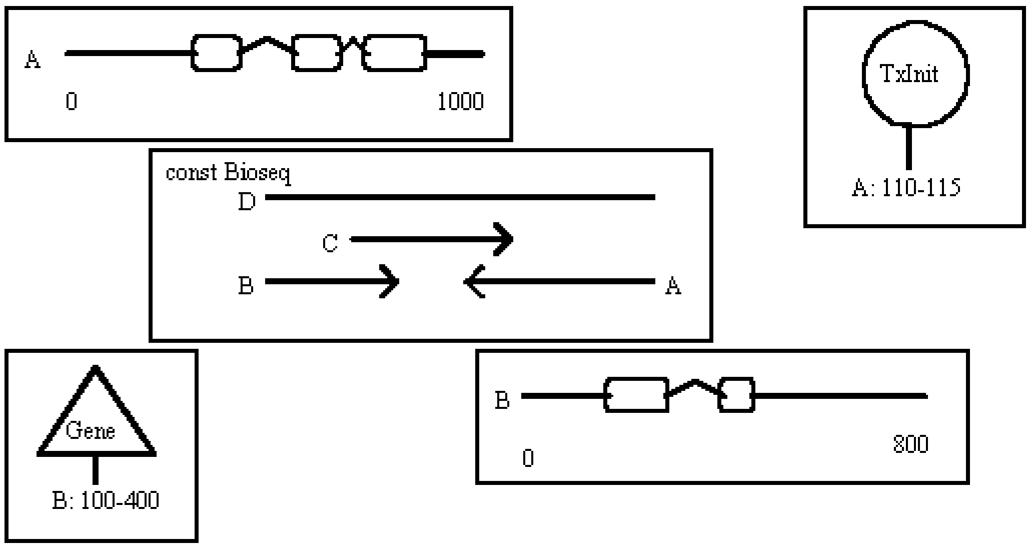
One researcher is a specialist in transcription initiation. He finds additional experimental information involving detailed work on initiation for the flanking region of Bioseq "A". He can then submit a feature table with a TxInit feature in it to the database with his summarized data. He need not contact the original author of "A", nor edit the original sequence entry for "A" to do this. The database staff, who are not experts in transcription initiation, need not attempt to annotate every transcription initiation paper in sufficient detail and accuracy to be of interest to a specialist in the area. The researcher submitting the feature need not use any particular software system or computer to participate, he need only submit a ASN.1 message which conforms to the specification for a feature.

Another researcher is a medical geneticist who is interested in the medical consequences of mutations in the gene on Bioseq "B". This individual can add annotation to "B" which is totally different in content to that added by the transcription specialist (in fact, it is unlikely that either follows the literature read by the other) and submit the data to the database in precisely the same way.

A third group may be doing bulk sequencing in the region of the human chromosome where "A" and "B" lie. They produce a third sequence, "C", which they discover by sequence similarity and mapping data, overlaps "A" at one end and "B" at the other. This group can submit not just the sequence of "C" but its relationship to "A" and "B" to the database and as part of their publication.

The database now has the information from five different research groups, experts in different fields, using different computer and software systems, and unaware, in many cases, of each other's work, to unambiguously pull together all this related information into an integrated high level view through the use of the shared data model and the controlled Seq-ids on common cited coordinate systems. This integration across disciplines and generation of high level views of the data is continuously and automatically available to all users and can be updated immediately on the arrival of new data without human intervention or interpretation by the database staff. This moves scientific databases from the role of curators of scientific data to the role of facilitators of discourse among researchers. It makes identification of potentially fruitful connections across disciplines an automatic result of data entry, rather than of painstaking analysis by a central group. It takes advantage of the growing rush of molecular biology data, making its volume and diversity advantages rather than liabilities.





Programming Considerations

To use the data model classes, add the following to your makefile:

```
LIB = general xser xutil xncbi
```

You will also need to include the relevant header files for the types you are using, and use the proper namespaces, for example:

```
#include <objects/general/Date_std.hpp>
USING_SCOPE(ncbi);
USING_SCOPE(ncbi::objects);
```

Types (such as Person-id) that contain other types can be constructed by assigning their contained types, beginning with the most nested level. For example, the following constructs a Person-id, which contains a Dbtag, which in turn contains an Object-id:

```
CObject_id obj;
obj.SetId(123);
```

```
CDbtag tag;
tag.SetDb("some db");
tag.SetTag(obj);

CPerson_id person;
person.SetDbtag(tag);
```

General Use Objects

This section describes the data objects defined in general.asn and their C++ classes and APIs. They are a miscellaneous collection of generally useful types.

- The Date: Date-std and Date
- Identifying Things: Object-id
- Identifying Things: Dbtag
- Identifying People: Name-std
- Identifying People: Person-id
- Expressing Uncertainty with Fuzzy Integers: Int-fuzz
- Creating Your Own Objects: User-object and User-field
- ASN.1 Specification: general.asn

The Date: Date-std and Date

ASN.1 has primitive types for recording dates, but they model a precise timestamp down to the minute, second, or even fraction of a second. For scientific and bibliographic data, it is common that only the date, or even just a portion of the date (e.g. month and year) is available - for example in a publication date. Rather than use artificial zero values for the unneeded fields of the ASN.1 types, we have created a specialized Date type. Date is a CHOICE of a simple, unparsed string or a structured Date-std. The string form is a fall-back for when the input data cannot be parsed into the standard date fields. It should only be used as a last resort to accommodate old data, as it is impossible to compute or index on.

When possible, the Date-std type should be used. In this case year is an integer (e.g. 1992), month is an integer from 1-12 (where January is 1), and day is an integer from 1-31. A string called "season" can be used, particularly for bibliographic citations (e.g. the "spring" issue). When a range of months is given for an issue (e.g. "JuneJuly") it cannot be represented directly. However, one would like to be able to index on integer months but still not lose the range. This is accomplished by putting 6 in the "month" slot and "July" in the "season" slot. Then the fields can be put back together for display and the issue can still be indexed by month. Year is the only required field in a Date-std.

The Date type can accommodate both the representation of the CHOICE itself (which kind of Date is this?) and the data for either CHOICE.

The Date and Date-std types are implemented with the CDate and CDate_std classes.

The CDate class should be used to create dates that can't be parsed into standard fields, for example:

```
CDate adate;
adate.SetStr("The birth of modern genetics.");
```

The CDate_std class should be used for parseable dates, i.e. dates with a given year, and optionally a month and day:

```
CDate_std adate;
adate.SetYear(2009);
adate.SetSeason("almost fall");
```

To include a time in the date:

```
CDate_std adate(CTime(CTime::eCurrent));
adate.SetSeason("late summer");
```

Identifying Things: Object-id

An Object-id is a simple structure used to identify a data object. It is just a CHOICE of an INTEGER or a VisibleString. It must always be used within some defining context (e.g. see Dbtag below) in order to have some global meaning. It allows flexibility in a host system's preference for identifying things by integers or strings.

The Object-id type is implemented by the CObject_id class. CObject_id includes the Match(), Compare(), and operator<() methods for determining whether two Object-id's are identical.

Types that include choices, such as Object-id, retain the last CHOICE assigned to them. For example, the following results in the Object-id being a string:

```
CObject_id obj;
obj.SetId(123);
obj.SetStr("some object");
```

Identifying Things: Dbtag

A Dbtag is an Object-id within the context of a database. The database is just defined by a VisibleString. The strings identifying the database are not centrally controlled, so it is possible that a conflict could occur. If there is a proliferation of Dbtags, then a registry might be considered at NCBI. Dbtags provide a simple, general way for small database providers to supply their own internal identifiers in a way which will, usually, be globally unique as well, yet requires no official sanction. So, for example, identifiers for features on sequences are not widely available at the present time. However, the Eukaryotic Promotor Database (EPD) can be provided as a set of features on sequences. The internal key to each EPD entry can be propagated as the Feature-id by using a Dbtag where "EPD" is the "db" field and an integer is used in the Object-id, which is the same integer identifying the entry in the normal EPD release.

The Dbtag type is implemented by the CDbtag class.

Identifying People: Name-std

A Name-std is a structured type for representing names with readily understood meanings for the fields. The full field is free-form and can include any or all of the other fields. The suffix field can be used for things like "Jr", "Sr", "III", etc. The title field can be used for things like "Dr.", "Sister", etc.

The Name-std type is implemented by the CName_std class.

Identifying People: Person-id

Person-id provides an extremely flexible way to identify people. There are five CHOICES from very explicit to completely unstructured. When one is building a database, one should select the most structured form possible. However, when one is processing data from other sources, one should pick the most structured form that adequately models the least structured input data expected.

The first Person-id CHOICE is a Dbtag. It would allow people to be identified by some formal registry. For example, in the USA, it might be possible to identify people by Social Security Number. Theoretically, one could then maintain a link to a person in database, even if they changed their name. Dbtag would allow other registries, such as professional societies, to be used as well. Frankly, this may be wishful thinking and possibly even socially inadvisable, though from a database standpoint, it would be very useful to have some stable identifier for people.

A Name-std CHOICE is the next most explicit form. It requires a last name and provides other optional name fields. This makes it possible to index by last name and disambiguate using one or more of the other fields (e.g. multiple people with the last name "Jones" might be distinguished by first name). This is the best choice when the data is available and its use should be encouraged by those building new databases wherever reasonable.

The next three choices contain just a single string. MEDLINE stores names in strings in a structured way (e.g. Jones JM). This means one can usually, but not always, parse out last names and can generally build indexes on the assumption that the last name is first. A consortium name can be used if the entity is a consortium rather than an individual, and finally a pure, unstructured string can be used.

The pure string form should be the CHOICE of last resort because no assumptions of any kind can be made about the structure of the name. It could be last name first, first name first, comma after last name, periods between initials, etc.

The Person-id type is implemented by the CPerson_id class.

Expressing Uncertainty with Fuzzy Integers: Int-fuzz

Lengths of Biological Sequences and locations on them are expressed with integers. However, sometimes it is desirable to be able to indicate some uncertainty about that length or location. Unfortunately, most software cannot make good use of such uncertainties, though in most cases this is fine. In order to provide both a simple, single integer view, as well as a more complex fuzzy view when appropriate, we have adopted the following strategy. In the NCBI specifications, all lengths and locations are always given by simple integers. If information about fuzziness is appropriate, then an Int-fuzz is ADDED to the data. In this case, the simple integer can be considered a "best guess" of the length or location. Thus simple software can ignore fuzziness, while it is not lost to more sophisticated uses.

Fuzziness can take a variety of forms. It can be plus or minus some fixed value. It can be somewhere in a range of values. It can be plus or minus a percentage of the best guess value. It may also be certain boundary conditions (greater than the value, less than the value) or refer to the bond BETWEEN residues of the biological sequence (bond to the right of this residue, bond to the left of that residue).

The Int-fuzz type is implemented by the CInt_fuzz class.

Creating Your Own Objects: User-object and User-field

One of the strengths of ASN.1 is that it requires a formal specification of data down to very detailed levels. This enforces clear definitions of data which greatly facilitates exchange of information in useful ways between different databases, software tools, and scientific enterprises. The problem with this approach is that it makes it very difficult for end users to add their own objects to the specification or enhance objects already in the specification. Certainly custom modules can be added to accommodate specific groups needs, but the data from such custom modules cannot be exchanged or passed through tools which adhere only to the common specification.

We have defined an object called a User-object, which can represent any class of simple, structured, or tabular data in a completely structured way, but which can be defined in any way that meets a user's needs. The User-object itself has a "class" tag which is a string used like the "db" string in Dbtag, to set the context in which this User-object is meaningful. The "class" strings are not centrally controlled, so again it is possible to have a conflict, but unlikely unless activity in this area becomes very great. Within a "class" one can define an object "type" by either a string or an integer. Thus any particular endeavor can define a wide variety of different types for their own use. The combination of "class" and "type" identifies the object to databases and software that may understand and make use this particular User-object's structure and properties. Yet, the generic definition means software that does not understand the purpose or use of any User-object can still parse it, pass it though, or even print it out for a user to peruse.

The attributes of the User-object are contained in one or more User-fields. Each User-field has a field label, which is either a string or an integer. It may contain any kind of data: strings; real numbers; integers; arrays of anything; or even sub-fields or complete sub-objects. When arrays and repeating fields are supplied, the optional "num" attribute of the User-field is used to tell software how many elements to prepare to receive. Virtually any structured data type from the simplest to the most complex can be built up from these elements.

The User-object is provided in a number of places in the public ASN.1 specifications to allow users to add their own structured features to Feature-tables or their own custom extensions to existing features. This allows new ideas to be tried out publicly, and allows software tools to be written to accommodate them, without requiring consensus among scientists or constant revisions to specifications. Those new ideas which time and experience indicate have become important concepts in molecular biology can be "graduated" to real ASN.1 specifications in the public scheme. A large body of structured data would presumably already exist in User-objects of this type, and these could all be back fitted into the new specified type, allowing data to "catch up" to the present specification. Those User-objects which do not turn out to be generally useful or important remain as harmless historical artifacts. User-objects could also be used for custom software to attach data only required for use by a particular tool to an existing standard object without harming it for use by standard tools.

The User-object and User-field types are implemented with the CUser_object and CUser_field classes.

Bibliographic References

The Bibliographic References section documents types for storing publications of any sort and collections of publications. The types are defined in biblio.asn and pub.asn modules.

Content

Introduction

The published literature is an essential component of any scientific endeavor, not just in molecular biology. The bibliographic component of the specification and the tools which go with it may find wide use then, permitting reuse of software and databases in many contexts. In addition, the fact that bibliographic citations appear in data from many sources, makes this data extremely valuable in linking data items from different databases to each other (i.e. indirectly through a shared literature citation) to build integrated views of complex data. For this reason, it is also important that database builders ensure that their literature component contain sufficient information to permit this mapping. By conforming to the specification below one can be assured that this will be the case.

Much of the following bibliographic specification was derived from the components recommended in the American National Standard for Bibliographic References (ANSI Z39.29-1977), and in interviews with professional librarians at the National Library of Medicine. The recommendations were then relaxed somewhat (by making certain fields OPTIONAL) to accommodate the less complete citation information available in current biomedical databases. Thus, although a field may be OPTIONAL, a database builder should still attempt to fill it, if it can reasonably be done.

In this section we also present a specification for the Pub type, publications of any sort and collections of publications. The MEDLINE specification has enough unique components that it is discussed separately in another section.

Citation Components: Affiliation

Affiliation is effectively the institutional affiliation of an author. Since it has the same fields needed to cite a publisher (of a book) it is reused in that context as well, although in that case it is not precisely an "affiliation". The Affil type is a CHOICE of two forms, a structured form which is preferred, or an unstructured string when that is all that is available.

The structured form has a number of fields taken from the ANSI guidelines. "affil" is institutional affiliation, such as "Harvard University". "div" is division within institution, such as "Department of Molecular Biology". "sub" is a subdivision of a country - in the United States this would be the state. "street" has been added to the specification (it is not included in ANSI) so that it is possible to produce a valid mailing address.

The Affil type is implemented by the CAffil class.

Citation Components: Authors

The Auth-list type represents the list of authors for the citation. It is a SEQUENCE, not a SET, since the order of author names matters. The names can be unstructured strings (the least desirable), semi-structured strings following the MEDLINE rules (e.g. "Jones JM"), or fully structured Author type objects (most desirable). An Affil can be associated with the whole list (typical of a scientific article). A more detailed discussion on the use of different types of names can be found in the "Identifying People" section of the "General Use Objects" section.

If fully structured Authors are used, each Author can have an individual Affil. The Author uses Person-id as defined above. The structured form also allows specification of the role of individual authors in producing the citation. The primary author(s) does not mean the "first" author, but rather that this author had a role in the original writing or experimental work. A secondary author is a reviewer or editor of the article. It is rare in a scientific work that a secondary author is ever mentioned by name. Authors may play different roles in the work, compiling, editing, and translating. Again, in a scientific work, the authors mentioned did none of these things, but were involved in the actual writing of the paper, although it would not be unusual anymore for one author to be the patent assignee. For scientific work, then, the main advantages of using the Author form are the use of fielded names and of individual Affils. For a book, being able to indicate the editors vs. the authors is useful also.

The Auth-list type is implemented by the CAuth_list class and the Author type is implemented by the CAuthor class.

Citation Components: Imprint

Imprint provides information about the physical form in which the citation appeared, such as what volume and issue of a journal it was in. For the "date" a structured Date is preferred. While "volume", "issue", and "pages" are commonly integers, there are many cases where they are not pure integers (e.g. pages xvi-xvii or issue 10A). Pages is given as a single string to simplify input from different sources. The convention is first page (hyphen) last page, or just page if it is on a single page. "section" may be relevant to a book or proceedings. "pub" is an Affil used to give the publisher of a book. The Affil.affil field is used to give the name of the publisher. "cprt" is the copyright date for a book. "part-sup" is for part or supplement and is not part of ANSI, but is used by MEDLINE. "language" is for the original language of the publication, which is also used by MEDLINE, but is not part of the ANSI standard. "prepub" is not part of the ANSI standard, but was added by NCBI to accommodate citations for as yet unpublished papers that can accompany data directly submitted by authors to the database.

The Imprint type is implemented by the CImprint class.

Citation Components: Title

A published work may have a number of Titles, each playing a particular role in specifying the work. There is the title of a paper, the title of a book it appears in, or the title of the journal, in which case it may come from a controlled list of serials. There may also be an original title and a translated title. For these reasons, Title is a defined entity rather than just a string, to allow the roles to be specified explicitly. Certain types of Title are legal for an Article, but not for a Journal or a Book. Rather than make three overlapping definitions, one for Article Titles, one for Journal Titles, and one for Book Titles, we have made one Title type and just indicated in the comments of the specification whether a particular form of Title is legal for an Article, Journal, or Book. Title is a SET OF because a work may have more than one title (e.g. an original and a translated title, or an ISO journal title abbreviation and an ISSN).

Title can be of a number of types. "name" is the full title of an article, or the full name of a book or journal. "tsub" is a subordinate title (e.g. "Hemoglobin Binds Oxygen" might be a primary title, while "Heme Groups in Biology: Part II" might be a subordinate title). "trans" is the translated title. So for an English language database like MEDLINE which contains an article originally published in French, the French title is "name" and the English version of it is "trans".

"jta" is a journal title abbreviation. It is only valid for a journal name, obviously. "jta" does not specify what kind of abbreviation it is, so it is the least useful of the journal designations available and should only be used as a last resort. "iso-jta" is an International Standards Organization (ISO) journal title abbreviation. This is the preferred form. A list of valid iso-jta's is available from NCBI or the National Library of Medicine. "ml-jta" is a MEDLINE journal title abbreviation. MEDLINE pre-dates the ISO effort, so it does not use iso-jta's. "coden" is a six letter code for journals which is used by a number of groups, particularly in Europe. "issn" is a code used by publishers to identify journals. To facilitate the use of controlled vocabularies for journal titles, NCBI maintains a file of mappings between "name", "iso-jta", "ml-jta", "coden", and "issn" where it is possible, and this file is available upon request.

"abr" is strictly the abbreviated title of a book. "isbn" is similar to "issn" in that it is a publishers abbreviation for a book. "isbn" is very useful, but one must be careful since it is used by publishers to list books, and to a publisher a hard cover book is different from a paperback (they have different "isbn"s) even if they have the same title.

The Title type is implemented by the CTitle class.

Citing an Article

An article always occurs within some other published medium. It can be an article in a journal or a chapter or section in a book or proceedings. Thus there are two components to an article citation; a citation for the work it was published in and a citation for the article within that work. Cit-art.title is the Title of the article and Cit-art.authors are the authors of the article. The "from" field is used to indicate the medium the article was published in, and reuses the standard definitions for citing a journal, book, or proceedings.

The Cit-art type is implemented by the CCit_art class.

Citing a Journal

Cit-jour is used to cite an issue of a journal, not an article within a journal (see Cit-art, above). Cit-jour.title is the title of the journal, and Cit-jour.imp gives the date, volume, issue of the journal. Cit-jour.imp also gives the pages of an article within the issue when used as part of a

Cit-art. This is not the purest possible split between article and journal, book, or proceedings, but does have the practical advantage of putting all such physical medium information together in a single common data structure. A controlled list of journal titles is maintained by NCBI, and database builders are encouraged to use this list to facilitate exchange and linking of data between databases.

The Cit-jour type is implemented by the CCit_jour class.

Citing a Book

Cit-book is used to cite a whole book, not an article within a book (see Cit-art, above). Cit-book.title is the title of this particular book. Cit-book.coll is used if the book if part of a collection, or muti-volume set (e.g. "The Complete Works of Charles Darwin"). Cit-book.authors is for the authors or editors of the book itself (not necessarily of any particular chapter). Cit-book.imp contains the publication information about the book. As with a Cit-art, if the Cit-book is being used to cite a chapter in a book, the pages in given in Cit-book.imp.

The Cit-book type is implemented by the CCit_book class.

Citing a Proceedings

A proceedings is a book published as a result or byproduct of a meeting. As such it contains all the same fields as a Cit-book and an additional block of information describing the meeting. These extra fields are the meeting number (as a string to accommodate things like "10A"), the date the meeting occurred, and an OPTIONAL Affil to record the place of the meeting. The name of the organization or meeting is normally the book title. Don't be confused by things like the Proceedings of the National Academy of Sciences, USA, which is really a journal.

The Cit-proc type is implemented by the CCit_proc class.

Citing a Letter, Manuscript, or Thesis

A letter, manuscript, or a thesis share most components and so are grouped together under type Cit-let. They all require most of the attributes of a book, and thus Cit-let incorporates the Cit-book structure. Unlike a normal book, they will not have a copyright date. A letter or manuscript will not have a publisher, although a thesis may. In addition, a manuscript may have a manuscript identifier (e.g. "Technical Report X1134").

The Cit-let type is implemented by the CCit_let class.

Citing Directly Submitted Data

The Cit-sub type is used to cite the submission of data directly to a database, independent of any publication(s) which may be associated with the data as well. Authors (of the submission) and Date (in an Imprint) are required. The Affiliation of the Authors should be filled in the Author-list. Optionally one may also record the medium in which the submission was made.

The Cit-sub type is implemented by the CCit_sub class.

Citing a Patent

A full patent citation, Cit-pat conveys not only enough information to identify a patent (see below) but to characterize it somewhat as well. A patent has a title and authors, the country in which the patent was issued, a document type and number, and the date the patent was issued. Patents are grouped into classes based on the patent subject, and this may be useful to know. In addition, when a patent is first filed it is issued an application number (different from the document number assigned to the issued patent). For tracking purposes, or issues of precedence, it is also helpful to know the application number and filing date.

The Cit-pat type is implemented by the CCit_pat class.

Identifying a Patent

When citing a patent, it may be sufficient to merely unambiguously identify it, on the assumption that more extensive information will be available from some other source, given the identifier. The Id-pat type contains fields only for the country in which the patent was applied for, or issued in, then a CHOICE of the patent document number (if issued) or the application number (if pending).

The CId-pat type is implemented by the CId_pat class.

Citing an Article or Book which is In Press

A number of the fields in Cit-art and Cit-book are OPTIONAL, not only to allow incorporation of older, incomplete databases, but also to allow partial information for works submitted, or in press. One simply fills in as many of the fields in Cit-art or Cit-book as possible. One must also set the "pre-pub" flag in Imprint to the appropriate status. That's it. Once the work is published, the remaining information is filled in and the "pre-pub" flag is removed. NOTE: this does NOT apply to work which is "unpublished" or "personal communication", or even "in preparation" because one knows nothing about where or when (or if) it will ever be published. One must use a Cit-gen for this (below).

Special Cases: Unpublished, Unparsed, or Unusual

A generic citation, Cit-gen, is used to hold anything not fitting into the more usual bibliographic entities described above. Cit-gen.cit is a string which can hold a unparsable citation (if you can parse it into a structured type, you should). Sometimes it is possible to parse some things but not everything. In this case, a number of fields, such as authors, journal, etc., which are similar to those in the structured types, can be populated as much as possible, and the remainder of the unparsed string can go in "cit".

Less standard citation types, such as a MEDLINE unique identifier, or the serial numbers used in the GenBank flatfile can be accommodated by Cit-gen. An unpublished citation normally has authors and date filled into the structured fields. Often a title is available as well (e.g. for a talk or for a manuscript in preparation). The string "unpublished" can then appear in the "cit" field.

Software developed to display or print a Cit-gen must be opportunistic about using whatever information is available. Obviously it is not possible to assume that all Cit-gens can be displayed in a uniform manner, but in practice at NCBI we have found they can generally be made fairly regular.

The Cit-gen type is implemented by the CCit_gen class.

Accommodating Any Publication Type

The Pub type is designed to accommodate a citation of any kind defined in the bibliographic specification, the MEDLINE specification, and more. It can also accommodate a collection of publications. It is very useful when one wishes to be able to associate a bibliographic reference in a very general way with a software tool or data item, yet still preserve the attributes specific for each class of citation. Pub is widely used for this purpose in the NCBI specifications.

The Pub type is implemented by the CPub class.

Grouping Different Forms of Citation for a Single Work

In some cases a database builder may wish to present more than one form of citation for the same bibliographic work. For example, in a sequence entry from the NCBI Backbone database, it is useful to provide the MEDLINE uid (for use as a link by other software tools), the Cit-art (for display to the user), and a Cit-gen containing the internal NCBI Backbone identifier for this publication as the string "pub_id = 188824" (for use in checking the database by in-house staff) for the same article. The Pub-equiv type provides this capability. It is a SET OF Pub. Each element in the SET is an equivalent citation for the same bibliographic work. Software can examine the SET and select the form most appropriate to the job at hand.

The Pub-equiv type is implemented by the CPub_equiv class.

Sets of Citations

One often needs to collect a set of citations together. Unlike the Pub-equiv type, the Pub-set type represents a set of citations for DIFFERENT bibliographic works. It is a CHOICE of types for a mixture of publication classes, or for a collection of the same publication class.

The Pub-set type is implemented by the CPub_set class.

MEDLINE Data

This section is an introduction to MEDLINE and the structure of a MEDLINE record. It describes types defined in the medline.asn module.

Module Types

Introduction

MEDLINE is the largest and oldest biomedical database in the world. It is built at the National Library of Medicine (NLM), a part of NIH. At this writing it contains over seven million citations from the scientific literature from over 3500 different journals. MEDLINE is a bibliographic database. It contains citation information (e.g. title, authors, journal, etc.). Many entries contain the abstract from the article. All articles are carefully indexed by professionals according to formal guidelines in a variety of ways. All entries can be uniquely identified by an integer key, the MEDLINE unique identifier (MEDLINE uid).

MEDLINE is a valuable resource in its own right. In addition, the MEDLINE uid can serve as a valuable link between entries in factual databases. When NCBI processes a new molecular biology factual database into the standardized format, we also normalize the bibliographic citations and attempt to map them to MEDLINE. For the biomedical databases we have tried thus far, we have succeeding in mapping most or all of the citations this way. From then on, linkage to other data objects can be made simply and easily through the shared MEDLINE uid. The MEDLINE uid also allows movement from the data item to the world of scientific literature in general and back.

Structure of a MEDLINE Entry

Each Medline-entry represents a single article from the scientific literature. The MEDLINE uid is an INTEGER which uniquely identifies the entry. If corrections are made to the contents of the entry, the uid is not changed. The MEDLINE uid is the simplest and most reliable way to identify the entry.

The entry-month (em) is the month and year in which the entry became part of the public view of MEDLINE. It is not the same as the date the article was published. It is mostly useful for tracking what is new since a previous query of MEDLINE.

The article citation itself is contained in a standard Cit-art, imported from the bibliographic module, so will not be discussed further here. The entry often contains the abstract from the article. The rest of the entry consists of various index terms, which will be discussed below.

MeSH Index Terms

Medical Subject Heading (MeSH) terms are a tree of controlled vocabulary maintained by the Library Operations division of NLM. The tree is arranged with parent terms above more specialized terms within the same concept. An entry in MEDLINE is indexed by the most specific MeSH term(s) available. Since the MeSH vocabulary is a tree, one may then query on specific terms directly, or on general terms by including all the child terms in the query as well.

A MeSH term may be qualified by one or more sub-headings. For example, the MeSH term "insulin" may carry quite a different meaning if qualified by "clinical trials" versus being qualified by "genetics".

A MeSH term or a sub-heading may be flagged as indicating the "main point" of the article. Again the most specific form is used. If the main point of the article was about insulin and they also discuss genetics, then the insulin MeSH term will be flagged but the genetics sub-heading will not be. However, if the main point of the article was the genetics of insulin, then the sub-heading genetics under the MeSH term insulin will be flagged but the MeSH term itself will not be.

Substance Records

If an article has substantial discussion of recognizable chemical compounds, they are indexed in the substance records. The record may contain only the name of the compound, or it may contain the name and a Chemical Abstracts Service (CAS) registry number or a Enzyme Commission (EC) number as appropriate.

Database Cross Reference Records

If an article cites an identifier recognized to be from a known list of biomedical databases, the cross reference is given in this field and the key for which database it was from. A typical example would be a GenBank accession number citing in an article.

Funding Identifiers

If an id number from a grant or contract is cited in the article (usually acknowledging support) it will appear in this field.

Gene Symbols

As an experiment, Library Operations at the NLM is putting in mnemonic symbols from articles, if they appear by form and usage to be gene symbols. Obviously such symbols vary and are not always properly used, so this field must be approached with caution. Nonetheless it can provide a route to a rich source of potentially relevant citations.

Biological Sequences

This section describes types used to represent biological data. These types are defined in the seq.asn, seqblock.asn, and seqcode.asn modules.

C++ Implementation Notes

Introduction

A biological sequence is a single, continuous molecule of nucleic acid or protein. It can be thought of as a multiple inheritance class hierarchy. One hierarchy is that of the underlying molecule type: DNA, RNA, or protein. The other hierarchy is the way the underlying biological sequence is represented by the data structure. It could be a physical or genetic map, an actual sequence of amino acids or nucleic acids, or some more complicated data structure building a composite view from other entries. An overview of this data model has been presented previously, in the Data Model section. The overview will not be repeated here so if you have not read that section, do so now. This section will concern itself with the details of the specification and representation of biological sequence data.

Bioseq: the Biological Sequence

A Bioseq represents a single, continuous molecule of nucleic acid or protein. It can be anything from a band on a gel to a complete chromosome. It can be a genetic or physical map. All Bioseqs have more common properties than differences. All Bioseqs must have at least one identifier, a Seq-id (i.e. Bioseqs must be citable). Seq-ids are discussed in detail in the Sequence Ids and Locations section. All Bioseqs represent an integer coordinate system (even maps). All positions on Bioseqs are given by offsets from the first residue, and thus fall in the range from zero to (length - 1). All Bioseqs may have specific descriptive data elements (descriptors) and/or annotations such as feature tables, alignments, or graphs associated with them.

The differences in Bioseqs arise primarily from the way they are instantiated (represented). Different data elements are required to represent a map than are required to represent a sequence of residues.

The C++ class for a Bioseq (CBioseq) has a list of Seq-id's, a Seq-descr, and a list of Seq-annot's, mapping quite directly from the ASN.1. However, since a Seq-inst is always required for a Bioseq, those fields have been incorporated into the Bioseq itself. Serialization is handled by CSerialObject from which CBioseq derives.

Related classes, such as CSeqdesc, provide enumerations for representing types of description, molecule types, and sequence encoding types used in the CBioseq class. Sequence encoding is discussed in more detail below.

The C++ Toolkit introduced some new methods for Bioseq's:

- CBioseq(CSeq_loc, string) - constructs a new delta sequence from the Seq-loc. The string argument may be used to specify local Seq-id text for the new Bioseq.
- GetParentEntry - returns Seq-entry containing the Bioseq.
- GetLabel - returns the Bioseq label.
- GetFirstId - returns the first element from the Bioseq's Id list or null.
- IsNa - true if the Bioseq is a nucleotide.
- IsAa - true if the Bioseq is a protein.

In addition, many utility functions for working with Bioseqs and sequence data are defined in the CSeqportUtil class.

Seq-id: Identifying the Bioseq

Every Bioseq MUST have at least one Seq-id, or sequence identifier. This means a Bioseq is always citable. You can refer to it by a label of some sort. This is a crucial property for different software tools or different scientists to be able to talk about the same thing. There is a wide range of Seq-ids and they are used in different ways. They are discussed in more detail in the Sequence Ids and Locations section.

Seq-annot: Annotating the Bioseq

A Seq-annot is a self-contained package of sequence annotations, or information that refers to specific locations on specific Bioseqs. Every Seq-annot can have an Object-id for local use by software, a Dbtag for globally identifying the source of the Seq-annot, and/or a name and description for display and use by a human. These describe the whole package of annotations and make it attributable to a source, independent of the source of the Bioseq.

A Seq-annot may contain a feature table, a set of sequence alignments, or a set of graphs of attributes along the sequence. These are described in detail in the Sequence Annotation section.

A Bioseq may have many Seq-annots. This means it is possible for one Bioseq to have feature tables from several different sources, or a feature table and set of alignments. A collection of sequences (see Sets Of Bioseqs) can have Seq-annots as well. Finally, a Seq-annot can stand alone, not directly attached to anything. This is because each element in the Seq-annot has specific references to locations on Bioseqs so the information is very explicitly associated with Bioseqs, not implicitly associated by attachment. This property makes possible the exchange of information about Bioseqs as naturally as the exchange of the Bioseqs themselves, be it among software tools or between scientists or as contributions to public databases.

Some of the important methods for the CSeq_annot class are:

- AddName() - adds or replaces annotation descriptor of type name.
- AddTitle(), SetTitle() - adds or replaces annotation descriptor of type title.
- AddComment() - adds annotation descriptor of type comment.
- SetCreateDate(), SetUpdateDate() - add or set annotation create/update time.
- AddUserObject() - add a user-object descriptor.

Seq-descr: Describing the Bioseq and Placing It In Context

A Seq-descr is meant to describe a Bioseq (or a set of Bioseqs) and place it in a biological and/or bibliographic context. Seq-descrs apply to the whole Bioseq. Some Seq-descr classes appear also as features, when used to describe a specific part of a Bioseq. But anything appearing at the Seq-descr level applies to the whole thing.

The C++ implementation of CSeq_descr uses a list of CSeqdesc objects, where each object contains a choice indicating what kind of CSeqdesc it is as well as the data representation of that choice. The CSeqdesc_Base header file lists the choice enumeration which are summarized in the following table. The Value column shows the numeric value of the choice.

Seqdesc Choice Variants

Value	Name	Explanation
0	e_not_set	choice not set
1	e_Mol_type	role of molecule in life
2	e_Modif	modifying keywords of mol-type
3	e_Method	protein sequencing method used
4	e_Name	a commonly used name (e.g. "SV40")
5	e_Title	a descriptive title or definition
6	e_Org	(single) organism from which mol comes
7	e_Comment	descriptive comment (may have many)
8	e_Num	a numbering system for whole Bioseq
9	e_Maploc	a map location from a mapping database
10	e_Pir	PIR specific data
11	e_Genbank	GenBank flatfile specific data
12	e_Pub	Publication citation and descriptive info from pub
13	e_Region	name of genome region (e.g. B-globin cluster)
14	e_User	user defined data object for any purpose
15	e_Sp	SWISSPROT specific data
16	e_Dbxref	cross reference to other databases
17	e_Embl	EMBL specific data

18	e_Create_date	date entry was created by source database
19	e_Update_date	date entry last updated by source database
20	e_Prf	PRF specific data
21	e_Pdb	PDB specific data
22	e_Het	heterogen: non-Bioseq atom/molecule
23	e_Source	source of materials, includes Org-ref
24	e_Molinfo	info on the molecule and techniques

mol-type: The Molecule Type

A Seq-descr.mol-type is of type GIBB-mol. It is derived from the molecule information used in the GenInfo BackBone database. It indicates the biological role of the Bioseq in life. It can be genomic (including organelle genomes). It can be a transcription product such as pre-mRNA, mRNA, rRNA, tRNA, snRNA (small nuclear RNA), or scRNA (small cytoplasmic RNA). All amino acid sequences are peptides. No distinction is made at this level about the level of processing of the peptide (but see Prot-ref in the Sequence Features section). The type other-genetic is provided for "other genetic material" such a B chromosomes or F factors that are not normal genomic material but are also not transcription products. The type genomic-mRNA is provided to describe sequences presented in figures in papers in which the author has combined genomic flanking sequence with cDNA sequence. Since such a figure often does not accurately reflect either the sequence of the mRNA or the sequence of genome, this practice should be discouraged.

modif: Modifying Our Assumptions About a Bioseq

A GIBB-mod began as a GenInfo BackBone component and was found to be of general utility. A GIBB-mod is meant to modify the assumptions one might make about a Bioseq. If a GIBB-mod is not present, it does not mean it does not apply, only that it is part of a reasonable assumption already. For example, a Bioseq with GIBB-mol = genomic would be assumed to be DNA, to be chromosomal, and to be partial (complete genome sequences are still rare). If GIBB-mod = mitochondrial and GIBB-mod = complete are both present in Seqdesc, then we know this is a complete mitochondrial genome. A Seqdesc contains a list of GIBB-mods.

The modifier concept permits a lot of flexibility. So a peptide with GIBB-mod = mitochondrial is a mitochondrial protein. There is no implication that it is from a mitochondrial gene, only that it functions in the mitochondrion. The assumption is that peptide sequences are complete, so GIBB-mod = complete is not necessary for most proteins, but GIBB-mod = partial is important information for some. A list of brief explanations of GIBB-mod values follows:

GIBB-mod

Value	Name	Explanation
0	eGIBB_mod_dna	molecule is DNA in life
1	eGIBB_mod_rna	molecule is RNA in life
2	eGIBB_mod_extrachrom	molecule is extrachromosomal
3	eGIBB_mod_plasmid	molecule is or is from a plasmid

4	eGIBB_mod_mitochondrial	molecule is from mitochondrion
5	eGIBB_mod_chloroplast	molecule is from chloroplast
6	eGIBB_mod_kinetoplast	molecule is from kinetoplast
7	eGIBB_mod_cyanelle	molecule is from cyanelle
8	eGIBB_mod_synthetic	molecule was synthesized artificially
9	eGIBB_mod_recombinant	molecule was formed by recombination
10	eGIBB_mod_partial	not a complete sequence for molecule
11	eGIBB_mod_complete	sequence covers complete molecule
12	eGIBB_mod_mutagen	molecule subjected to mutagenesis
13	eGIBB_mod_natmut	molecule is a naturally occurring mutant
14	eGIBB_mod_transposon	molecule is a transposon
15	eGIBB_mod_insertion_seq	molecule is an insertion sequence
16	eGIBB_mod_no_left	partial molecule is missing left end 5' end for nucleic acid, NH3 end for peptide
17	eGIBB_mod_no_right	partial molecule is missing right end 3' end for nucleic acid, COOH end for peptide
18	eGIBB_mod_macronuclear	molecule is from macronucleus
19	eGIBB_mod_proviral	molecule is an integrated provirus
20	eGIBB_mod_est	molecule is an expressed sequence tag
21	eGIBB_mod_sts	sequence tagged site
22	eGIBB_mod_survey	one pass survey sequence
23	eGIBB_mod_chromoplast	
24	eGIBB_mod_genemap	genetic map
25	eGIBB_mod_restmap	ordered restriction map
26	eGIBB_mod_physmap	physical map (not ordered restriction map)
255	eGIBB_mod_other	

method: Protein Sequencing Method

The method GetMethod() gives the method used to obtain a protein sequence. The values for a GIBB-method are stored in the object as enumerated values mapping directly from the ASN. 1 ENUMERATED type. They are:

GIBB-method

Value	Name	Explanation
1	eGIBB_method_concept_trans	conceptual translation
2	eGIBB_method_seq_pept	peptide itself was sequenced
3	eGIBB_method_both	conceptual translation with partial peptide sequencing

4	eGIBB_method_seq_pept_overlap	peptides sequenced, fragments ordered by overlap
5	eGIBB_method_seq_pept_homol	peptides sequenced, fragments ordered by homology
6	eGIBB_method_concept_trans_a	conceptual translation, provided by author of sequence

name: A Descriptive Name

A sequence name is very different from a sequence identifier. A Seq-id uniquely identifies a specific Bioseq. A Seq-id may be no more than an integer and will not necessarily convey any biological or descriptive information in itself. A name is not guaranteed to uniquely identify a single Bioseq, but if used with caution, can be a very useful tool to identify the best current entry for a biological entity. For example, we may wish to associate the name "SV40" with a single Bioseq for the complete genome of SV40. Let us suppose this Bioseq has the Seq-id 10. Then it is discovered that there were errors in the original Bioseq designated 10, and it is replaced by a new Bioseq from a curator with Seq-id 15. The name "SV40" can be moved to Seq-id 15 now. If a biologist wishes to see the "best" or "most typical" sequence of the SV40 genome, she would retrieve on the name "SV40". At an earlier point in time she would get Bioseq 10. At a later point she would get Bioseq 15. Note that her query is always answered in the context of best current data. On the other hand, if she had done a sequence analysis on Bioseq 10 and wanted to compare results, she would cite Seq-id 10, not the name "SV40", since her results apply to the specific Bioseq, 10, not necessarily to the "best" or "most typical" entry for the virus at the moment.

title: A Descriptive Title

A title is a brief, generally one line, description of an entry. It is extremely useful when presenting lists of Bioseqs returned from a query or search. This is the same as the familiar GenBank flatfile DEFINITION line.

Because of the utility of such terse summaries, NCBI has been experimenting with algorithmically generated titles which try to pack as much information as possible into a single line in a regular and readable format. You will see titles of this form appearing on entries produced by the NCBI journal scanning component of GenBank.

```
DEFINITION atp6=F0-ATPase subunit 6 {RNA edited} [Brassica napus=rapeseed,
 mRNA Mitochondrial, 905 nt]
DEFINITION mprA=metalloprotease, mprR=regulatory protein [Streptomyces
 coelicolor, Muller DSM3030, Genomic, 3 genes, 2040 nt]
DEFINITION pelBC gene cluster: pelB=pectate lyase isozyme B, pelC=pectate
 lyase isozyme C [Erwinia chrysanthemi, 3937, Genomic, 2481 nt]
DEFINITION glycoprotein J...glycoprotein I [simian herpes B virus SHBV,
 prototypic B virus, Genomic, 3 genes, 2652 nt]
DEFINITION glycoprotein B, gB [human herpesvirus-6 HHV6, GS, Peptide, 830
 aa]
DEFINITION {pseudogene} RESA-2=ring-infected erythrocyte surface antigen 2
 [Plasmodium falciparum, FCR3, Genomic, 3195 nt]
DEFINITION microtubule-binding protein tau {exons 4A, 6, 8 and 13/14} [human,
 Genomic, 954 nt, segment 1 of 4]
DEFINITION CAD protein carbamylphosphate synthetase domain {5' end} [Syrian
 hamsters, cell line 165-28, mRNA Partial, 553 nt]
DEFINITION HLA-DPB1 (SSK1)=MHC class II antigen [human, Genomic, 288 nt]
```

Gene and protein names come first. If both gene name and protein name are know they are linked with "=". If more than two genes are on a Bioseq then the first and last gene are given, separated by "...". A region name, if available, will precede the gene names. Extra comments will appear in {}. Organism, strain names, and molecule type and modifier appear in [] at the end. Note that the whole definition is constructed from structured information in the ASN.1 data structure by software. It is not composed by hand, but is instead a brief, machine generated summary of the entry based on data within the entry. We therefore discourage attempts to machine parse this line. It may change, but the underlying structured data will not. Software should always be designed to process the structured data.

org: What Organism Did this Come From?

If the whole Bioseq comes from a single organism (the usual case). See the Feature Table section for a detailed description of the Org-ref (organism reference) data structure.

comment: Commentary Text

A comment that applies to the whole Bioseq may go here. A comment may contain many sentences or paragraphs. A Bioseq may have many comments.

num: Applying a Numbering System to a Bioseq

One may apply a custom numbering system over the full length of the Bioseq with this Seqdescr. See the section on Numbering later in this chapter for a detailed description of the possible forms this can take. To report the numbering system used in a particular publication, the Pubdesc Seq-descr has its own Numbering slot.

maploc: Map Location

The map location given here is a Dbtag, to be able to cite a map location given by a map database to this Bioseq (e.g. "GDB", "4q21"). It is not necessarily the map location published by the author of the Bioseq. A map location published by the author would be part of a Pubdesc Seq-descr.

pir: PIR Specific Data

sp: SWISSPROT Data

embl: EMBL Data

prf: PRF Data

pdb: PDB Data

NCBI produces ASN.1 encoded entries from data provided by many different sources. Almost all of the data items from these widely differing sources are mapped into the common ASN.1 specifications described in this document. However, in all cases a small number of elements are unique to a particular data source, or cannot be unambiguously mapped into the common ASN.1 specification. Rather than lose such elements, they are carried in small data structures unique to each data source. These are specified in seqblock.asn and implemented by the C++ classes CGB_block, CEMBL_block, CSP_block, CPIR_block, CPRF_block, and CPDB_block.

genbank: GenBank Flatfile Specific Data

A number of data items unique to the GenBank flatfile format do not map readily to the common ASN.1 specification. These fields are partially populated by NCBI for Bioseqs derived from other sources than GenBank to permit the production of valid GenBank flatfile entries from those Bioseqs. Other fields are populated to preserve information coming from older GenBank entries.

pub: Description of a Publication

This Seq-descr is used both to cite a particular bibliographic source and to carry additional information about the Bioseq as it appeared in that publication, such as the numbering system to use, the figure it appeared in, a map location given by the author in that paper, and so. See the section on the Pubdesc later in this chapter for a more detailed description of this data type.

region: Name of a Genomic Region

A region of genome often has a name which is a commonly understood description for the Bioseq, such as "B-globin cluster".

user: A User-defined Structured Object

This is a place holder for software or databases to add their own structured datatypes to Bioseqs without corrupting the common specification or disabling the automatic ASN.1 syntax checking. A User-object can also be used as a feature. See the chapter on General User Objects for a detailed explanation of User-objects.

neighbors: Bioseqs Related by Sequence Similarity

NCBI computes a list of "neighbors", or closely related Bioseqs based on sequence similarity for use in the Entrez service. This descriptor is so that such context setting information could be included in a Bioseq itself, if desired.

create-date

This is the date a Bioseq was created for the first time. It is normally supplied by the source database. It may not be present when not normally distributed by the source database.

update-date

This is the date of the last update to a Bioseq by the source database. For several source databases this is the only date provided with an entry. The nature of the last update done is generally not available in computer readable (or any) form.

het: Heterogen

A "heterogen" is a non-biopolymer atom or molecule associated with Bioseqs from PDB. When a heterogen appears at the Seq-descr level, it means it was resolved in the crystal structure but is not associated with specific residues of the Bioseq. Heterogens which are associated with specific residues of the Bioseq are attached as features.

Seq-inst: Instantiating the Bioseq

Seq-inst.mol gives the physical type of the Bioseq in the living organism. If it is not certain if the Bioseq is DNA (dna) or RNA (rna), then (na) can be used to indicate just "nucleic acid". A protein is always (aa) or "amino acid". The values "not-set" or "other" are provided for internal use by editing and authoring tools, but should not be found on a finished Bioseq being sent to an analytical tool or database.

The representation class to which the Bioseq belongs is encoded in Seq-inst.repr. The values "not-set" or "other" are provided for internal use by editing and authoring tools, but should not be found on a finished Bioseq being sent to an analytical tool or database. The Data Model chapter discusses the representation class hierarchy in general. Specific details follow below.

Some of the important methods for Seq-inst are:

- IsAa - determines if the sequence type is amino acid
- IsNa - determines if the sequence type is nucleic acid

Seq-inst: Virtual Bioseq

A "virtual" Bioseq is one in which we know the type of molecule, and possibly its length, topology, and/or strandedness, but for which we do not have sequence data. It is not unusual to have some uncertainty about the length of a virtual Bioseq, so Seq-inst.fuzz may be used. The fields Seq-inst.seq-data and Seq-inst.ext are not appropriate for a virtual Bioseq.

Seq-inst: Raw Bioseq

A "raw" Bioseq does have sequence data, so Seq-inst.length must be set and there should be no Seq-inst.fuzz associated with it. Seq-inst.seq-data must be filled in with the sequence itself and a Seq-data encoding must be selected which is appropriate to Seq-inst.mol. The topology and strandedness may or may not be available. Seq-inst.ext is not appropriate.

Seq-inst: Segmented Bioseq

A segmented ("seg") Bioseq has all the properties of a virtual Bioseq, except that Seq-hist.ext of type Seq-ext.seg must be used to indicate the pieces of other Bioseqs to assemble to make the segmented Bioseq. A Seq-ext.seg is defined as a SEQUENCE OF Seq-loc, or a series of locations on other Bioseqs, taken in order.

For example, a segmented Bioseq (called "X") has a SEQUENCE OF Seq-loc which are an interval from position 11 to 20 on Bioseq "A" followed by an interval from position 6 to 15 on Bioseq "B". So "X" is a Bioseq with no internal gaps which is 20 residues long (no Seq-inst.fuzz). The first residue of "X" is the residue found at position 11 in "A". To obtain this residue, software must retrieve Bioseq "A" and examine the residue at "A" position 11. The segmented Bioseq contains no sequence data itself, only pointers to where to get the sequence data and what pieces to assemble in what order.

The type of segmented Bioseq described above might be used to represent the putative mRNA by simply pointing to the exons on two pieces of genomic sequence. Suppose however, that we had only sequenced around the exons on the genomic sequence, but wanted to represent the putative complete genomic sequence. Let us assume that Bioseq "A" is the genomic sequence of the first exon and some small amount of flanking DNA and that Bioseq "B" is the genomic sequence around the second exon. Further, we may know from mapping that the exons are separated by about two kilobases of DNA. We can represent the genomic region by creating a segmented sequence in which the first location is all of Bioseq "A". The second location will be all of a virtual Bioseq (call it "C") whose length is two thousand and which has a Seq-inst.fuzz representing whatever uncertainty we may have about the exact length of the intervening genomic sequence. The third location will be all of Bioseq "B". If "A" is 100 base pairs long and "B" is 200 base pairs, then the segmented entry is 2300 base pairs long ("A"+"C"+"B") and has the same Seq-inst.fuzz as "C" to express the uncertainty of the overall length.

A variation of the case above is when one has no idea at all what the length of the intervening genomic region is. A segmented Bioseq can also represent this case. The Seq-inst.ext location chain would be first all of "A", then a Seq-loc of type "null", then all of "B". The "null" indicates that there is no available information here. The length of the segmented Bioseq is just the sum of the length of "A" and the length of "B", and Seq-inst.fuzz is set to indicate the real length is greater-than the length given. The "null" location does not add to the overall length of the segmented Bioseq and is ignored in determining the integer value of a location on the segmented Bioseq itself. If "A" is 100 base pairs long and "B" is 50 base pairs long, then position 0 on the segmented Bioseq is equivalent to the first residue of "A" and position 100 on the segmented Bioseq is equivalent to the first residue of "B", despite the intervening "null" location indicating the gap of unknown length. Utility functions in the CSeqportUtil class can be configured to signal when crossing such boundaries, or to ignore them.

The Bioseqs referenced by a segmented Bioseq should always be from the same Seq-inst.mol class as the segmented Bioseq, but may well come from a mixture of Seq-inst.repr classes (as for example the mixture of virtual and raw Bioseq references used to describe sequenced and unsequenced genomic regions above). Other reasonable mixtures might be raw and map (see below) Bioseqs to describe a region which is fully mapped and partially sequenced, or even a mixture of virtual, raw, and map Bioseqs for a partially mapped and partially sequenced region. The "character" of any region of a segmented Bioseq is always taken from the underlying Bioseq to which it points in that region. However, a segmented Bioseq can have its own annotations. Things like feature tables are not automatically propagated to the segmented Bioseq.

Seq-inst: Reference Bioseq

A reference Bioseq is effectively a segmented Bioseq with only one pointer location. It behaves exactly like a segmented Bioseq in taking its data and "character" from the Bioseq to which it points. Its purpose is not to construct a new Bioseq from others like a segmented Bioseq, but to refer to an existing Bioseq. It could be used to provide a convenient handle to a frequently used region of a larger Bioseq. Or it could be used to develop a customized, personally annotated view of a Bioseq in a public database without losing the "live" link to the public sequence.

In the first example, software would want to be able to use the Seq-loc to gather up annotations and descriptors for the region and display them to user with corrections to align them appropriately to the sub region. In this form, a scientist my refer to the "lac region" by name, and analyze or annotate it as if it were a separate Bioseq, but each retrieve starts with a fresh copy of the underlying Bioseq and annotations, so corrections or additions made to the underlying Bioseq in the public database will be immediately visible to the scientist, without either having to always look at the whole Bioseq or losing any additional annotations the scientist may have made on the region themselves.

In the second example, software would not propagate annotations or descriptors from the underlying Bioseq by default (because presumably the scientist prefers his own view to the public one) but the connection to the underlying Bioseq is not lost. Thus the public annotations are available on demand and any new annotations added by the scientist share the public coordinate system and can be compared with those done by others.

Seq-inst: Constructed Bioseq

A constructed (const) Bioseq inherits all the attributes of a raw Bioseq. It is used to represent a Bioseq which has been constructed by assembling other Bioseqs. In this case the component Bioseqs normally overlap each other and there may be considerable redundancy of component Bioseqs. A constructed Bioseq is often also called a "contig" or a "merge".

Most raw Bioseqs in the public databases were constructed by merging overlapping gel or sequencer readings of a few hundred base pairs each. While the const Bioseq data structure can easily accommodate this information, the const Bioseq data type was not really intended for this purpose. It was intended to represent higher level merges of public sequence data and private data, such as when a number of sequence entries from different authors are found to overlap or be contained in each other. In this case a view of the larger sequence region can be constructed by merging the components. The relationship of the merge to the component Bioseqs is preserved in the constructed Bioseq, but it is clear that the constructed Bioseq is a "better" or "more complete" view of the overall region, and could replace the component Bioseqs in some views of the sequence database. In this way an author can submit a data structure to the database which in this author's opinion supersedes his own or other scientist's database entries, without the database actually dropping the other author's entries (who may not necessarily agree with the author submitting the constructed Bioseq).

The constructed Bioseq is like a raw, rather than a segmented, Bioseq because Seq-inst.seq-data must be present. The sequence itself is part of the constructed Bioseq. This is because the component Bioseqs may overlap in a number of ways, and expert knowledge or voting rules may have been applied to determine the "correct" or "best" residue from the overlapping regions. The Seq-inst.seq-data contains the sequence which is the final result of such a process.

Seq-inst.ext is not used for the constructed Bioseq. The relationship of the merged sequence to its component Bioseqs is stored in Seq-inst.hist, the history of the Bioseq (described in more detail below). Seq-hist.assembly contains alignments of the constructed Bioseq with its component Bioseqs. Any Bioseq can have a Seq-hist.assembly. A raw Bioseq may use this to show its relationship to its gel readings. The constructed Bioseq is special in that its Seq-hist.assembly shows how a high level view was constructed from other pieces. The sequence in a constructed Bioseq is only posited to exist. However, since it is constructed from data by possibly many different laboratories, it may never have been sequenced in its entirety from a single biological source.

Seq-inst: Typical or Consensus Bioseq

A consensus (consen) Bioseq is used to represent a pattern typical of a sequence region or family of sequences. There is no assertion that even one sequence exists that is exactly like this one, or even that the Bioseq is a best guess at what a real sequence region looks like. Instead it summarizes attributes of an aligned collection of real sequences. It could be a "typical" ferredoxin made by aligning ferredoxin sequences from many organisms and producing a protein sequence which is by some measure "central" to the group. By using the NCBIpaa encoding for the protein, which permits a probability to be assigned to each position that any of the standard amino acids occurs there, one can create a "weight matrix" or "profile" to define the sequence.

While a consensus Bioseq can represent a frequency profile (including the probability that any amino acid can occur at a position, a type of gap penalty), it cannot represent a regular expression per se. That is because all Bioseqs represent fixed integer coordinate systems. This property is essential for attaching feature tables or expressing alignments. There is no clear way to attach a fixed coordinate system to a regular expression, while one can approximate allowing weighted gaps in specific regions with a frequency profile. Since the consensus Bioseq is like any other, information can be attached to it through a feature table and alignments of the consensus pattern to other Bioseqs can be represented like any other alignment (although it may be computed a special way). Through the alignment, annotated features on the pattern can be related to matching regions of the aligned sequence in a straightforward way.

Seq-hist.assembly can be used in a consensus Bioseq to record the sequence regions used to construct the pattern and their relationships with it. While Seq-hist.assembly for a constructed Bioseq indicates the relationship with Bioseqs which are meant to be superseded by the constructed Bioseq, the consensus Bioseq does not in any way replace the Bioseqs in its Seq-hist.assembly. Rather it is a summary of common features among them, not a "better" or "more complete" version of them.

Seq-inst: Map Bioseqs

A map Bioseq inherits all the properties of a virtual Bioseq. For a consensus genetic map of E.coli, we can posit that the chromosome is DNA, circular, double-stranded, and about 5 million base pairs long. Given this coordinate system, we estimate the positions of genes on it based on genetic evidence. That is, we build a feature table with Gene-ref features on it (explained in more detail in the Feature Table chapter). Thus, a map Bioseq is a virtual Bioseq with a Seq-inst.ext which is a feature table. In this case the feature table is an essential part of instantiating the Bioseq, not simply an annotation on the Bioseq. This is not to say a map Bioseq cannot have a feature table in the usual sense as well. It can. It can also be used in alignments, displays, or by any software that can process or store Bioseqs. This is the great strength of this approach. A genetic or physical map is just another Bioseq and can be stored or analyzed right along with other more typical Bioseqs.

It is understood that within a particular physical or genetic mapping research project more data will have to be present than the map Bioseq can represent. But the same is true for a big sequencing project. The Bioseq is an object for reporting the result of such projects to others in a way that preserves most or all the information of use to workers outside the particular research group. It also preserves enough information to be useful to software tools within the project, such as display tools or analysis tools which were written by others.

A number of attributes of Bioseqs can make such a generic representation more "natural" to a particular research community. For the E.coli map example, above, no E.coli geneticist thinks of the positions of genes in base pairs (yet). So a Num-ref annotation (see Seq-descr, below) can be attached to the Bioseq, which provides information to convert the internal integer coordinate system of the map Bioseq to "minutes", the floating point numbers from 0.0 to 100.0 that E.coli gene positions are traditionally given in. Seq-loc objects which the Gene-ref features use to indicate their position can represent uncertainty, and thus give some idea of the accuracy of the mapping in a simple way. This representation cannot store order information directly (e.g. B and C are after A and before D, but we don't know the absolute distance and we don't know the relative order of B and C), which would need to be stored in a genetic mapping research database. However, a reasonable enough presentation can be made of this situation using locations and uncertainties to be very useful for a wide variety of purposes. As more sequence and physical map information become available, such uncertainties in gene position, at least for the "typical" chromosome, will gradually be resolved and will then map very will to such a generic model.

A physical map Bioseq has similar strengths and weaknesses as the genetic map Bioseq. It can represent an ordered map (such as an ordered restriction map) very well and easily. For some contig building approaches, ordering information is essential to the process of building the physical map and would have to be stored and processed separately by the map building research group. However, the map Bioseq serves very well as a vehicle for periodic reports of the group's best view of the physical map for consumption by the scientific public. The map Bioseq data structure maps quite well to the figures such groups publish to summarize their work. The map Bioseq is an electronic summary that can be integrated with other data and software tools.

Seq-hist: History of a Seq-inst

Seq-hist is literally the history of the Seq-inst part of a Bioseq. It does not track changes in annotation at all. However, since the coordinate system provided by the Seq-inst is the critical element for tying annotations and alignments done at various times by various people into a single consistent database, this is the most important element to track.

While Seq-hist can use any valid Seq-id, in practice NCBI will use the best available Seq-id in the Seq-hist. For this purpose, the Seq-id most tightly linked to the exact sequence itself is best. See the Seq-id discussion.

Seq-hist.assembly has been mentioned above. It is a SET OF Seq-align which show the relationship of this Bioseq to any older components that might be merged into it. The Bioseqs included in the assembly are those from which this Bioseq was made or is meant to supersede. The Bioseqs in the assembly need not all be from the author, but could come from anywhere. Assembly just sets the Bioseq in context.

Seq-hist.replaces makes an editorial statement using a Seq-hist-rec. As of a certain date, this Bioseq should replace the following Bioseqs. Databases at NCBI interpret this in a very specific way. Seq-ids in Seq-hist.replaces, which are owned by the owner of the Bioseq, are taken from the public view of the database. The author has told us to replace them with this one. If the author does not own some of them, it is taken as advice that the older entries may be obsolete, but they are not removed from the public view.

Seq-hist.replaced-by is a forward pointer. It means this Bioseq was replaced by the following Seq-id(s) on a certain date. In the case described above, that an author tells NCBI that a new Bioseq replaces some of his old ones, not only is the backward pointer (Seq-hist.replaces) provided by the author in the database, but NCBI will update the Seq-hist.replaced-by forward pointer when the old Bioseq is removed from public view. Since such old entries are still available for specific retrieval by the public, if a scientist does have annotation pointing to the old entry, the new entry can be explicitly located. Conversely, the older versions of a Bioseq can easily be located as well. Note that Seq-hist.replaced-by points only one generation forward and Seq-hist.replaces points only one generation back. This makes Bioseqs with a Seq-hist a doubly linked list over its revision history. This is very different from GenBank/EMBL/DDBJ secondary accession numbers, which only indicate "some relationship" between entries. When that relationship happens to be the replacement relationship, they still carry all accession numbers in the secondary accessions, not just the last ones, so reconstructing the entry history is impossible, even in a very general way.

Another fate which may await a Bioseq is that it is completely withdrawn. This is relatively rare but does happen. Seq-hist.deleted can either be set to just TRUE, or the date of the deletion event can be entered (preferred). If the deleted date is present, the ASN.1 will have the Date CHOICE for Seq-hist.deleted, else if the deleted boolean is TRUE the ASN.1 will have the BOOLEAN form.

Seq-data: Encoding the Sequence Data Itself

In the case of a raw or constructed Bioseq, the sequence data itself is stored in Seq-inst.seq-data, which is the data type Seq-data. Seq-data is a CHOICE of different ways of encoding the data, allowing selection of the optimal type for the case in hand. Both nucleic acid and amino acid encoding are given as CHOICEs of Seq-data rather than further subclassing first. But it is still not reasonable to encode a Bioseq of Seq-inst.mol of "aa" using a nucleic acid Seq-data type.

The Seq-data type is implemented in C++ with the CSeq_data class. This class has an E_Choice enumeration to identify the data representation scheme and a union to hold the sequence data.

The ASN.1 module seqcode.asn and C++ class CSeq_code_table define the allowed values for the various sequence encoding and the ways to display or map between codes. This permits useful information about the allowed encoding to be stored as ASN.1 data and read into a program at runtime. Some of the encodings are presented in tables in the following discussion of the different sequence encodings. The "value" is the internal numerical value of a residue in the C++ code. The "symbol" is a one letter or multi-letter symbol to be used in display to a human. The "name" is a descriptive name for the residue.

Some of the important methods for CSeq_data are:

- CSeq_data() - constructors to create objects from string or vector of char

IUPACaa: The IUPAC-IUB Encoding of Amino Acids

A set of one letter abbreviations for amino acids were suggested by the IUPAC-IUB Commission on Biochemical Nomenclature, published in J. Biol. Chem. (1968) 243: 3557-3559. It is very widely used in both printed and electronic forms of protein sequence, and many computer programs have been written to analyze data in this form internally (that is the actual ASCII value of the one letter code is used internally). To support such approaches, the IUPACaa encoding represents each amino acid internally as the ASCII value of its external one letter symbol. Note that this symbol is UPPER CASE. One may choose to display the value as lower case to a user for readability, but the data itself must be the UPPER CASE value.

In the NCBI C++ implementation, the values are stored one value per byte.

IUPACaa

Value	Symbol	Name
65	A	Alanine
66	B	Asp or Asn
67	C	Cysteine
68	D	Aspartic Acid
69	E	Glutamic Acid
70	F	Phenylalanine
71	G	Glycine
72	H	Histidine
73	I	Isoleucine
74	J	Leu or Ile
75	K	Lysine
76	L	Leucine
77	M	Methionine
78	N	Asparagine
79	O	Pyrrolysine
80	P	Proline
81	Q	Glutamine

82	R	Arginine
83	S	Serine
84	T	Threoine
86	V	Valine
87	W	Tryptophan
88	X	Undetermined or atypical
89	Y	Tyrosine
90	Z	Glu or Gln

NCBIeaa: Extended IUPAC Encoding of Amino Acids

The official IUPAC amino acid code has some limitations. One is the lack of symbols for termination, gap, or selenocysteine. Such extensions to the IUPAC codes are also commonly used by sequence analysis software. NCBI has created such a code which is simply the IUPACaa code above extended with the additional symbols.

In the NCBI C++ implementation, the values are stored one value per byte.

NCBIeaa

Value	Symbol	Name
42	*	Termination
45	-	Gap
65	A	Alanine
66	B	Asp or Asn
67	C	Cysteine
68	D	Aspartic Acid
69	E	Glutamic Acid
70	F	Phenylalanine
71	G	Glycine
72	H	Histidine
73	I	Isoleucine
74	J	Leu or Ile
75	K	Lysine
76	L	Leucine
77	M	Methionine
78	N	Asparagine
79	O	Pyrrolysine
80	P	Proline

81	Q	Glutamine
82	R	Arginine
83	S	Serine
84	T	Threoine
85	U	Selenocysteine
86	V	Valine
87	W	Tryptophan
88	X	Undetermined or atypical
89	Y	Tyrosine
90	Z	Glu or Gln

NCBIstdaa: A Simple Sequential Code for Amino Acids

It is often very useful to separate the external symbol for a residue from its internal representation as a data value. For amino acids NCBI has devised a simple continuous set of values that encompasses the set of "standard" amino acids also represented by the NCBIeaa code above. A continuous set of values means that compact arrays can be used in computer software to look up attributes for residues simply and easily by using the value as an index into the array. The only significance of any particular mapping of a value to an amino acid is that zero is used for gap and the official IUPAC amino acids come first in the list. In general, we recommend the use of this encoding for standard amino acid sequences.

In the NCBI C++ implementation, the values are stored one value per byte.

NCBIstdaa

Value	Symbol	Name
0	-	Gap
1	A	Alanine
2	B	Asp or Asn
3	C	Cysteine
4	D	Aspartic Acid
5	E	Glutamic Acid
6	F	Phenylalanine
7	G	Glycine
8	H	Histidine
9	I	Isoleucine
10	K	Lysine
11	L	Leucine
12	M	Methionine

13	N	Asparagine
14	P	Proline
15	Q	Glutamine
16	R	Arginine
17	S	Serine
18	T	Threoine
19	V	Valine
20	W	Tryptophan
21	X	Undetermined or atypical
22	Y	Tyrosine
23	Z	Glu or Gln
24	U	Selenocysteine
25	*	Termination
26	O	Pyrrolysine
27	J	Leu or Ile

NCBI8aa: An Encoding for Modified Amino Acids

Post-translational modifications can introduce a number of non-standard or modified amino acids into biological molecules. The NCBI8aa code will be used to represent up to 250 possible amino acids by using the remaining coding space in the NCBIstdaa code. That is, for the first 26 values, NCBI8aa will be identical to NCBIstdaa. The remaining 224 values will be used for the most commonly encountered modified amino acids. Only the first 250 values will be used to signify amino acids, leaving values in the range of 250-255 to be used for software control codes. Obviously there are a very large number of possible modified amino acids, especially if one takes protein engineering into account. However, the intent here is to only represent commonly found biological forms. This encoding is not yet available since decisions about what amino acids to include not all have been made yet.

NCBIpaa: A Profile Style Encoding for Amino Acids

The NCBIpaa encoding is designed to accommodate a frequency profile describing a protein motif or family in a form which is consistent with the sequences in a Bioseq. Each position in the sequence is defined by 30 values. Each of the 30 values represents the probability that a particular amino acid (or gap, termination, etc.) will occur at that position. One can consider each set of 30 values an array. The amino acid for each cell of the 30 value array corresponds to the NCBIstdaa index scheme. This means that currently only the first 26 array elements will ever have a meaningful value. The remaining 4 cells are available for possible future additions to NCBIstdaa. Each cell represents the probability that the amino acid defined by the NCBIstdaa index to that cell will appear at that position in the motif or protein. The probability is encoded as an 8-bit value from 0-255 corresponding to a probability from 0.0 to 1.0 by interpolation.

This type of encoding would presumably never appear except in a Bioseq of type "consensus". In the C++ implementation these amino acids are encoded at 30 bytes per amino acid in a simple linear order. That is, the first 30 bytes are the first amino acid, the second 30 the next amino acid, and so on.

IUPACna: The IUPAC-IUB Encoding for Nucleic Acids

Like the IUPACaa codes the IUPACna codes are single letters for nucleic acids and the value is the same as the ASCII value of the recommended IUPAC letter. The IUPAC recommendations for nucleic acid codes also include letters to represent all possible ambiguities at a single position in the sequence except a gap. To make the values non-redundant, U is considered the same as T. Whether a sequence actually contains U or T is easily determined from Seq-inst.mol. Since some software tools are designed to work directly on the ASCII representation of the IUPAC letters, this representation is provided. Note that the ASCII values correspond to the UPPER CASE letters. Using values corresponding to lower case letters in Seq-data is an error. For display to a user, any readable case or font is appropriate.

The C++ implementation encodes one value for a nucleic acid residue per byte.

IUPACna

Value	Symbol	Name
65	A	Adenine
66	B	G or T or C
67	C	Cytosine
68	D	G or A or T
71	G	Guanine
72	H	A or C or T
75	K	G or T
77	M	A or C
78	N	A or G or C or T
82	R	G or A
83	S	G or C
84	T	Thymine
86	V	G or C or A
87	W	A or T
89	Y	T or C

NCBI4na: A Four Bit Encoding of Nucleic Acids

It is possible to represent the same set of nucleic acid and ambiguities with a four bit code, where one bit corresponds to each possible base and where more than one bit is set to represent ambiguity. The particular encoding used for NCBI4na is the same as that used on the GenBank Floppy Disk Format. A four bit encoding has several advantages over the direct mapping of the ASCII IUPAC codes. One can represent "no base" as 0000. One can match various ambiguous or unambiguous bases by a simple AND. For example, in NCBI4na 0001=A, 0010=C, 0100=G, 1000=T/U. Adenine (0001) then matches Purine (0101) by the AND method. Finally, it is possible to store the sequence in half the space by storing two bases per byte. This is done both in the ASN.1 encoding and in the NCBI C++ software implementation.

Utility functions (see the CSeqportUtil class) allow the developer to ignore the complexities of storage while taking advantage of the greater packing. Since nucleic acid sequences can be very long, this is a real savings.

NCBI4na

Value	Symbol	Name
0	-	Gap
1	A	Adenine
2	C	Cytosine
3	M	A or C
4	G	Guanine
5	R	G or A
6	S	G or C
7	V	G or C or A
8	T	Thymine/Uracil
9	W	A or T
10	Y	T or C
11	H	A or C or T
12	K	G or T
13	D	G or A or T
14	B	G or T or C
15	N	A or G or C or T

NCBI2na: A Two Bit Encoding for Nucleic Acids

If no ambiguous bases are present in a nucleic acid sequence it can be completely encoded using only two bits per base. This allows encoding into ASN.1 or storage in the NCBI C++ implementation with a four fold savings in space. As with the four bit packing, the CSeqportUtil class allows the programmer to ignore the complexities introduced by the packing. The two bit encoding selected is the same as that proposed for the GenBank CDROM.

NCBI2na

Value	Symbol	Name
0	A	Adenine
1	C	Cytosine
2	G	Guanine
3	T	Thymine/Uracil

NCBI8na: An Eight Bit Sequential Encoding for Modified Nucleic Acids

The first 16 values of NCBI8na are identical with those of NCBI4na. The remaining possible 234 values will be used for common, biologically occurring modified bases such as those found in tRNAs. This full encoding is still being determined at the time of this writing. Only the first 250 values will be used, leaving values in the range of 250-255 to be used as control codes in software.

NCBIpna: A Frequency Profile Encoding for Nucleic Acids

Frequency profiles have been used to describe motifs and signals in nucleic acids. This can be encoded by using five bytes per sequence position. The first four bytes are used to express the probability that particular bases occur at that position, in the order A, C, G, T as in the NCBI2na encoding. The fifth position encodes the probability that a base occurs there at all. Each byte has a value from 0-255 corresponding to a probability from 0.0-1.0.

The sequence is encoded as a simple linear sequence of bytes where the first five bytes code for the first position, the next five for the second position, and so on. Typically the NCBIpna notation would only be found on a Bioseq of type consensus. However, one can imagine other uses for such an encoding, for example to represent knowledge about low resolution sequence data in an easily computable form.

Tables of Sequence Codes

Various sequence alphabets can be stored in tables of type Seq-code-table, defined in seqcode.asn. An enumerated type, Seq-code-type is used as a key to each table. Each code can be thought of as a square table essentially like those presented above in describing each alphabet. Each "residue" of the code has a numerical one-byte value used to represent that residue both in ASN.1 data and in internal C++ structures. The information necessary to display the value is given by the "symbol". A symbol can be in a one-letter series (e.g. A,G,C,T) or more than one letter (e.g. Met, Leu, etc.). The symbol gives a human readable representation that corresponds to each numerical residue value. A name, or explanatory string, is also associated with each.

So, the NCBI2na code above would be coded into a Seq-code-table very simply as:

```
{ -- NCBI2na
 code ncbi2na ,
 num 4 , -- continuous 0-3
 one-letter TRUE , -- all one letter codes
 table {
 { symbol "A", name "Adenine" },
 { symbol "C", name "Cytosine" },
 { symbol "G", name "Guanine" },
 { symbol "T", name "Thymine/Uracil"}
 } , -- end of table
 comps { -- complements
 3,
 2,
 1,
 0
 }
} ,
```

The table has 4 rows (with values 0-3) with one letter symbols. If we wished to represent a code with values which do not start at 0 (such as the IUPAC codes) then we would set the OPTIONAL "start-at" element to the value for the first row in the table.

In the case of nucleic acid codes, the Seq-code-table also has rows for indexes to complement the values represented in the table. In the example above, the complement of 0 ("A") is 3 ("T").

Mapping Between Different Sequence Alphabets

A Seq-map-table provides a mapping from the values of one alphabet to the values of another, very like the way complements are mapped above. A Seq-map-table has two Seq-code-types, one giving the alphabet to map from and the other the alphabet to map to. The Seq-map-table has the same number of rows and the same "start-at" value as the Seq-code-table for the alphabet it maps FROM. This makes the mapping a simple array lookup using the value of a residue of the FROM alphabet and subtracting "start-at". Remember that alphabets are not created equal and mapping from a bigger alphabet to a smaller may result in loss of information.

Pubdesc: Publication Describing a Bioseq

A Pubdesc is a data structure used to record how a particular publication described a Bioseq. It contains the citation itself as a Pub-equiv (see the Bibliographic References chapter) so that equivalent forms of the citation (e.g. a MEDLINE uid and a Cit-Art) can all be accommodated in a single data structure. Then a number of additional fields allow a more complete description of what was presented in the publication. These extra fields are generally only filled in for entries produced by the NCBI journal scanning component of GenBank, also known as the Backbone database. This information is not generally available in data from any other database yet.

Pubdesc.name is the name given the sequence in the publication, usually in the figure. Pubdesc.fig gives the figure the Bioseq appeared in so a scientist can locate it in the paper. Pubdesc.num preserves the numbering system used by the author (see Numbering below). Pubdesc.numexc, if TRUE, indicates that a "numbering exception" was found (i.e. the author's numbering did not agree with the number of residues in the sequence). This usually indicates an error in the preparation of the figure. If Pubdesc.poly-a is TRUE, then a poly-A tract was indicated for the Bioseq in the figure, but was not explicitly preserved in the sequence itself (e.g. ...AGAATTTCT (Poly-A)). Pubdesc.maploc is the map location for this sequence as given by the author in this paper. Pubdesc.seq-raw allows the presentation of the sequence exactly as typed from the figure. This is never used now. Pubdesc.align-group, if present, indicates the Bioseq was presented in a group aligned with other Bioseqs. The align-group value is an arbitrary integer. Other Bioseqs from the same publication which are part of the same alignment will have the same align-group number.

Pubdesc.comment is simply a free text comment associated with this publication. SWISSPROT entries may also have this field filled.

Numbering: Applying a Numbering System to a Bioseq

Internally, locations on Bioseqs are ALWAYS integer offsets in the range 0 to (length - 1). However, it is often helpful to display some other numbering system. The Numbering data structure supports a variety of numbering styles and conventions. In the ASN.1 specification, it is simply a CHOICE of the four possible types. When a Numbering object is supplied as a Seq-descr, then it applies to the complete length of the Bioseq. A Numbering object can also be a feature, in which case it only applies to the interval defined by the feature's location.

Num-cont: A Continuous Integer Numbering System

The most widely used numbering system for sequences is some form of a continuous integer numbering. Num-cont.refnum is the number to assign to the first residue in the Bioseq. If Num-cont.has-zero is TRUE, the numbering system uses zero. When biologists start numbering with a negative number, it is quite common for them to skip zero, going directly from -1 to +1, so the DEFAULT for has-zero is FALSE. This only reflects common usage, not any recommendation in terms of convention. Any useful software tool should support both conventions, since they are both used in the literature. Finally, the most common numbering systems are ascending; however descending numbering systems are encountered from time to time, so Num-cont.ascending would then be set to FALSE.

Num-real: A Real Number Numbering Scheme

Genetic maps may use real numbers as "map units" since they treat the chromosome as a continuous coordinate system, instead of a discrete, integer coordinate system of base pairs. Thus a Bioseq of type "map" which may use an underlying integer coordinate system from 0 to 5 million may be best presented to the user in the familiar 0.0 to 100.0 map units. Num-real supports a simple linear equation specifying the relationship:

map units = (Num-real.a * base_pair_position) + Num-real.b

in this example. Since such numbering systems generally have their own units (e.g. "map units", "centisomes", "centimorgans", etc), Num-real.units provides a string for labeling the display.

Num-enum: An Enumerated Numbering Scheme

Occasionally biologists do not use a continuous numbering system at all. Crystallographers and immunologists, for example, who do extensive studies on one or a few sequences, may name the individual residues in the sequence as they fit them into a theoretical framework. So one might see residues numbered ... "10" "11" "12" "12A" "12B" "12C" "13" "14" ... To accommodate this sort of scheme the "name" of each residue must be explicitly given by a string, since there is no anticipating any convention that may be used. The Num-enum.num gives the number of residue names (which should agree with the number of residues in the Bioseq, in the case of use as a Seq-descr), followed by the names as strings.

Num-ref: Numbering by Reference to Another Bioseq

Two types of references are allowed. The "sources" references are meant to apply the numbering system of constituent Bioseqs to a segmented Bioseq. This is useful for seeing the mapping from the parts to the whole.

The "aligns" reference requires that the Num-ref-aligns alignment be filled in with an alignment of the target Bioseq with one or more pieces of other Bioseqs. The numbering will come from the aligned pieces.

Numbering: C++ Class

A Numbering object is implemented by the C++ class CNumbering. The choice of numbering type is represented by the E_Choice enumeration. The class contains methods for getting and setting the various types of numbering.

Collections of Sequences

This section describes the types used to organize multiple Bioseqs into tree structures. The types are located in the seqset.asn module.

C++ Implementation Notes

Introduction

A biological sequence is often most appropriately stored in the context of other, related sequences. Such a collection might have a biological basis (e.g. a nucleic acid and its translated proteins, or the chains of an enzyme complex) or some other basis (e.g. a release of GenBank, or the sequences published in an article). The Bioseq-set provides a framework for collections of sequences.

Seq-entry: The Sequence Entry

Sometimes a sequence is not part of a collection (e.g. a single annotated protein). Thus a sequence entry could be either a single Bioseq or a collection of them. A Seq-entry is an entity which represents this choice. A great deal of NCBI software is designed to accept a Seq-entry as the primary unit of data. This is the most powerful and flexible object to use as a target software development in general.

Some of the important methods for CSeq_entry are:

- GetLabel() - append a label based on type or content of the current Seq-entry
- GetParentEntry() - gets the parent of the current Seq-entry
- Parentize() - recursive update of parent Seq-entries

Bioseq-set: A Set Of Seq-entry's

A Bioseq-set contains a convenient collection of Seq-entry's. It can have descriptors and annotations just like a single Bioseq (see Biological Sequences). It can have identifiers for the set, although these are less thoroughly controlled than Seq-ids at this time. Since the "heart" of a Bioseq-set is a collection of Seq-entry's, which themselves are either a Bioseq or a Bioseq-set, a Bioseq-set can recursively contain other sets. This recursive property makes for a very rich data structure, and a necessary one for biological sequence data, but presents new challenges for software to manipulate and display it. We will discuss some guidelines for building and using Bioseq-sets below, based on the NCBI experience to date.

Some of the important methods for CBioseq_set are:

- GetLabel() - append a label based on type or content of CBioseq_set
- GetParentSet() - gets the parent of the current CBioseq_set

id: local identifier for this set

The id field just contains an integer or string to identify this set for internal use by a software system or database. This is useful for building collections of sequences for temporary use, but still be able to cite them.

coll: global identifier for this set

The coll field is a Dbtag, which will accept a string to identify a source database and a string or integer as an identifier within that database. This semi-controlled form provides a global identifier for the set of sequences in a simple way.

level: nesting level of set

Since Bioseq-sets are recursive, the level integer was conceived as a way of explicitly indicating the nesting level. In practice we have found this to be of little or no use and recommend it be ignored and eventually removed.

class: classification of sets

The class field is an attempt to classify sets of sequences that may be widely used. There are a number which are just releases of well known databases and others which represent biological groupings.

Bioseq-set classes

The following table summarizes the types of Bioseq-sets:

Value	ASN.1 name	Explanation
0	not-set	not determined
1	nuc-prot	a nucleic acid and the proteins from its coding regions
2	segset	a segmented Bioseq and the Bioseqs it is made from
3	conset	a constructed Bioseq and the Bioseqs it was assembled from
4	parts	a set cotained within a segset or conset holding the Bioseqs which are the components of the segmented or constructed Bioseq
5	gibb	GenInfo Backbone entries (NCBI Journal Scanning Database)
6	gi	GenInfo entries (NCBI ID Database)
7	genbank	GenBank entries
8	pir	PIR entries
9	pub-set	all the Seq-entry's from a single publication
10	equiv	a set of equivalent representations of the same sequence (e.g. a genetic map Bioseq and a physical map Bioseq for the same chromosome)
11	swissprot	SWISSPROT entries
12	pdb-entry	all the Bioseqs associated with a single PDB structure
255	other	new type. Usually Bioseq-set.release will have an explanatory string

release: an explanatory string

This is just a free text field which can contain a human readable description of the set. Often used to show which release of GenBank, for example.

date

This is a date associated with the creation of this set.

descr: Seq-descr for this set

Just like a Bioseq, a Bioseq-set can have Seq-descr (see Biological Sequences) which set it in a biological or bibliographic context, or confer a title or a name. The rule for descriptors at the set level is that they apply to "all of everything below". So if an Org-ref is given at the set level, it means that every Bioseq in the set comes from that organism. If this is not true, then Org-ref would not appear on the set, but different Org-refs would occur on lower level members.

For any Bioseq in arbitrarily deeply nested Bioseq-sets, one should be able to collect all Bioseq-set.descr from all higher level Bioseq-sets that contain the Bioseq, and move them to the Bioseq. If this process introduces any confusion or contradiction, then the set level descriptor has been incorrectly used.

The only exception to this is the title and name types, which often refer to the set level on which they are placed (a nuc-prot may have the title "Adh gene and ADH protein", while the Bioseqs have the titles "Adh gene" and "ADH protein". The gain in code sharing by using exactly the same Seq-descr for Bioseq or Bioseq-set seemed to outweigh the price of this one exception to the rule.

To simplify access to elements like this that depend on a set context, a series of BioseqContext () functions are provided in utilities which allow easy access to all relevant descriptors starting with a specific Bioseq and moving up the levels in the set.

seq-set: the sequences and sets within the Bioseq-set

The seq-set field contains a SEQUENCE OF Seq-entry which represent the contents of the Bioseq-set. As mentioned above, these may be nested internally to any level. Although there is no guarantee that members of a set will come in any particular order, NCBI finds the following conventions useful and natural.

For sets of entries from specific databases, each Seq-entry is the "natural" size of an entry from those databases. Thus GenBank will contain a set of Seq-entry which will be a mixture of Bioseq (just a nucleic acid, no coding regions), seg-set (segmented nucleic acid, no coding regions), or nuc-prot (nucleic acid (as Bioseq or seg-set) and proteins from the translated coding regions). PDB will contain a mixture of Bioseq (single chain structures) or pdb-entry (multi-chain structures).

A segset, representing a segmented sequence combines the segmented Bioseq with the set of the Bioseqs that make it up.

```
segset (Bioseq-set) contains
 segmented sequence (Bioseq)
 parts (Bioseq-set) contains
 first piece (Bioseq)
 second piece (Bioseq
 etc
```

A consset has the same layout as a segset, except the top level Bioseq is constructured rather than segmented.

A nuc-prot set gives the nucleic acid and its protein products at the same levels.

```
nuc-prot (Bioseq-set) contains
 nucleic acid (Bioseq)
 protein1 (Bioseq)
 protein2 (Bioseq)
 etc.
```

A nuc-prot set where the nucleic acid is segmented simply replaces the nucleic acid Bioseq with a seg-set.

```
nuc-prot (Bioseq-set) contains
 nucleic acid segset (Bioseq-set) contains
 segmented sequence (Bioseq)
 parts (Bioseq-set) contains
 first piece (Bioseq)
 second piece (Bioseq
 etc
 protein1 (Bioseq)
 protein2 (Bioseq)
 etc.
```

annot: Seq-annots for the set

A Bioseq-set can have Seq-annots just like a Bioseq can. Because all forms of Seq-annot use explicit ids for the Bioseqs they reference, there is no dependence on context. Any Seq-annot can appear at any level of nesting in the set (or even stand alone) without any loss of information.

However, as a convention, NCBI puts the Seq-annot at the nesting level of the set that contains all the Bioseqs referenced by it, if possible. So if a feature applies just to one Bioseq, it goes in the Bioseq.annot itself. If it applies to all the members of a segmented set, it goes in Bioseq-set.annot of the segset. If, like a coding region, it points to both nucleic acid and protein sequences, it goes in the Bioseq-set.annot of the nuc-prot set.

Bioseq-sets are Convenient Packages

Remember that Bioseq-sets are just convenient ways to package Bioseqs and associated annotations. But Bioseqs may appear in various contexts and software should always be prepared to deal with them that way. A segmented Bioseq may not appear as part of a segset and a Bioseq with coding regions may not appear as part of a nuc-prot set. In both cases the elements making up the segmented Bioseq and the Bioseqs involved in the coding regions all use Seq-locs, which explicit reference Seq-ids. So they are not dependent on context. NCBI packages Bioseqs in sets for convenience, so all the closely related elements can be retrieved together. But this is only a convenience, not a requirement of the specification. The same caveat applies to the ordering conventions within a set, described above.

Sequence Locations and Identifiers

This section contains documentation for types used to identify Bioseqs and describe locations on them. These types are defined in the seqloc.asn module.

C++ Implementation Notes

- Introduction
- Seq-id: Identifying Sequences
- Seq-id: Semantics of Use
- Seq-id: The C++ Implementation

- NCBI ID Database: Imposing Stable Seq-ids
- Seq-loc: Locations on a Bioseq
- Seq-loc: The C++ Implementation
- ASN.1 Specification: seqloc.asn

Introduction

As described in the Biological Sequences chapter, a Bioseq always has at least one identifier. This means that any valid biological sequence can be referenced by using this identifier. However, all identifiers are not created equal. They may differ in their basic structure (e.g. a GenBank accession number is required to have an uppercase letter followed by exactly five digits while the NCBI GenInfo Id uses a simple integer identifier). They also differ in how they are used (e.g. the sequence identified by the GenBank accession number may change from release to release while the sequence identified by the NCBI GenInfo Id will always be exactly the same sequence).

Locations of regions on Bioseqs are always given as integer offsets, also described in the Biological Sequences chapter. So the first residue is always 0 and the last residue is always (length - 1). Further, since all the classes of Bioseqs from bands on a gel to genetic or physical maps to sequenced DNA use the same integer offset convention, locations always have the same form and meaning even when moving between very different types of Bioseq representations. This allows alignment, comparison, and display functions, among others, to have the same uniform interface and semantics, no matter what the underlying Bioseq class. Specialized numbering systems are supported but only as descriptive annotation (see Numbering in Biological Sequences and Feature types "seq" and "num" in Sequence Features). The internal conventions for positions on sequences are always the same.

There are no implicit Bioseq locations. All locations include a sequence identifier. This means Features, Alignments, and Graphs are always independent of context and can always be exchanged, submitted to databases, or stored as independent objects. The main consequence of this is that information ABOUT regions of Bioseqs can be developed and contributed to the public scientific discussion without any special rights of editing the Bioseq itself needing to be granted to anyone but the original author of the Bioseq. Bioseqs in the public databases, then, no longer need an anointed curator (beyond the original author) to be included in ongoing scientific discussion and data exchange by electronic media.

In addition to the various sequence location and identifier classes, several convenience functions for comparing or manipulating Na-strands are defined in Na_strand.hpp:

- IsForward()
- IsReverse()
- Reverse()
- SameOrientation()

Seq-id: Identifying Sequences

In a pure sense, a Seq-id is meant to unambiguously identify a Bioseq. Unfortunately, different databases have different semantic rules regarding the stability and ambiguity of their best available identifiers. For this reason a Bioseq can have more than one Seq-id, so that the Seq-id with the best semantics for a particular use can be selected from all that are available for that Bioseq, or so that a new Seq-id with different semantics can be conferred on an existing Bioseq. Further, Seq-id is defined as a CHOICE of datatypes which may differ considerably in their structure and semantics from each other. Again, this is because differing sequence databases use different conventions for identifying sequences and it is important not to lose this critical information from the original data source.

One Seq-id type, "gi", has been implemented specifically to make a simple, absolutely stable Seq-id available for sequence data derived from any source. It is discussed in detail below.

A Textseq-id structure is used in many Seq-ids described below. It has four possible fields; a name, an accession number, a release, and a version. Formally, all fields are OPTIONAL, although to be useful, a Textseq-id should have at least a name or an accession or both. This style of Seq-id is used by GenBank, EMBL, DDBJ, PIR, SWISS-PROT, and PRF, but the semantics of its use differ considerably depending on the database. However none of these databases guarantees the stability of name or accession (i.e. that it points at a specific sequence), so to be unambiguous the id must also have the version. See the discussion under Seq-id: Semantics for details.

Some important methods of the CSeq_id class are:

- CSeq_id() -- constructors to simplify creation of Seq-ids from primitive types (string, int). Some of these constructors auto-detect the type of the Seq-id from its string representation.
- Compare() -- compare Seq-ids.
- GetTextseq_Id () -- checks whether the Seq-id subtype is Textseq-id compatible and returns its value.
- IdentifyAccession() -- deduces Seq-id information from a bare accession.
- Match() -- compare Seq-ids.

Some important nonmember template functions are:

- FindGi() -- returns gi from id list if exists, returns 0 otherwise.
- FindTextseq_id() -- returns text seq-id from id list if exists, returns 0 otherwise.
- GetSeq_idByType() -- search the container of CRef<CSeq_id> for the id of given type.

Seq-id: Semantics of Use

Different databases use their ids in different ways and these patterns may change over time. An attempt is made is this section to describe current usage and offer some guidelines for interpreting Seq-ids.

local: Privately Maintained Data

The local Seq-id is an Object-id (see discussion in General Use Objects), which is a CHOICE of a string or an integer. This is to reconcile the requirement that all Bioseqs have a Seq-id and the needs of local software tools to manipulate data produced or maintained privately. This might be pre-publication data, data still being developed, or proprietary data. The Object-id will accommodate either a string or a number as is appropriate for the local environment. It is the responsibility of local software to keep the local Seq-ids unique. A local Seq-id is not globally unique, so when Bioseqs with such identifiers are published or exchanged, context (i.e. the submitter or owner of the id) must be maintained or a new id class must be applied to the Bioseq (e.g. the assignment of a GenBank accession upon direct data submission to GenBank).

refseq: From the Reference Sequence project at the NCBI

The Reference Sequence project at the NCBI aims to provide a comprehensive, integrated, non-redundant, well-annotated set of sequences, including genomic DNA, transcripts, and proteins. RefSeq assigns accessions (but not a LOCUS) to all entries. RefSeq accessions begin with two letters followed by an underscore, with additional letters after the underscore for some accessions. The leading characters have a distinct meaning as described in the RefSeq accession format reference.

general: Ids from Local Databases

The Seq-id type "general" uses a Dbtag (see discussion in General Use Objects), which is an Object-id as in Seq-id.local, above, with an additional string to identify a source database. This means that an integer or string id from a smaller database can create Seq-ids which both cite the database source and make the local Seq-ids globally unique (usually). For example, the EcoSeq database is a collection of E.coli sequences derived from many sources, maintained by Kenn Rudd. Each sequence in EcoSeq has a unique descriptive name which is used as its primary identifier. A "general" Seq-id could be make for the EcoSeq entry "EcoAce" by making the following "general" Seq-id:

```
Seq-id ::= general {
 db "EcoSeq" ,
 tag str "EcoAce" }
```

gibbsq, gibbmt: GenInfo Backbone Ids

The "gibbsq" and "gibbmt" IDs were formerly used to access the "GenInfo Backbone" database. They are now obsolete.

genbank, embl, ddbj: The International Nucleic Acid Sequence Databases

NCBI (GenBank) in the U.S., the European Molecular Biology Laboratory datalibrary (EMBL) in Europe, and the DNA Database of Japan (DDBJ) in Japan are members of an international collaboration of nucleic acid sequence databases. Each collects data, often directly submitted by authors, and makes releases of its data in its own format independently of each other. However, there are agreements in place for all the parties to exchange information with each other in an attempt to avoid duplication of effort and provide a world wide comprehensive database to their users. So a release by one of these databases is actually a composite of data derived from all three sources.

All three databases assign a mnemonic name (called a LOCUS name by GenBank and DDBJ, and an entry name by EMBL) which is meant to carry meaning encoded into it. The first few letters indicate the organism and next few a gene product, and so on. There is no concerted attempt to keep an entry name the same from release to release, nor is there any attempt for the same entry to have the same entry name in the three different databases (since they construct entry names using different conventions). While many people are used to referring to entries by name (and thus name is included in a Textseq-id) it is a notoriously unreliable way of identifying a Bioseq and should normally be avoided.

All three databases also assign an Accession Number to each entry. Accession numbers do not convey meaning, other than in a bookkeeping sense. Unlike names, accession numbers are meant to be same for the same entry, no matter which database one looks in. Thus, accession number is the best id for a Bioseq from this collaboration. Unfortunately rules for the use of accession numbers have not required that an accession number uniquely identify a sequence. A database may change an accession when it merely changes the annotation on an entry. Conversely, a database may not change an accession even though it has changed the sequence itself. There is no consistency about when such events may occur. There is also no exact method of recording the history of an entry in this collaboration, so such accession number shifts make it possible to lose track of entries by outside users of the databases. With all these caveats, accession numbers are still the best identifiers available within this collaboration.

To compensate for such shifts, it is advisable to supplement accession numbers with version numbers to yield stable, unique identifiers for all three databases. (Historically, it was likewise possible to supplement them with release fields, but those are no longer in active use and retrieval services will disregard them.)

pir: PIR International

The PIR database is also produced through an international collaboration with contributors in the US at the Protein Identification Resource of the National Biomedical Research Foundation (NBRF), in Europe at the Martinsried Institute for Protein Sequences (MIPS), and in Japan at the International Protein Information Database in Japan (JIPID). They also use an entry name and accession number. The PIR accession numbers, however, are not related to the GenBank/EMBL/DDBJ accession numbers in any way and have a very different meaning. In PIR, the entry name identifies the sequence, which is meant to be the "best version" of that protein. The accession numbers are in transition from a meaning more similar to the GenBank/EMBL/DDBJ accessions, to one in which an accession is associated with protein sequences exactly as they appeared in specific publications. Thus, at present, PIR ids may have both an accession and a name, they will move to more typically having either a name or an accession, depending on what is being cited, the "best" sequence or an original published sequence.

swissprot: UniProt Knowledgebase

Originally the Seq-id type "swissprot" referred to the Swiss-Prot database, but now it refers to the UniProtKB database. This change was made after the Swiss-Prot, TrEMBL, and PIR databases were combined to form UniProtKB. The swissprot name is meant to be easily remembered and it codes biological information, but it is not a stable identifier from release to release. The accession, which only conveys bookkeeping information, serves as a relatively stable identifier from release to release, and in conjunction with a version uniquely identifies a UniProtKB entry.

With the exception of legacy PIR entry names (which the C++ Toolkit cannot recognize when untagged), UniProtKB identifiers are coordinated with those of GenBank, EMBL, and DDBJ and do not conflict.

prf: Protein Research Foundation

The Protein Research Foundation in Japan has a large database of protein sequence and peptide fragments derived from the literature. Again, there is a name and an accession number. Since this database is meant only to record the sequence as it appeared in a particular publication, the relationship between the id and the sequence is quite stable in practice.

patent: Citing a Patent

The minimal information to unambiguously identify a sequence in a patent is first to unambiguously identify the patent (by the Patent-seq-id.cit, see Bibliographic References for a discussion of Id-pat) and then providing an integer serial number to identify the sequence within the patent. The sequence data for sequence related patents are now being submitted to the international patent offices in computer readable form, and the serial number for the sequence is assigned by the processing office. However, older sequence related patents were not assigned serial numbers by the processing patent offices. For those sequences the serial number is assigned arbitrarily (but still uniquely). Note that a sequence with a Patent-seq-id just appeared as part of a patent document. It is NOT necessarily what was patented by the patent document.

pdb: Citing a Biopolymer Chain from a Structure Database

The Protein Data Bank (PDB, also known as the Brookhaven Database), is a collection of data about structures of biological entities such hemoglobin or cytochrome c. The basic entry in PDB is a structural model of a molecule, not a sequence as in most sequence databases. A molecule may have multiple chains. So a PDB-seq-id has a string for the PDB entry name (called PDB-mol-id here) and a single character for a chain identifier within the molecule. The use of the single character just maps the PDB practice. The character may be a digit, a letter, or even a space (ASCII 32). As with the databases using the Textseq-id, the sequence of the chain in PDB associated with this information is not stable, so to be unambiguous the id must also include the release date.

giim: GenInfo Import Id

A Giimport-id ("giim") was a temporary id used to identify sequences imported into the GenInfo system at NCBI before long term identifiers, such as "gi", became stable. It is now obsolete.

gi: A Stable, Uniform Id Applied to Sequences From All Sources

A Seq-id of type "gi" is a simple integer assigned to a sequence by the NCBI "ID" database. It can be applied to a Bioseq of any representation class, nucleic acid or protein. It uniquely identifies a sequence from a particular source. If the sequence changes at all, then a new "gi" is assigned. The "gi" does not change if only annotations are changed. Thus the "gi" provides a simple, uniform way of identifying a stable coordinate system on a Bioseq provided by data sources which themselves may not have stable ids. This is the identifier of choice for all references to Bioseqs through features or alignments. See discussion below.

Seq-id: The C++ Implementation

A Seq-id is implemented in C++ as a choice, summarized in the following table:

Seq-id

Value	Enum name	Description
0	e_not_set	no variant selected
1	e_Local	local use
2	e_Gibbsq	GenInfo back-bone seq id
3	e_Gibbmt	GenInfo back-bone molecule
4	e_Giim	GenInfo import id
5	e_Genbank	genbank
6	e_Embl	embl
7	e_Pir	pir
8	e_Swissprot	swissprot
9	e_Patent	patent
10	e_Other	for historical reasons, 'other' = 'refseq'

11	e_General	general - for other databases
12	e_Gi	GenInfo integrated database
13	e_Ddbj	DDBJ
14	e_Prf	PRF SEQDB
15	e_Pdb	PDB sequence
16	e_Tpg	3rd party annot/seq Genbank
17	e_Tpeq	3rd party annot/seq EMBL
18	e_Tpd	3rd party annot/seq DDBJ
19	e_Gpipe	Internal NCBI genome pipeline processing id
20	e_Named_annot_track	Internal named annotation tracking id

A large number of additional functions for manipulating SeqIds are described in the Sequence Utilities chapter.

NCBI ID Database: Imposing Stable Seq-ids

As described in the Data Model section, Bioseqs provide a simple integer coordinate system through which a host of different data and analytical results can be easily associated with each other, even with scientists working independently of each other and on heterogeneous systems. For this model to work, however, requires stable identifiers for these integer coordinate systems. If one scientist notes a coding region from positions 10-50 of sequence "A", then the database adds a single base pair at position 5 of "A" without changing the identifier of "A", then at the next release of the database the scientist's coding region is now frame-shifted one position and invalid. Unfortunately this is currently the case due to the casual use of sequence identifiers by most existing databases.

Since NCBI integrates data from many different databases which follow their own directions, we must impose stable ids on an unstable starting material. While a daunting task, it is not, in the main, impossible. We have built a database called "ID", whose sole task is to assign and track stable sequence ids. ID assigns "gi" numbers, simple arbitrary integers which stably identify a particular sequence coordinate system.

The first time ID "sees" a Bioseq, say EMBL accession A00000, it checks to see if it has a Bioseq from EMBL with this accession already. If not, it assigns a new GI, say 5, to the entry and adds it to the Bioseq.id chain (the original EMBL id is not lost). It also replaces all references in the entry (say in the feature table) to EMBL A00000 to GI 5. This makes the annotations now apply to a stable coordinate system.

Now EMBL sends an update of the entry which is just a correction to the feature table. The same process occurs, except this time there is a previous entry with the same EMBL accession number. ID retrieves the old entry and compares the sequence of the old entry with the new entry. Since they are identical it reassigns GI 5 to the same entry, converts the new annotations, and stores it as the most current view of that EMBL entry.

Now ID gets another update to A00000, but this time the sequence is different. ID assigns a new GI, say 6, to this entry. It also updates the sequence history (Seq-inst.hist, see the Biological Sequences section) of both old and new entries to make a doubly linked list. The GI 5 entry has a pointer that it has been replaced by GI 6, and the GI 6 entry has a pointer showing it replaced GI 5. When NCBI makes a new data release the entry designated GI 6 will be released to represent EMBL entry A00000. However, the ASN.1 form of the data contains an explicit history. A scientist who annotated a coding region on GI 5 can discover that it has been replaced by GI 6. The GI 5 entry can still be retrieved from ID, aligned with GI 6, and the scientist can determine if her annotation is still valid on the new entry. If she annotated using the accession number instead of the GI, of course, she could be out of luck.

Since ID is attempting to order a chaotic world, mistakes will inevitably be made. However, it is clear that in the vast majority of cases it is possible to impose stable ids. As scientists and software begin to use the GI ids and reap the benefits of stable ids, the world may gradually become less chaotic. The Seq-inst.hist data structure can even be used by data suppliers to actively maintain an explicit history without ID having to infer it, which would be the ideal case.

Seq-loc: Locations on a Bioseq

A Seq-loc is a location on a Bioseq of any representation class, nucleic acid or protein. All Bioseqs provide a simple integer coordinate system from 0 to (length -1) and all Seq-locs refer to that coordinate system. All Seq-locs also explicitly the Bioseq (coordinate system) to which they apply with a Seq-id. Most objects which are attached to or reference sequences do so through a Seq-loc. Features are blocks of data attached by a Seq-loc. An alignment is just a collection of correlated Seq-locs. A segmented sequence is built from other sequences by reference to Seq-locs.

Some important methods of the CSeq_loc class and some of the subtype classes (CSeq_interval, CSeq_loc_mix etc.) are:

- CSeq_loc() -- constructors to simplify creation of simple Seq-loc objects.
- Compare() -- compares two Seq-locs if they are defined on the same Bioseq.
- GetTotalRange() -- returns range, covering the whole Seq-loc. If the Seq-loc refers multiple Bioseqs, exception is thrown.
- IsReverseStrand() -- returns true if all ranges in the Seq-loc have reverse strand.
- GetStart(), GetStop() -- return start and stop positions of the Seq-loc. This may be different from GetTotalRange if the related Bioseq is circular or if the order of ranges in the Seq-loc is non-standard.
- GetCircularLength() -- returns length of the Seq-loc. If the sequence length is provided, the method checks whether the Seq-loc is circular and calculates the correct length, even if the location crosses a sequence start.
- CheckId() -- checks whether the Seq-loc refers to only one Seq-id and returns it; otherwise, it sends an exception.
- Add() -- adds a sub-location to the existing one.

Beside these methods, a new class CSeq_loc_CI is defined in Seq_loc.hpp, which provides simplified access to individual ranges of any Seq-loc, regardless of its real type and structure.

null: A Gap

A null Seq-loc can be used in a Seq-loc with many components to indicate a gap of unknown size. For example it is used in segmented sequences to indicate such gaps between the sequenced pieces.

empty: A Gap in an Alignment

A alignment (see Sequence Alignments) may require that every Seq-loc refer to a Bioseq, even for a gap. They empty type fulfills this need.

whole: A Reference to a Whole Bioseq

This is just a shorthand for the Bioseq from 0 to (length -1). This form is falling out of favor at NCBI because it means one must retrieve the referenced Bioseq to determine the length of the location. An interval covering the whole Bioseq is equivalent to this and more useful. On the other hand, if an unstable Seq-id is used here, it always applies to the full length of the Bioseq, even if the length changes. This was the original rationale for this type. And it may still be valid while unstable sequences persist.

int: An Interval on a Bioseq

An interval is a single continuous region of defined length on a Bioseq. A single integer value (Seqinterval.from), another single integer value (Seq-interval.to), and a Seq-id (Seq-interval.id) are required. The "from" and "to" values must be in the range 0 to (length -1) of the Bioseq cited in "id". If there is uncertainty about either the "from" or "to" values, it is expressed in additional fields "fuzz-from" and/or "fuzz-to", and the "from" and "to" values can be considered a "best guess" location. This design means that simple software can ignore fuzzy values, but they are not lost to more sophisticated tools.

The "from" value is ALWAYS less than or equal to the "to" value, no matter what strand the interval is on. It may be convenient for software to present intervals on the minus strand with the "to" value before the "from" value, but internally this is NEVER the case. This requirement means that software which determines overlaps of locations need never treat plus or minus strand locations differently and it greatly simplifies processing.

The value of Seq-interval.strand is the only value different in intervals on the plus or minus strand. Seq-interval.strand is OPTIONAL since it is irrelevant for proteins, but operationally it will DEFAULT to plus strand on nucleic acid locations where it is not supplied.

The plus or minus strand is an attribute on each simple Seq-loc (interval or point) instead of as an operation on an arbitrarily complex location (as in the GenBank/EMBL/DDBJ flatfile Feature Table) since it means even very complex locations can be processed to a base pair location in simple linear order, instead of requiring that the whole expression be processed and resolved first.

packed-int: A Series of Intervals

A Packed-seqint is simply a SEQUENCE OF Seq-interval. That means the location is resolved by evaluating a series of Seq-interval in order. Note that the Seq-intervals in the series need not all be on the same Bioseq or on the same strand.

pnt: A Single Point on a Sequence

A Seq-point is essentially one-half of a Seq-interval and the discussion (above) about fuzziness and strand applies equally to Seq-point.

packed-pnt: A Collection of Points

A Packed-seqpnt is an optimization for attaching a large number of points to a single Bioseq. Information about the Seq-id, strand, or fuzziness need not be duplicated for every point. Of course, this also means it must apply equally to all points as well. This would typically be the case for listing all the cut sites of a certain restriction enzyme, for example.

mix: An Arbitrarily Complex Location

A Seq-loc-mix is simply a SEQUENCE OF Seq-loc. The location is resolved by resolving each Seq-loc in order. The component Seq-locs may be of any complexity themselves, making this definition completely recursive. This means a relatively small amount of software code can process locations of extreme complexity with relative ease.

A Seq-loc-mix might be used to represent a segmented sequence with gaps of unknown length. In this case it would consist of some elements of type "int" for intervals on Bioseqs and some of type "null" representing gaps of unknown length. Another use would be to combine a Seq-interval representing an untranslated leader, with a Packed-seqint from a multi-exon coding region feature, and another Seq-interval representing an untranslated 3' end, to define the extent of an mRNA on a genomic sequence.

equiv: Equivalent Locations

This form is simply a SET OF Seq-locs that are equivalent to each other. Such a construct could be used to represent alternative splicing, for example (and is when translating the GenBank/EMBL/DDBJ location "one-of"). However note that such a location can never resolve to a single result. Further, if there are multiple "equiv" forms in a complex Seq-loc, it is unclear if all possible combinations are valid. In general this construct should be avoided unless there is no alternative.

bond: A Chemical Bond Between Two Residues

The data elements in a Seq-bond are just two Seq-points. The meaning is that these two points have a chemical bond between them (which is different than describing just the location of two points). At NCBI we have restricted its use to covalent bonds. Note that the points may be on the same (intra-chain bond) or different (inter-chain bond) Bioseqs.

feat: A Location Indirectly Referenced Through A Feature

This one is really for the future, when not only Bioseqs, but features have stable ids. The meaning is "the location of this feature". This way one could give a valid location by citing, for example a Gene feature, which would resolve to the location of that gene on a Bioseq. When identifiable features become common (see Sequence Features) this will become a very useful location.

Seq-loc: The C++ Implementation

The following table summarizes the Choice variants for CSeq_loc objects.

Seq-loc

Enum Value	Enum name	ASN.1 name
0	e_not_set	
1	e_Null	null
2	e_Empty	empty
3	e_Whole	whole
4	e_Int	int
5	e_Packed_int	packed-int
6	e_Pnt	pnt

7	e_Packed_pnt	packed-pnt
8	e_Mix	mix
9	e_Equiv	equiv
10	e_Bond	bond
11	e_Feat	feat

Note that e_Mix and e_Equiv Seq-loc types can recursively contain other Seq-locs. Also, the e_Int type (implemented by CSeq_interval) has the following strand enumeration:

Enum Value	Enum name	Notes
0	e_Na_strand_unknown	
1	e_Na_strand_plus	
2	e_Na_strand_minus	
3	e_Na_strand_both	in forward direction
4	e_Na_strand_both_rev	in reverse direction
5	e_Na_strand_other	

In addition, there are a large number of utility functions for working with SeqLocs described in the chapter on Sequence Utilities. This allow traversal of complex locations, comparison of locations for overlap, conversion of coordinates in locations, and ability to open a window on a Bioseq through a location.

Sequence Features

This section documents data structures used to describe regions of Bioseqs. The types are located in the seqfeat.asn module.

C++ Implementation Notes

In the C++ Toolkit, many types defined in the seqfeat ASN.1 module are extended to simplify access to the feature data. The CSeq_feat class has methods for comparing features by type and location. The CSeqFeatData class defines feature subtypes and qualifiers so that you can better identify individual features.

- Txinit: Transcription Initiation
- Current Genetic Code Table: gc.prt
- ASN.1 Specification: seqfeat.asn

Introduction

A sequence feature (Seq-feat) is a block of structured data (SeqFeatData) explicitly attached to a region of a Bioseq through one or two Seq-locs (see Sequence Locations and Identifiers). The Seq-feat itself can carry information common to all features, as well as serving as the junction between the SeqFeatData and Seq-loc(s). Since a Seq-feat references a Bioseq through an explicit Seq-loc, a Seq-feat is an entity which can stand alone, or be moved between contexts without loss of information. Thus, information ABOUT Bioseqs can be created, exchanged, and compared independently from the Bioseq itself. This is an important attribute of the NCBI data model.

A feature table is a set of Seq-feat gathered together within a Seq-annot (see Biological Sequences). The Seq-annot allows the features to be attributed to a source and be associated with a title or comment. Seq-feats are normally exchanged "packaged" into a feature table.

Seq-feat: Structure of a Feature

A Seq-feat is a data structure common to all features. The fields it contains can be evaluated by software the same way for all features, ignoring the "data" element which is what makes each feature class unique.

id: Features Can Have Identifiers

At this time unique identifiers for features are even less available or controlled than sequence identifiers. However, as molecular biology informatics becomes more sophisticated, it will become not only useful, but essential to be able to cite features as precisely as NCBI is beginning to be able to cite sequences. The Seq-feat.id slot is where these identifiers will go. The Feat-id object for features, meant to be equivalent of the Seq-id object for Bioseqs, is not very fully developed yet. It can accommodate feature ids from the NCBI Backbone database, local ids, and the generic Dbtag type. Look for better characterized global ids to appear here in future as the requirement for structured data exchange becomes increasingly accepted.

data: Structured Data Makes Feature Types Unique

Each type of feature can have a data structure which is specifically designed to accommodate all the requirements of that type with no concern about the requirements of other feature types. Thus a coding region data structure can have fielded elements for reading frame and genetic code, while a tRNA data structure would have information about the amino acid transferred.

This design completely modularizes the components required specifically by each feature type. If a new field is required by a particular feature type, it does not affect any of the others. A new feature type, even a very complex one, can be added without affecting any of the others.

Software can be written in a very modular fashion, reflecting the data design. Functions common to all features (such as determining all features in a sequence region) simply ignore the "data" field and are robust against changes or additions to this component. Functions which process particular types have a well defined data interface unique to each type.

Perhaps a less obvious consequence is code and data reuse. Data objects used in other contexts can be used as features simply by making them a CHOICE in SeqFeatData. For example, the publication feature reuses the Pubdesc type used for Bioseq descriptors. This type includes all the standard bibliographic types (see Bibliographic References) used by MEDLINE or other bibliographic databases. Software which displays, queries, or retrieves publications will work without change on the "data" component of a publication feature because it is EXACTLY THE SAME object. This has profound positive consequences for both data and code development and maintenance.

This modularization also makes it natural to discuss each allowed feature type separately as is done in the SeqFeatData section below.

partial: This Feature is Incomplete

If Seq-feat.partial is TRUE, the feature is incomplete in some (unspecified) way. The details of incompleteness may be specified in more detail in the Seq-feat.location field. This flag allows quick exclusion of incomplete features when doing a database wide survey. It also allows the feature to be flagged when the details of incompleteness may not be know.

Seq-feat.partial should ALWAYS be TRUE if the feature is incomplete, even if Seq-feat.location indicates the incompleteness as well.

except: There is Something Biologically Exceptional

The Seq-feat.except flag is similar to the Seq-feat.partial flag in that it allows a simple warning that there is something unusual about this feature, without attempting to structure a detailed explanation. Again, this allows software scanning features in the database to ignore atypical cases easily. If Seq-feat.except is TRUE, Seq-feat.comment should contain a string explaining the exceptional situation.

Seq-feat.except does not necessarily indicate there is something wrong with the feature, but more that the biological exceeds the current representational capacity of the feature definition and that this may lead to an incorrect interpretation. For example, a coding region feature on genomic DNA where post-transcriptional editing of the RNA occurs would be a biological exception. If one translates the region using the frame and genetic code given in the feature one does not get the protein it points to, but the data supplied in the feature is, in fact, correct. It just does not take into account the RNA editing process.

Ideally, one should try to avoid or minimize exceptions by the way annotation is done. An approach to minimizing the RNA editing problem is described in the "product" section below. If one is forced to use exception consistently, it is a signal that a new or revised feature type is needed.

comment: A Comment About This Feature

No length limit is set on the comment, but practically speaking brief is better.

product: Does This Feature Produce Another Bioseq?

A Seq-feat is unusual in that it can point to two different sequence locations. The "product" location enables two Bioseqs to be linked together in a source/product relationship explicitly. This is very valuable for features which describe a transformation from one Bioseq to another, such as coding region (nucleic acid to protein) or the various RNA types (genomic nucleic acid to RNA product).

This explicit linkage is extremely valuable for connecting diverse types. Linkage of nucleic acid to protein through coding region makes data traversal from gene to product or back simple and explicit, but clearly of profound biological significance. Less obvious, but nonetheless useful is the connection between a tRNA gene and the modified sequence of the tRNA itself, or of a transcribed coding region and an edited mRNA.

Note that such a feature is as valuable in association with its product Bioseq alone as it is with its source Bioseq alone, and could be distributed with either or both.

location: Source Location of This Feature

The Seq-feat.location is the traditional location associated with a feature. While it is possible to use any Seq-loc type in Seq-feat.location, it is recommended to use types which resolve to a single unique sequence. The use of a type like Seq-loc-equiv to represent alternative splicing of exons (similar to the GenBank/EMBL/DDBJ feature table "one-of") is strongly discouraged. Consider the example of such an alternatively spliced coding region. What protein sequence is coded for by such usage? This problem is accentuated by the availability of the "product" slot. Which protein sequence is the product of this coding region? While such a short hand notation may seem attractive at first glance, it is clearly much more useful to represent each splicing alternative, and its associated protein product, times of expression, etc. separately.

qual: GenBank Style Qualifiers

The GenBank/EMBL/DDBJ feature table uses "qualifiers", a combination of a string key and a string value. Many of these qualifiers do not map to the ASN.1 specification, so this provides a means of carrying them in the Seq-feat for features derived from those sources.

title: A User Defined Name

This field is provided for naming features for display. It would be used by end-user software to allow the user to add locally meaningful names to features. This is not an id, as this is provided by the "id" slot.

ext: A User Defined Structured Extension

The "ext" field allows the extension of a standard feature type with a structured User-object (see General Use Objects) defined by a user. For example, a particular scientist may have additional detailed information about coding regions which do not fit into the standard CdRegion data type. Rather than create a completely new feature type, the CdRegion type can be extended by filling in as much of the standard CdRegion fields as possible, then putting the additional information in the User-object. Software which only expects a standard coding region will operate on the extended feature without a problem, while software that can make use of the additional data in the User-object can operate on exactly the same the feature.

cit: Citations For This Feature

This slot is a set of Pubs which are citations about the feature itself, not about the Bioseq as a whole. It can be of any type, although the most common is type "pub", a set of any kind of Pubs. The individual Pubs within the set may be Pub-equivs (see Bibliographic References) to hold equivalent forms for the same publication, so some thought should be given to the process of accessing all the possible levels of information in this seemingly simple field.

exp-ev: Experimental Evidence

If it is known for certain that there is or is not experimental evidence supporting a particular feature, Seq-feat.exp-ev can be "experimental" or "not-experimental" respectively. If the type of evidence supporting the feature is not known, exp-ev should not be given at all.

This field is only a simple flag. It gives no indication of what kind of evidence may be available. A structured field of this type will differ from feature type to feature type, and thus is inappropriate to the generic Seq-feat. Information regarding the quality of the feature can be found in the CdRegion feature and even more detail on methods in the Tx-init feature. Other feature types may gain experimental evidence fields appropriate to their types as it becomes clear what a reasonable classification of that evidence might be.

xref: Linking To Other Features

SeqFeatXrefs are copies of the Seq-feat.data field and (optionally) the Seq-feat.id field from other related features. This is a copy operation and is meant to keep some degree of connectivity or completeness with a Seq-feat that is moved out of context. For example, in a collection of data including a nucleic acid sequence and its translated protein product, there would be a Gene feature on the nucleic acid, a Prot-ref feature on the protein, and a CdRegion feature linking all three together. However, if the CdRegion feature is taken by itself, the name of the translated protein and the name of the gene are not immediately available. The Seq-feat.xref provides a simple way to copy the relevant information. Note that there is a danger to any such copy operation in that the original source of the copied data may be modified without updating the copy. Software should be careful about this, and the best course is to take the original data if it is available to the software, using any copies in xref only as a last resort. If the "id" is included in the xref, this makes it easier for software to keep the copy up to date. But it depends on widespread use of feature ids.

SeqFeatData: Type Specific Feature Data

The "data" slot of a Seq-feat is filled with SeqFeatData, which is just a CHOICE of a variety of specific data structures. They are listed under their CHOICE type below, but for most types a detailed discussion will be found under the type name itself later in this chapter, or in another chapter. That is because most types are data objects in their own right, and may find uses in many other contexts than features.

gene: Location Of A Gene

A gene is a feature of its own, rather than a modifier of other features as in the GenBank/EMBL/DDBJ feature tables. A gene is a heritable region of nucleic acid sequence which confers a measurable phenotype. That phenotype may be achieved by many components of the gene including but not limited to coding regions, promoters, enhancers, terminators, and so on. The gene feature is meant to approximately cover the region of nucleic acid considered by workers in the field to be the gene. This admittedly fuzzy concept has an appealing simplicity and fits in well with higher level views of genes such as genetic maps.

The gene feature is implemented by a Gene-ref object, or a "reference to" a gene. The Gene-ref object is discussed below.

org: Source Organism Of The Bioseq

Normally when a whole Bioseq or set of Bioseqs is from the same organism, the Org-ref (reference to Organism) will be found at the descriptor level of the Bioseq or Bioseq-set (see Biological Sequences). However, in some cases the whole Bioseq may not be from the same organism. This may occur naturally (e.g. a provirus integrated into a host chromosome) or artificially (e.g. recombinant DNA techniques).

The org feature is implemented by an Org-ref object, or a "reference to" an organism. The Orgref is discussed below.

cdregion: Coding Region

A cdregion is a region of nucleic acid which codes for a protein. It can be thought of as "instructions to translate" a nucleic acid, not simply as a series of exons or a reflection of an mRNA or primary transcript. Other features represent those things. Unfortunately, most existing sequences in the database are only annotated for coding region, so transcription and splicing information must be inferred (often inaccurately) from it. We encourage the annotation of transcription features in addition to the coding region. Note that since the cdregion is "instructions to translate", one can represent translational stuttering by having overlapping intervals in the Seq-feat.location. Again, beware of assuming a cdregion definitely reflects transcription.

A cdregion feature is implemented by a Cdregion object, discussed below.

prot: Describing A Protein

A protein feature describes and/or names a protein or region of a protein. It uses a Prot-ref object, or "reference to" a protein, described in detail below.

A single amino acid Bioseq can have many protein features on it. It may have one over its full length describing a pro-peptide, then a shorter one describing the mature peptide. An extreme case might be a viral polyprotein which would have one protein feature for the whole polyprotein, then additional protein features for each of the component mature proteins. One should always take into account the "location" slot of a protein feature.

rna: Describing An RNA

An RNA feature can describe both coding intermediates and structural RNAs using an RNA-ref, or "reference to" an RNA. The RNA-ref is described in more detail below. The Seq-feat.location for an RNA can be attached to either the genomic sequence coding for the RNA, or to the sequence of the RNA itself, when available. The determination of whether the Bioseq the RNA feature is attached to is genomic or an RNA type is made by examining the Bioseq.descr.mol-type, not by making assumptions based on the feature. When both the genomic Bioseq and the RNA Bioseq are both available, one could attach the RNA Seq-feat.location to the genomic sequence and the Seq-feat.product to the RNA to connect them and capture explicitly the process by which the RNA is created.

pub: Publication About A Bioseq Region

When a publication describes a whole Bioseq, it would normally be at the "descr" slot of the Bioseq. However, if it applies to a sub region of the Bioseq, it is convenient to make it a feature. The pub feature uses a Pubdesc (see Biological Sequences for a detailed description) to describe a publication and how it relates to the Bioseq. To indicate a citation about a specific feature (as opposed to about the sequence region in general), use the Seq-feat.cit slot of that feature.

seq: Tracking Original Sequence Sources

The "seq" feature is a simple way to associate a region of sequence with a region of another. For example, if one wished to annotate a region of a recombinant sequence as being from "pBR322 10-50" one would simply use a Seq-loc (see Sequence Locations and Identifiers) for the interval 10-50 on Seq-id pBR322. Software tools could use such information to provide the pBR322 numbering system over that interval.

This feature is really meant to accommodate older or approximate data about the source of a sequence region and is no more than annotation. More specific and computationally useful ways of doing this are (1) create the recombinant sequence as a segmented sequence directly (see Biological Sequences), (2) use the Seq-hist field of a Bioseq to record its history, (3) create alignments (see Sequence Alignments) which are also valid Seq-annots, to indicate more complex relationships of one Bioseq to others.

imp: Importing Features From Other Data Models

The SeqFeatData types explicitly define only certain well understood or widely used feature types. There may be other features contained in databases converted to this specification which are not represented by this ASN.1 specification. At least for GenBank, EMBL, DDBJ, PIR, and SWISS-PROT, these can be mapped to an Imp-feat structure so the features are not lost, although they are still unique to the source database. All these features have the basic form of a string key, a location (carried as the original string), and a descriptor (another string). In the GenBank/EMBL/DDBJ case, any additional qualifiers can be carried on the Seq-feat.qual slot.

GenBank/EMBL/DDBJ use a "location" called "replace" which is actually an editing operation on the sequence which incorporates literal strings. Since the locations defined in this specification are locations on sequences, and not editing operations, features with replace operators are all converted to Imp-feat so that the original location string can be preserved. This same strategy is taken in the face of incorrectly constructed locations encountered in parsing outside databases into ASN.1.

region: A Named Region

The region feature provides a simple way to name a region of a Bioseq (e.g. "globin locus", "LTR", "subrepeat region", etc).

comment: A Comment On A Region Of Sequence

The comment feature allows a comment to be made about any specified region of sequence. Since comment is already a field in Seq-feat, there is no need for an additional type specific data item in this case, so it is just NULL.

bond: A Bond Between Residues

This feature annotates a bond between two residues. A Seq-loc of type "bond" is expected in Seq-feat.location. Certain types of bonds are given in the ENUMERATED type. If the bond type is "other" the Seq-feat.comment slot should be used to explain the type of the bond. Allowed bond types are:

```
disulfide (1) ,
thiolester (2) ,
xlink (3) ,
thioether (4) ,
other (255)
```

site: A Defined Site

The site feature annotates a know site from the following specified list. If the site is "other" then Seq-feat.comment should be used to explain the site.

```
active (1) ,
binding (2) ,
cleavage (3) ,
inhibit (4) ,
modified (5),
glycosylation (6) ,
myristoylation (7) ,
mutagenized (8) ,
metal-binding (9) ,
phosphorylation (10) ,
acetylation (11) ,
amidation (12) ,
methylation (13) ,
hydroxylation (14) ,
sulfatation (15) ,
oxidative-deamination (16) ,
pyrrolidone-carboxylic-acid (17) ,
gamma-carboxyglutamic-acid (18) ,
blocked (19) ,
lipid-binding (20) ,
np-binding (21) ,
dna-binding (22) ,
other (255)
```

rsite: A Restriction Enzyme Cut Site

A restriction map is basically a feature table with rsite features. Software which generates such a feature table could then use any sequence annotation viewer to display its results. Restriction maps generated by physical methods (before sequence is available), can use this feature to create a map type Bioseq representing the ordered restriction map. For efficiency one would probably create one Seq-feat for each restriction enzyme used and used the Packed-pnt Seq-loc in the location slot. See Rsite-ref, below.

user: A User Defined Feature

An end-user can create a feature completely of their own design by using a User-object (see General Use Objects) for SeqFeatData. This provides a means for controlled addition and testing of new feature types, which may or may not become widely accepted or to "graduate" to a defined SeqFeatData type. It is also a means for software to add structured information to Bioseqs for its own use and which may never be intended to become a widely used standard. All the generic feature operations, including display, deletion, determining which features are carried on a sub region of sequence, etc, can be applied to an user feature with no knowledge of the particular User-object structure or meaning. Yet software which recognizes that User-object can take advantage of it.

If an existing feature type is available but lacks certain additional fields necessary for a special task or view of information, then it should be extended with the Seq-feat.ext slot, rather than building a complete user feature de novo.

txinit: Transcription Initiation

This feature is used to designate the region of transcription initiation, about which considerable knowledge is available. See Txinit, below.

num: Applying Custom Numbering To A Region

A Numbering object can be used as a Bioseq descriptor to associate various numbering systems with an entire Bioseq. When used as a feature, the numbering system applies only to the region in Seq-feat.location. This make multiple, discontinuous numbering systems available on the same Bioseq. See Biological Sequences for a description of Numbering, and also Seq-feat.seq, above, for an alternative way of applying a sequence name and its numbering system to a sequence region.

psec-str: Protein Secondary Structure

Secondary structure can be annotated on a protein sequence using this type. It can be predicted by algorithm (in which case Seq-feat.exp-ev should be "not-experimental") or by analysis of the known protein structure (Seq-feat.exp-ev = "experimental"). Only three types of secondary structure are currently supported. A "helix" is any helix, a "sheet" is beta sheet, and "turn" is a beta or gamma turn. Given the controversial nature of secondary structure classification (not be mention prediction), we opted to keep it simple until it was clear that more detail was really necessary or understood.

non-std-residue: Unusual Residues

When an unusual residue does not have a direct sequence code, the "best" standard substitute can be used in the sequence and the residue can be labeled with its real name. No attempt is made to enforce a standard nomenclature for this string.

het: Heterogen

In the PDB structural database, non-biopolymer atoms associated with a Bioseq are referred to as "heterogens". When a heterogen appears as a feature, it is assumed to be bonded to the sequence positions in Seq-feat.location. If there is no specific bonding information, the heterogen will appear as a descriptor of the Bioseq. The Seq-loc for the Seq-feat.location will probably be a point or points, not a bond. A Seq-loc of type bond is between sequence residues.

Seq-feat Implementation in C++

The C++ implementation of a Seq-feat is mostly straightforward. However, some explanation of the "id" and "data" slots will be helpful. Both are implemented as a Choice and contained in the CSeq_feat object. The tables below summarize the id and data choice variants.

SeqFeat.id

ASN.1 name	Value
(not present)	0
gibb	1
giim	2
local	3
general	4

SeqFeat.data

ASN.1 name	Value
(not present)	0
gene	1
org	2
cdregion	3

prot	4
rna	5
pub	6
seq	7
imp	8
region	9
comment	10
bond	11
site	12
rsite	13
user	14
txinit	15
num	16
psec-str	17
non-std-residue	18
het	19
biosrc	20
clone	21

Of course, within the software tools for producing GenBank, report, or other formats from ASN.1 are functions to format and display features as well. There are some functions to manipulate the CSeqFeatData objects, such as the translation of a CdRegion, and a host of functions to use and compare the Seq-locs of "product" and "location" or easily access and use the sequence regions they point to. These functions are discussed in the Sequence Utilities chapter. Additional functions, described in Exploring The Data, allow one to easily locate features of interest by type, in arbitrarily complex objects.

CdRegion: Coding Region

A CdRegion, in association with a Seq-feat, is considered "instructions to translate" to protein. The Seq-locs used by the Seq-feat do not necessarily reflect the exon structure of the primary transcript (although they often do). A Seq-feat of type CdRegion can point both to the source nucleic acid and to the protein sequence it produces. Most of the information about the source nucleic acid (such as the gene) or the destination protein (such as its name) is associated directly with those Bioseqs. The CdRegion only serves as a link between them, and as a method for explicitly encoding the information needed to derive one from the other.

orf: Open Reading Frame

CdRegion.orf is TRUE if the coding region is only known to be an open reading frame. This is a signal that nothing is known about the protein product, or even if it is produced. In this case the translated protein sequence will be attached, but there will be no other information associated with it. This flag allows such very speculative coding regions to be easily ignored when scanning the database for genuine protein coding regions.

The orf flag is not set when any reasonable argument can be made that the CdRegion is really expressed, such as detection of mRNA or strong sequence similarity to known proteins.

Translation Information

CdRegion has several explicit fields to define how to translate the coding region. Reading frame is explicitly given or defaults to frame one.

The genetic code is assumed to be the universal code unless given explicitly. The code itself is given, rather than requiring software to determine the code at run-time by analyzing the phylogenetic position of the Bioseq. Genetic code is described below.

Occasionally the genetic code is not followed at specific positions in the sequence. Examples are the use of alternate initiation codons only in the first position, the effects of suppresser tRNAs, or the addition of selenocysteine. The Code-break object specifies the three bases of the codon in the Bioseq which is treated differently and the amino acid which is generated at that position. During translation the genetic code is followed except at positions indicated by Code-breaks, where the instructions in the Code-break are followed instead.

Problems With Translations

In a surprising number of cases an author publishes both a nucleic acid sequence and the protein sequence produced by its coding region, but the translation of the coding region does not yield the published protein sequence. On the basis of the publication it is not possible to know for certain which sequence is correct. In the NCBI Backbone database both sequences are preserved as published by the author, but the conflict flag is set to TRUE in the CdRegion. If available, the number of gaps and mismatches in the alignment of the translated sequence to the published protein sequence are also given so a judgment can be made about the severity of the problem.

Genetic Codes

A Genetic-code is a SET which may include one or more of a name, integer id, or 64 cell arrays of amino acid codes in different alphabets. Thus, in a CdRegion, one can either refer to a genetic code by name or id; provide the genetic code itself, or both. Tables of genetic codes are provided in the NCBI software release with most possibilities filled in.

The Genetic-code.name is a descriptive name for the genetic code, mainly for display to humans. The integer id refers to the ids in the gc.val (binary ASN.1) or gc.prt (text ASN.1) file of genetic codes maintained by NCBI, distributed with the software tools and Entrez releases, and published in the GenBank/EMBL/DDBJ feature table document. Genetic-code.id is the best way to explicitly refer to a genetic code.

The genetic codes themselves are arrays of 64 amino acid codes. The index to the position in the array of the amino acid is derived from the codon by the following method:

index = (base1 * 16) + (base2 * 4) + base3

where T=0, C=1, A=2, G=3

Note that this encoding of the bases is not the same as any of the standard nucleic acid encoding described in Biological Sequence. This set of values was chosen specifically for genetic codes because it results in the convenient groupings of amino acid by codon preferred for display of genetic code tables.

The genetic code arrays have names which indicate the amino acid alphabet used (e.g. ncbieaa). The same encoding technique is used to specify start codons. Alphabet names are prefixed with "s" (e.g. sncbieaa) to indicate start codon arrays. Each cell of a start codon array contains either the gap code ("-" for ncbieaa) or an amino acid code if it is valid to use the codon as a start codon. Currently all starts are set to code for methionine, since it has never been convincingly demonstrated that a protein can start with any other amino acid. However, if other amino acids are shown to be used as starts, this structure can easily accommodate that information.

The contents of gc.prt, the current supported genetic codes, is given at the end of this chapter.

C++ Implementation Of Genetic Codes

The GeneticCode type is summarized as follows:

GeneticCode Elements

ASN.1 name	Value
name	1
id	2
ncbieaa	3
ncbi8aa	4
ncbistdaa	5
sncbieaa	6
sncbi8aa	7
sncbistdaa	8

Rsite-ref: Reference To A Restriction Enzyme

This simple data structure just references a restriction enzyme. It is a choice of a simple string (which may or may not be from a controlled vocabulary) or a Dbtag, in order to cite an enzyme from a specific database such as RSITE. The Dbtag is preferred, if available.

Note that this reference is not an Rsite-entry which might contain a host of information about the restriction enzyme, but is only a reference to the enzyme.

RNA-ref: Reference To An RNA

An RNA-ref allows naming and a minimal description of various RNAs. The "type" is a controlled vocabulary for dividing RNAs into broad, well accepted classes. The "pseudo" field is used for RNA pseudogenes.

The "ext" field allows the addition of structure information appropriate to a specific RNA class as appropriate. The "name" extension allows naming the "other" type or adding a modifier, such as "28S" to rRNA. For tRNA there is a structured extension which as fields for the amino acid transferred, drawn from the standard amino acid alphabets, and a value for one or more codons that this tRNA recognizes. The values of the codons are calculated as a number from 0 to 63 using the same formula as for calculating the index to Genetic Codes, above.

As nomenclature and attributes for classes of RNAs becomes better understood and accepted, the RNA-ref.ext will gain additional extensions.

Gene-ref: Reference To A Gene

A Gene-ref is not intended to carry all the information one might want to know about a gene, but to provide a small set of information and reference some larger body of information, such as an entry in a genetic database.

The "locus" field is for the gene symbol, preferably an official one (e.g. "Adh"). The "allele" field is for an allele symbol (e.g. "S"). The "desc" field is for a descriptive name for the gene (e.g. "Alcohol dehydrogenase, SLOW allele"). One should fill in as many of these fields as possible.

The "maploc" field accepts a string with a map location using whatever conventions are appropriate to the organism. This field is hardly definitive and if up to date mapping information is desired a true mapping database should always be consulted.

If "pseudo" is TRUE, this is a pseudogene.

The "db" field allows the Gene-ref to be attached to controlled identifiers from established gene databases. This allows a direct key to a database where gene information will be kept up to date without requiring that the rest of the information in the Gene-ref necessarily be up to date as well. This type of foreign key is essential to keeping loosely connected data up to date and NCBI is encouraging gene databases to make such controlled keys publicly available.

The "syn" field holds synonyms for the gene. It does not attempt to discriminate symbols, alleles, or descriptions.

Prot-ref: Reference To A Protein

A Prot-ref is meant to reference a protein very analogous to the way a Gene-ref references a gene. The "name" field is a SET OF strings to allow synonyms. The first name is presumed to be the preferred name by software tools. Since there is no controlled vocabulary for protein names this is the best that can be done at this time. "ADH" and "alcohol dehydrogenase" are both protein names.

The "desc" field is for a description of the protein. This field is often not necessary if the name field is filled in, but may be informative in some cases and essential in cases where the protein has not yet been named (e.g. ORF21 putative protein).

The "ec" field contains a SET of EC numbers. These strings are expected to be only numbers separated by periods (no leading "EC"). Sometimes the last few positions will be occupied by dashes or not filled in at all if the protein has not been fully characterized. Examples of EC numbers are (1.14.13.8 or 1.14.14.- or 1.14.14.3 or 1.14.--.-- or 1.14).

The "activity" field allows the various known activities of the protein to be specified. This can be very helpful, especially when the name is not informative.

The "db" field is to accommodate keys from protein databases. While protein nomenclature is not well controlled, there are subfields such as immunology which have controlled names. There are also databases which characterize proteins in other ways than sequence, such as 2-d spot databases which could provide such a key.

Txinit: Transcription Initiation

This is an example of a SeqFeatData block designed and built by a domain expert, an approach the NCBI strongly encourages and supports. The Txinit structure was developed by Philip Bucher and David Ghosh. It carries most of the information about transcription initiation represented in the Eukaryotic Promoter Database (EPD). The Txinit structure carries a host of detailed experimental information, far beyond the simple "promoter" features in GenBank/EMBL/DDBJ. EPD is released as a database in its own right and as Txinit Seq-feats. NCBI will be incorporating the EPD in its feature table form to provide expert annotation of the sequence databases in the manner described in the Data Model chapter.

The Txinit object is well described by its comments in the ASN.1 definition. The best source of more in depth discussion of these fields is in the EPD documentation, and so it will not be reproduced here.

Current Genetic Code Table: gc.prt

```
--**************************************************************************
-- This is the NCBI genetic code table
-- Base 1-3 of each codon have been added as comments to facilitate
-- readability at the suggestion of Peter Rice, EMBL
--*************************************************************************
Genetic-code-table ::= {
{
name "Standard" ,
name "SGC0" ,
id 1 ,
ncbieaa "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "-----------------------------------M----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Vertebrate Mitochondrial" ,
name "SGC1" ,
id 2 ,
ncbieaa "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIMMTTTTNNKKSS**VVVVAAAADDEEGGGG",
sncbieaa "--------------------------------MMMM---------------M------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Yeast Mitochondrial" ,
name "SGC2" ,
id 3 ,
ncbieaa "FFLLSSSSYY**CCWWTTTTPPPPHHQQRRRRIIMMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "-----------------------------------M----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Mold Mitochondrial and Mycoplasma" ,
name "SGC3" ,
id 4 ,
ncbieaa "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "-----------------------------------M----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Invertebrate Mitochondrial" ,
name "SGC4" ,
id 5 ,
ncbieaa "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIMMTTTTNNKKSSSSVVVVAAAADDEEGGGG",
sncbieaa "---M----------------------------M-MM----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Ciliate Macronuclear and Daycladacean" ,
name "SGC5" ,
id 6 ,
ncbieaa "FFLLSSSSYYQQCC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "-----------------------------------M----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Protozoan Mitochondrial (and Kinetoplast)" ,
name "SGC6" ,
id 7 ,
ncbieaa "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "--MM---------------M------------MMMM---------------M------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Plant Mitochondrial/Chloroplast (posttranscriptional variant)" ,
name "SGC7" ,
id 8 ,
ncbieaa "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRWIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "--M-----------------------------MMMM---------------M------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Echinoderm Mitochondrial" ,
name "SGC8" ,
id 9 ,
ncbieaa "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIIMTTTTNNNKSSSSVVVVAAAADDEEGGGG",
sncbieaa "-----------------------------------M----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Euplotid Macronuclear" ,
name "SGC9" ,
id 10 ,
ncbieaa "FFLLSSSSYY*QCCCWLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "-----------------------------------M----------------------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
} ,
{
name "Eubacterial" ,
id 11 ,
ncbieaa "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "---M---------------M------------M--M---------------M------------"
-- Base1 TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2 TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3 TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
}
}
```

Sequence Alignments

Sequence Alignments

Introduction

A sequence alignment is a mapping of the coordinates of one Bioseq onto the coordinates of one or more other Bioseqs. Such a mapping may be associated with a score and/or a method for doing the alignment. An alignment can be generated algorithmically by software or manually by a scientist. The Seq-align object is designed to capture the final result of the process, not the process itself.

A Seq-align is one of the forms of Seq-annot and is as acceptable a sequence annotation as a feature table. Seq-aligns would normally be "packaged" in a Seq-annot for exchange with other tools or databases so the alignments can be identified and given a title.

The most common sequence alignment is from one sequence to another with a one to one relationship between the aligned residues of one sequence with the residues of the other (with allowance for gaps). Two types of Seq-align types, Dense-seg and Dense-diag are specifically for this type of alignment. The Std-seg, on the other hand, is very generic and does not assume that the length of one aligned region is necessarily the same as the other. This permits expansion and contraction of one Bioseq relative to another, which is necessary in the case of a physical map Bioseq aligned to a genetic map Bioseq, or a sequence Bioseq aligned with any map Bioseq.

All the forms of Seq-align are composed of segments. Each segment is an aligned region which contains only sequence or only a gap for any sequence in the alignment. Below is a three dimensional alignment with six segments:

```
Seq-ids
id=100 AAGGCCTTTTAGAGATGATGATGATGATGA
id=200 AAGGCCTaTTAG.......GATGATGATGA
id=300 ....CCTTTTAGAGATGATGAT....ATGA
| 1 | 2 | 3 |4| 5 | 6| Segments
```

Taking only two of the sequences in a two way alignment, only three segments are needed to define the alignment:

```
Seq-ids
id=100 AAGGCCTTTTAGAGATGATGATGATGATGA
id=200 AAGGCCTaTTAG.......GATGATGATGA
| 1 | 2 | 3 | Segments
```

Seq-align

A Seq-align is a collection of segments representing one complete alignment. The whole Seq-align may have a Score representing some measure of quality or attributing the method used to build the Seq-align. In addition, each segment may have a score for that segment alone.

type: global

A global alignment is the alignment of Bioseqs over their complete length. It expresses the relationship between the intact Bioseqs. As such it is typically used in studies of homology between closely related proteins or genomes where there is reason to believe they share a common origin over their complete lengths.

The segments making up a global alignment are assumed to be connected in order from first to last to make up the alignment, and that the full lengths of all sequences will be accounted for in the alignment.

type: partial

A partial alignment only defines a relationship between sequences for the lengths actually included in the alignment. No claim is made that the relationship pertains to the full lengths of any of the sequences.

Like a global alignment, the segments making up a partial alignment are assumed to be connected in order from first to last to make up the alignment. Unlike a global alignment, it is not assumed the alignment will necessarily account for the full lengths of any or all sequences.

A partial or global alignment may use either the denseg choice of segment (for aligned Bioseqs with one to one residue mappings, such as protein or nucleic acid sequences) or the std choice for any Bioseqs including maps. In both cases there is an ordered relationship between one segment and the next to make the complete alignment.

type: diags

A Seq-align of type diags means that each segment is independent of the next and no claims are made about the reasonableness of connecting one segment to another. This is the kind of relationship shown by a "dot matrix" display. A series of diagonal lines in a square matrix indicate unbroken regions of similarity between the sequences. However, diagonals may overlap multiple times, or regions of the matrix may have no diagonals at all. The diags type of alignment captures that kind of relationship, although it is not limited to two dimensions as a dot matrix is.

The diags type of Seq-align may use either the dendiag choice of segment (for aligned Bioseqs with one to one residue mappings, such as protein or nucleic acid sequences) or the std choice for any Bioseqs including maps. In both cases the SEQUENCE OF does not imply any ordered relationship between one segment and the next. Each segment is independent of any other.

Type:disc

A discontinuous alignment is a set of alignments between two or more sequences. The alignments in the set represent the aligned chunks, broken by unaligned regions (represented by the implicit gaps in-between the alignments in the set).

Each chunk is a non-recursive Seq-align of type "global" or "partial" and with the same dimension. Seq-ids in all Seq-aligns are identical (and in the same order).

Examples of usage include mRNA-to-genomic alignments representing exons or genomic-to-genomic alignments containing unaligned regions.

dim: Dimensionality Of The Alignment

Most scientists are familiar with pairwise, or two dimensional, sequence alignments. However, it is often useful to align sequences in more dimensions. The dim attribute of Seq-align indicates the number of sequences which are **simultaneously** aligned. A three dimensional alignment is a true three way alignment (ABC), not three pairwise alignments (AB, AC, BC). Three pairwise alignments are three Seq-align objects, each with dimension equal to two.

Another common situation is when many sequences are aligned to one, as is the case of a merge of a number of components into a larger sequence, or the relationship of many mutant alleles to the wild type sequence. This is also a collection of two dimensional alignments, where one of the Bioseqs is common to all alignments. If the wild type Bioseq is A, and the mutants are B, C, D, then the Seq-annot would contain three two dimensional alignments, AB, AC, AD.

The dim attribute at the level of the Seq-align is **optional**, while the dim attribute is required on **each** segment. This is because it is convenient for a global or partial alignment to know the dimensionality for the whole alignment. It is also an integrity check that every segment in such a Seq-align has the same dimension. For diags however, the segments are independent of each other, and may even have different dimensions. This would be true for algorithms that locate the best n-way diagonals, where n can be 2 to the number of sequences. For a simple dot-matrix, all segments would be dimension two.

Score: Score Of An Alignment Or Segment

A Score contains an id (of type Object-id) which is meant to identify the method used to generate the score. It could be a string (e.g. "BLAST raw score", "BLAST p value") or an integer for use by a software system planning to process a number of defined values. The value of the Score is either an integer or real number. Both Seq-align and segment types allow more than one Score so that a variety of measures for the same alignment can be accommodated.

Dense-diag: Segments For diags Seq-align

A Seq-align of type diags represents a series of unconnected diagonals as a SEQUENCE OF Dense-diag. Since each Dense-diag is unrelated to the next the SEQUENCE OF just suggests a presentation order. It does not imply anything about the reasonableness of joining one Dense-diag to the next. In fact, for a multi-sequence comparison, each Dense-diag may have a different dimension and/or include Bioseqs not included by another Dense-diag.

A single Dense-diag defines its dimension with dim. There should be dim number of Seq-id in ids, indicating the Bioseqs involved in the segment, in order. There should be dim number of integers in starts (offsets into the Bioseqs, starting with 0, as in any Seq-loc) indicating the first (lowest numbered) residue of each Bioseq involved in the segment is, in the same order as ids. The len indicates the length of all Bioseqs in the segment. Thus the last residue involved in the segment for every Bioseq will be its start plus len - 1.

In the case of nucleic acids, if any or all of the segments are on the complement strand of the original Bioseq, then there should be dim number of Na-strand in len in the same order as ids, indicating which segments are on the plus or minus strands. The fact that a segment is on the minus strand or not does NOT affect the values chosen for starts. It is still the lowest numbered offset of a residue involved in the segment.

Clearly all Bioseq regions involved in a Dense-diag must have the same length, so this form does not allow stretching of one Bioseq compared to another, as may occur when comparing a genetic map Bioseq to a physical map or sequence Bioseq. In this case one would use Std-seg.

Dense-seg: Segments for "global" or "partial" Seq-align

A Dense-seg is a single entity which describes a complete global or partial alignment containing many segments. Like Dense-diag above, it is only appropriate when there is no stretching of the Bioseq coordinates relative to each other (as may happen when aligning a physical to a genetic map Bioseq). In that case, one would use a SEQUENCE OF Std-seg, described below.

A Dense-seg must give the dimension of the alignment in dim and the number of segments in the alignment in numseg. The ids slot must contain dim number of Seq-ids for the Bioseqs used in the alignment.

The starts slot contains the lowest numbered residue contained in each segment, in ids order. The starts slot should have numseg times dim integers, or the start of each Bioseq in the first segment in ids order, followed by the start of each Bioseq in the second segment in ids order and so on. A start of minus one indicates that the Bioseq is not present in the segment (i.e. a gap in a Bioseq).

The lens slot contains the length of each segment in segment order, so lens will contain numseg integers.

If any or all of the sequences are on the minus strand of the original Bioseq, then there should be numseg times dim Na-strand values in len in the same order as starts. Whether a sequence segment is on the plus or minus strand has **no** effect on the value selected for starts. It is **always** the lowest numbered residue included in the segment.

The scores is a SEQUENCE OF Score, one for each segment. So there should be numseg Scores, if scores is filled. A single Score for the whole alignment would appear in the score slot of the Seq-align.

The three dimensional alignment show above is repeated below, followed by its ASN.1 encoding into a Seq-align using Dense-seg. The Seq-ids are given in the ASN.1 as type "local".

```
 Seq-ids

 id=100 AAGGCCTTTTAGAGATGATGATGATGATGA
 id=200 AAGGCCTaTTAG.......GATGATGATGA
 id=300 ....CCTTTTAGAGATGATGAT....ATGA
 | 1 | 2 | 3 |4| 5 | 6| Segments

Seq-align ::= {
 type global ,
 dim 3 ,
 segs denseg {
 dim 3 ,
 numseg 6 ,
 ids {
 local id 100 ,
 local id 200 ,
 local id 300 } ,
 starts { 0,0,-1, 4,4,0, 12,-1,8, 19,12,15, 22,15,-1, 26,19,18 } ,
 lens { 4, 8, 7, 3, 4, 4 } } }
```

Std-seg: Aligning Any Bioseq Type With Any Other

A SEQUENCE OF Std-seg can be used to describe any Seq-align type on any types of Bioseqs. A Std-seg is very purely a collection of correlated Seq-locs. There is no requirement that the length of each Bioseq in a segment be the same as the other members of the segment or that the same Seq-loc type be used for each member of the segment. This allows stretching of one Bioseq relative to the other(s) and potentially very complex descriptions of relationships between sequences.

Each Std-seg must give its dimension, so it can be used for diags. Optionally it can give the Seq-ids for the Bioseqs used in the segment (again a convenience for Seq-align of type diags). The loc slot gives the locations on the Bioseqs used in this segment. As usual, there is also a place for various Score(s) associated with the segment. The example given above is presented again, this time as a Seq-align using Std-segs. Note the use of Seq-loc type "empty" to indicate a gap. Alternatively one could simply change the dim for each segment to exclude the Bioseqs not present in the segment, although this would require more interpretation by software.

```
 Seq-ids
 id=100 AAGGCCTTTTAGAGATGATGATGATGATGA
 id=200 AAGGCCTaTTAG.......GATGATGATGA
 id=300 ....CCTTTTAGAGATGATGAT....ATGA
 | 1 | 2 | 3 |4| 5 | 6| Segments
```

```
Seq-align ::= {
 type global ,
 dim 3 ,
 segs std {
 {
 dim 3 ,
 loc {
 int {
 id local id 100 ,
 from 0 ,
 to 3
 } ,
 int {
 id local id 200 ,
 from 0 ,
 to 3
 } ,
 empty local id 300
 }
 } ,
 {
 dim 3 ,
 loc {
 int {
 id local id 100 ,
 from 4 ,
 to 11
 } ,
 int {
 id local id 200 ,
 from 4 ,
 to 11
 } ,
 int {
 id local id 300 ,
 from 0 ,
 to 7
 }
 }
 } ,
 {
 dim 3 ,
 loc {
 int {
 id local id 100 ,
 from 12 ,
 to 18
 } ,
 empty local id 200 ,
 int {
 id local id 300 ,
 from 8 ,
 to 14
 }
 }
 } ,
 {
 dim 3 ,
 loc {
 int {
 id local id 100 ,
 from 19 ,
 to 21
 } ,
 int {
 id local id 200 ,
 from 12 ,
 to 14
 } ,
 int {
 id local id 300 ,
 from 15 ,
 to 17
 }
 }
 } ,
 {
 dim 3 ,
 loc {
 int {
 id local id 100 ,
 from 22 ,
 to 25
 } ,
 int {
 id local id 200 ,
 from 15 ,
 to 18
 } ,
 empty local id 300
 }
 } ,
 {
 dim 3 ,
 loc {
 int {
 id local id 100 ,
 from 26 ,
 to 29
 } ,
 int {
 id local id 200 ,
 from 19 ,
 to 22
 } ,
 int {
 id local id 300 ,
 from 18 ,
 to 21
 }
 }
 }
 }
}
```

Clearly the Std-seg method should only be used when its flexibility is required. Nonetheless, there is no ready substitute for Std-seg when flexibility is demanded.

C++ Implementation Notes

The C++ Toolkit adds several methods to the classes generated from ASN.1 specifications to simplify alignment data access and manipulation. The CSeq_align class has methods returning Seq-id, start, stop, and strand for a particular alignment row, regardless of its representation; it allows swapping alignment rows or converting the alignment from one type to another. The CDense_seg class extends the default set of alignment members with sequence character width (1 or 3, depending on molecule type).

Sequence Graphs

The Sequence Graphs section describes the Seq-graph type used to associate some analytical data with a region on a Bioseq. The type definition is located in the seqres.asn module.

Introduction

Analytical tools can attach results to Bioseqs in named collections as Seq-annots. This allows analytical programs developed from various sources to add information to a standard object (the Bioseq) and then let a single program designed for displaying a Bioseq and its associated information show the analytical results in an integrated fashion. Feature tables have been discussed previously, and can serve as the vehicle for results from restriction mapping programs, motif searching programs, open reading frame locators, and so on. Alignment programs and curator tools can produce Seq-annots containing Seq-aligns. In this chapter we present the third annotation type, a graph, which can be used to show properties like G+C content, surface potential, hydrophobicity, and so on.

Seq-graph: Graph on a Bioseq

A Seq-graph defines some continuous set of values over a defined interval on a Bioseq. It has slots for a title and a comment. The "loc" field defines the region of the Bioseq to which the graph applies. Titles can be given for the X (graph value) axis and/or the Y (sequence axis) of the graph. The "comp" slot allows a compression to supplied (i.e. how many residues are represented by a single value of the graph?). Compression is assumed to be one otherwise. Scaling values, a and b can be used to scale the values given in the Seq-graph to those displayed on the graph (by the formula "display value" = (a times "graph value") plus b). Finally, the number of values in the graph must be given (and should agree with the length of "loc" divided by "comp").

The graphs themselves can be coded as byte, integer, or real values. Each type defines the maximum and minimum values to show on the graph (no given values need necessarily reach the maximum or minimum) and the value along which to draw the X axis of the graph.

Common ASN.1 Specifications

Following are the ASN.1 specifications referenced in this chapter:

- general.asn
- biblio.asn
- pub.asn
- medline.asn
- seq.asn
- seqblock.asn
- seqcode.asn
- seqset.asn
- seqloc.asn
- seqfeat.asn
- seqalign.asn
- seqres.asn

ASN.1 Specification: general.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI General Data elements
-- by James Ostell, 1990
-- Version 3.0 - June 1994
--
--**********************************************************************

NCBI-General DEFINITIONS ::=
BEGIN

EXPORTS Date, Person-id, Object-id, Dbtag, Int-fuzz, User-object, User-field;

-- StringStore is really a VisibleString. It is used to define very
-- long strings which may need to be stored by the receiving program
-- in special structures, such as a ByteStore, but it's just a hint.
-- AsnTool stores StringStores in ByteStore structures.
-- OCTET STRINGs are also stored in ByteStores by AsnTool
--
-- typedef struct bsunit { /* for building multiline strings */
 -- Nlm_Handle str; /* the string piece */
 -- Nlm_Int2 len_avail,
 -- len;
 -- struct bsunit PNTR next; } /* the next one */
-- Nlm_BSUnit, PNTR Nlm_BSUnitPtr;
--
-- typedef struct bytestore {
 -- Nlm_Int4 seekptr, /* current position */
 -- totlen, /* total stored data length in bytes */
 -- chain_offset; /* offset in ByteStore of first byte in curchain */
 -- Nlm_BSUnitPtr chain, /* chain of elements */
 -- curchain; /* the BSUnit containing seekptr */
-- } Nlm_ByteStore, PNTR Nlm_ByteStorePtr;
--
-- AsnTool incorporates this as a primitive type, so the definition
-- is here just for completeness
--
-- StringStore ::= [APPLICATION 1] IMPLICIT OCTET STRING
--

-- BigInt is really an INTEGER. It is used to warn the receiving code to
expect
-- a value bigger than Int4 (actually Int8). It will be stored in
DataVal.bigintvalue
--
-- Like StringStore, AsnTool incorporates it as a primitive. The definition
would be:
-- BigInt ::= [APPLICATION 2] IMPLICIT INTEGER
--

-- Date is used to replace the (overly complex) UTCTtime, GeneralizedTime
-- of ASN.1
-- It stores only a date
--

Date ::= CHOICE {
 str VisibleString , -- for those unparsed dates
 std Date-std } -- use this if you can

Date-std ::= SEQUENCE { -- NOTE: this is NOT a unix tm struct
 year INTEGER , -- full year (including 1900)
 month INTEGER OPTIONAL , -- month (1-12)
 day INTEGER OPTIONAL , -- day of month (1-31)
 season VisibleString OPTIONAL , -- for "spring", "may-june", etc
 hour INTEGER OPTIONAL , -- hour of day (0-23)
 minute INTEGER OPTIONAL , -- minute of hour (0-59)
 second INTEGER OPTIONAL } -- second of minute (0-59)

-- Dbtag is generalized for tagging
-- eg. { "Social Security", str "023-79-8841" }
-- or { "member", id 8882224 }
```

```
Dbtag ::= SEQUENCE {
 db VisibleString , -- name of database or system
 tag Object-id } -- appropriate tag

-- Object-id can tag or name anything
--

Object-id ::= CHOICE {
 id INTEGER ,
 str VisibleString }

-- Person-id is to define a std element for people
--

Person-id ::= CHOICE {
 dbtag Dbtag , -- any defined database tag
 name Name-std , -- structured name
 ml VisibleString , -- MEDLINE name (semi-structured)
 -- eg. "Jones RM"
 str VisibleString, -- unstructured name
 consortium VisibleString } -- consortium name

Name-std ::= SEQUENCE { -- Structured names
 last VisibleString ,
 first VisibleString OPTIONAL ,
 middle VisibleString OPTIONAL ,
 full VisibleString OPTIONAL , -- full name eg. "J. John Smith, Esq"
 initials VisibleString OPTIONAL, -- first + middle initials
 suffix VisibleString OPTIONAL , -- Jr, Sr, III
 title VisibleString OPTIONAL } -- Dr., Sister, etc

--**** Int-fuzz **********************************************
--*
--* uncertainties in integer values

Int-fuzz ::= CHOICE {
 p-m INTEGER , -- plus or minus fixed amount
 range SEQUENCE { -- max to min
 max INTEGER ,
 min INTEGER } ,
 pct INTEGER , -- % plus or minus (x10) 0-1000
 lim ENUMERATED { -- some limit value
 unk (0) , -- unknown
 gt (1) , -- greater than
 lt (2) , -- less than
 tr (3) , -- space to right of position
 tl (4) , -- space to left of position
 circle (5) , -- artificial break at origin of circle
 other (255) } , -- something else
 alt SET OF INTEGER } -- set of alternatives for the integer
```

```
--**** User-object **********************************************
--*
--* a general object for a user defined structured data item
--* used by Seq-feat and Seq-descr

User-object ::= SEQUENCE {
 class VisibleString OPTIONAL , -- endeavor which designed this object
 type Object-id , -- type of object within class
 data SEQUENCE OF User-field } -- the object itself

User-field ::= SEQUENCE {
 label Object-id , -- field label
 num INTEGER OPTIONAL , -- required for strs, ints, reals, oss
 data CHOICE { -- field contents
 str VisibleString ,
 int INTEGER ,
 real REAL ,
 bool BOOLEAN ,
 os OCTET STRING ,
 object User-object , -- for using other definitions
 strs SEQUENCE OF VisibleString ,
 ints SEQUENCE OF INTEGER ,
 reals SEQUENCE OF REAL ,
 oss SEQUENCE OF OCTET STRING ,
 fields SEQUENCE OF User-field ,
 objects SEQUENCE OF User-object } }



END
```

ASN.1 Specification: biblio.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--*****************************************************************
--
-- NCBI Bibliographic data elements
-- by James Ostell, 1990
--
-- Taken from the American National Standard for
-- Bibliographic References
-- ANSI Z39.29-1977
-- Version 3.0 - June 1994
-- PubMedId added in 1996
-- ArticleIds and eprint elements added in 1999
--
--*****************************************************************

NCBI-Biblio DEFINITIONS ::=
BEGIN

EXPORTS Cit-art, Cit-jour, Cit-book, Cit-pat, Cit-let, Id-pat, Cit-gen,
 Cit-proc, Cit-sub, Title, Author, PubMedId;

IMPORTS Person-id, Date, Dbtag FROM NCBI-General;

 -- Article Ids

ArticleId ::= CHOICE { -- can be many ids for an article
 pubmed PubMedId , -- see types below
 medline MedlineUID ,
 doi DOI ,
 pii PII ,
 pmcid PmcID ,
 pmcpid PmcPid ,
 pmpid PmPid ,
 other Dbtag } -- generic catch all

PubMedId ::= INTEGER -- Id from the PubMed database at NCBI
MedlineUID ::= INTEGER -- Id from MEDLINE
DOI ::= VisibleString -- Document Object Identifier
PII ::= VisibleString -- Controlled Publisher Identifier
PmcID ::= INTEGER -- PubMed Central Id
PmcPid ::= VisibleString -- Publisher Id supplied to PubMed Central
PmPid ::= VisibleString -- Publisher Id supplied to PubMed

ArticleIdSet ::= SET OF ArticleId

 -- Status Dates

PubStatus ::= INTEGER { -- points of publication
 received (1) , -- date manuscript received for review
 accepted (2) , -- accepted for publication
 epublish (3) , -- published electronically by publisher
 ppublish (4) , -- published in print by publisher
 revised (5) , -- article revised by publisher/author
 pmc (6) , -- article first appeared in PubMed Central
 pmcr (7) , -- article revision in PubMed Central
 pubmed (8) , -- article citation first appeared in PubMed
 pubmedr (9) , -- article citation revision in PubMed
 aheadofprint (10), -- epublish, but will be followed by print
 premedline (11), -- date into PreMedline status
 medline (12), -- date made a MEDLINE record
 other (255) }

PubStatusDate ::= SEQUENCE { -- done as a structure so fields can be added
 pubstatus PubStatus ,
 date Date } -- time may be added later

PubStatusDateSet ::= SET OF PubStatusDate
```

```
 -- Citation Types

Cit-art ::= SEQUENCE { -- article in journal or book
 title Title OPTIONAL , -- title of paper (ANSI requires)
 authors Auth-list OPTIONAL , -- authors (ANSI requires)
 from CHOICE { -- journal or book
 journal Cit-jour ,
 book Cit-book ,
 proc Cit-proc } ,
 ids ArticleIdSet OPTIONAL } -- lots of ids

Cit-jour ::= SEQUENCE { -- Journal citation
 title Title , -- title of journal
 imp Imprint }

Cit-book ::= SEQUENCE { -- Book citation
 title Title , -- Title of book
 coll Title OPTIONAL , -- part of a collection
 authors Auth-list, -- authors
 imp Imprint }

Cit-proc ::= SEQUENCE { -- Meeting proceedings
 book Cit-book , -- citation to meeting
 meet Meeting } -- time and location of meeting

 -- Patent number and date-issue were made optional in 1997 to
 -- support patent applications being issued from the USPTO
 -- Semantically a Cit-pat must have either a patent number or
 -- an application number (or both) to be valid

Cit-pat ::= SEQUENCE { -- patent citation
 title VisibleString ,
 authors Auth-list, -- author/inventor
 country VisibleString , -- Patent Document Country
 doc-type VisibleString , -- Patent Document Type
 number VisibleString OPTIONAL, -- Patent Document Number
 date-issue Date OPTIONAL, -- Patent Issue/Pub Date
 class SEQUENCE OF VisibleString OPTIONAL , -- Patent Doc Class Code
 app-number VisibleString OPTIONAL , -- Patent Doc Appl Number
 app-date Date OPTIONAL , -- Patent Appl File Date
 applicants Auth-list OPTIONAL , -- Applicants
 assignees Auth-list OPTIONAL , -- Assignees
 priority SEQUENCE OF Patent-priority OPTIONAL , -- Priorities
 abstract VisibleString OPTIONAL } -- abstract of patent

Patent-priority ::= SEQUENCE {
 country VisibleString , -- Patent country code
 number VisibleString , -- number assigned in that country
 date Date } -- date of application
```

```
Id-pat ::= SEQUENCE { -- just to identify a patent
 country VisibleString , -- Patent Document Country
 id CHOICE {
 number VisibleString , -- Patent Document Number
 app-number VisibleString } , -- Patent Doc Appl Number
 doc-type VisibleString OPTIONAL } -- Patent Doc Type

Cit-let ::= SEQUENCE { -- letter, thesis, or manuscript
 cit Cit-book , -- same fields as a book
 man-id VisibleString OPTIONAL , -- Manuscript identifier
 type ENUMERATED {
 manuscript (1) ,
 letter (2) ,
 thesis (3) } OPTIONAL }
 -- NOTE: this is just to cite a
 -- direct data submission, see NCBI-Submit
 -- for the form of a sequence submission
Cit-sub ::= SEQUENCE { -- citation for a direct submission
 authors Auth-list , -- not necessarily authors of the paper
 imp Imprint OPTIONAL , -- this only used to get date.. will go
 medium ENUMERATED { -- medium of submission
 paper (1) ,
 tape (2) ,
 floppy (3) ,
 email (4) ,
 other (255) } OPTIONAL ,
 date Date OPTIONAL , -- replaces imp, will become required
 descr VisibleString OPTIONAL } -- description of changes for public view

Cit-gen ::= SEQUENCE { -- NOT from ANSI, this is a catchall
 cit VisibleString OPTIONAL , -- anything, not parsable
 authors Auth-list OPTIONAL ,
 muid INTEGER OPTIONAL , -- medline uid
 journal Title OPTIONAL ,
 volume VisibleString OPTIONAL ,
 issue VisibleString OPTIONAL ,
 pages VisibleString OPTIONAL ,
 date Date OPTIONAL ,
 serial-number INTEGER OPTIONAL , -- for GenBank style references
 title VisibleString OPTIONAL , -- eg. cit="unpublished",title="title"
 pmid PubMedId OPTIONAL } -- PubMed Id


 -- Authorship Group
Auth-list ::= SEQUENCE {
 names CHOICE {
 std SEQUENCE OF Author , -- full citations
 ml SEQUENCE OF VisibleString , -- MEDLINE, semi-structured
 str SEQUENCE OF VisibleString } , -- free for all
 affil Affil OPTIONAL } -- author affiliation
```

```
Author ::= SEQUENCE {
 name Person-id , -- Author, Primary or Secondary
 level ENUMERATED {
 primary (1),
 secondary (2) } OPTIONAL ,
 role ENUMERATED { -- Author Role Indicator
 compiler (1),
 editor (2),
 patent-assignee (3),
 translator (4) } OPTIONAL ,
 affil Affil OPTIONAL ,
 is-corr BOOLEAN OPTIONAL } -- TRUE if corresponding author

Affil ::= CHOICE {
 str VisibleString , -- unparsed string
 std SEQUENCE { -- std representation
 affil VisibleString OPTIONAL , -- Author Affiliation, Name
 div VisibleString OPTIONAL , -- Author Affiliation, Division
 city VisibleString OPTIONAL , -- Author Affiliation, City
 sub VisibleString OPTIONAL , -- Author Affiliation, County Sub
 country VisibleString OPTIONAL , -- Author Affiliation, Country
 street VisibleString OPTIONAL , -- street address, not ANSI
 email VisibleString OPTIONAL ,
 fax VisibleString OPTIONAL ,
 phone VisibleString OPTIONAL ,
 postal-code VisibleString OPTIONAL }}

 -- Title Group
 -- Valid for = A = Analytic (Cit-art)
 -- J = Journals (Cit-jour)
 -- B = Book (Cit-book)
 -- Valid for:
Title ::= SET OF CHOICE {
 name VisibleString , -- Title, Anal,Coll,Mono AJB
 tsub VisibleString , -- Title, Subordinate A B
 trans VisibleString , -- Title, Translated AJB
 jta VisibleString , -- Title, Abbreviated J
 iso-jta VisibleString , -- specifically ISO jta J
 ml-jta VisibleString , -- specifically MEDLINE jta J
 coden VisibleString , -- a coden J
 issn VisibleString , -- ISSN J
 abr VisibleString , -- Title, Abbreviated B
 isbn VisibleString } -- ISBN B

Imprint ::= SEQUENCE { -- Imprint group
 date Date , -- date of publication
 volume VisibleString OPTIONAL ,
 issue VisibleString OPTIONAL ,
 pages VisibleString OPTIONAL ,
 section VisibleString OPTIONAL ,
 pub Affil OPTIONAL, -- publisher, required for book
 cprt Date OPTIONAL, -- copyright date, " " "
 part-sup VisibleString OPTIONAL , -- part/sup of volume
 language VisibleString DEFAULT "ENG" , -- put here for simplicity
 prepub ENUMERATED { -- for prepublication citations
 submitted (1) , -- submitted, not accepted
 in-press (2) , -- accepted, not published
 other (255) } OPTIONAL ,
 part-supi VisibleString OPTIONAL , -- part/sup on issue
 retract CitRetract OPTIONAL , -- retraction info
 pubstatus PubStatus OPTIONAL , -- current status of this publication
 history PubStatusDateSet OPTIONAL } -- dates for this record

CitRetract ::= SEQUENCE {
 type ENUMERATED { -- retraction of an entry
 retracted (1) , -- this citation retracted
 notice (2) , -- this citation is a retraction notice
 in-error (3) , -- an erratum was published about this
 erratum (4) } , -- this is a published erratum
 exp VisibleString OPTIONAL } -- citation and/or explanation

Meeting ::= SEQUENCE {
 number VisibleString ,
 date Date ,
 place Affil OPTIONAL }

END
```

ASN.1 Specification: pub.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- Publication common set
-- James Ostell, 1990
--
-- This is the base class definitions for Publications of all sorts
--
-- support for PubMedId added in 1996
--**********************************************************************

NCBI-Pub DEFINITIONS ::=
BEGIN

EXPORTS Pub, Pub-set, Pub-equiv;

IMPORTS Medline-entry FROM NCBI-Medline
 Cit-art, Cit-jour, Cit-book, Cit-proc, Cit-pat, Id-pat, Cit-gen,
 Cit-let, Cit-sub, PubMedId FROM NCBI-Biblio;

Pub ::= CHOICE {
 gen Cit-gen , -- general or generic unparsed
 sub Cit-sub , -- submission
 medline Medline-entry ,
 muid INTEGER , -- medline uid
 article Cit-art ,
 journal Cit-jour ,
 book Cit-book ,
 proc Cit-proc , -- proceedings of a meeting
 patent Cit-pat ,
 pat-id Id-pat , -- identify a patent
 man Cit-let , -- manuscript, thesis, or letter
 equiv Pub-equiv, -- to cite a variety of ways
 pmid PubMedId } -- PubMedId

Pub-equiv ::= SET OF Pub -- equivalent identifiers for same citation

Pub-set ::= CHOICE {
 pub SET OF Pub ,
 medline SET OF Medline-entry ,
 article SET OF Cit-art ,
 journal SET OF Cit-jour ,
 book SET OF Cit-book ,
 proc SET OF Cit-proc , -- proceedings of a meeting
 patent SET OF Cit-pat }

END
```

ASN.1 Specification: medline.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- MEDLINE data definitions
-- James Ostell, 1990
--
-- enhanced in 1996 to support PubMed records as well by simply adding
-- the PubMedId and making MedlineId optional
--
--**********************************************************************

NCBI-Medline DEFINITIONS ::=
BEGIN

EXPORTS Medline-entry, Medline-si;

IMPORTS Cit-art, PubMedId FROM NCBI-Biblio
 Date FROM NCBI-General;

 -- a MEDLINE or PubMed entry
Medline-entry ::= SEQUENCE {
 uid INTEGER OPTIONAL , -- MEDLINE UID, sometimes not yet available if from
PubMed
 em Date , -- Entry Month
 cit Cit-art , -- article citation
 abstract VisibleString OPTIONAL ,
 mesh SET OF Medline-mesh OPTIONAL ,
 substance SET OF Medline-rn OPTIONAL ,
 xref SET OF Medline-si OPTIONAL ,
 idnum SET OF VisibleString OPTIONAL , -- ID Number (grants, contracts)
 gene SET OF VisibleString OPTIONAL ,
 pmid PubMedId OPTIONAL , -- MEDLINE records may include the PubMedId
 pub-type SET OF VisibleString OPTIONAL, -- may show publication types
(review, etc)
 mlfield SET OF Medline-field OPTIONAL , -- additional Medline field types
 status INTEGER {
 publisher (1) , -- record as supplied by publisher
 premedline (2) , -- premedline record
 medline (3) } DEFAULT medline } -- regular medline record

Medline-mesh ::= SEQUENCE {
 mp BOOLEAN DEFAULT FALSE , -- TRUE if main point (*)
 term VisibleString , -- the MeSH term
 qual SET OF Medline-qual OPTIONAL } -- qualifiers

Medline-qual ::= SEQUENCE {
 mp BOOLEAN DEFAULT FALSE , -- TRUE if main point
 subh VisibleString } -- the subheading

Medline-rn ::= SEQUENCE { -- medline substance records
 type ENUMERATED { -- type of record
 nameonly (0) ,
 cas (1) , -- CAS number
 ec (2) } , -- EC number
 cit VisibleString OPTIONAL , -- CAS or EC number if present
 name VisibleString } -- name (always present)

Medline-si ::= SEQUENCE { -- medline cross reference records
 type ENUMERATED { -- type of xref
 ddbj (1) , -- DNA Data Bank of Japan
 carbbank (2) , -- Carbohydrate Structure Database
 embl (3) , -- EMBL Data Library
 hdb (4) , -- Hybridoma Data Bank
 genbank (5) , -- GenBank
 hgml (6) , -- Human Gene Map Library
 mim (7) , -- Mendelian Inheritance in Man
 msd (8) , -- Microbial Strains Database
 pdb (9) , -- Protein Data Bank (Brookhaven)
 pir (10) , -- Protein Identification Resource
 prfseqdb (11) , -- Protein Research Foundation (Japan)
 psd (12) , -- Protein Sequence Database (Japan)
 swissprot (13) , -- SwissProt
 gdb (14) } , -- Genome Data Base
 cit VisibleString OPTIONAL } -- the citation/accession number

Medline-field ::= SEQUENCE {
 type INTEGER { -- Keyed type
 other (0) , -- look in line code
 comment (1) , -- comment line
 erratum (2) } , -- retracted, corrected, etc
 str VisibleString , -- the text
 ids SEQUENCE OF DocRef OPTIONAL } -- pointers relevant to this text

DocRef ::= SEQUENCE { -- reference to a document
 type INTEGER {
 medline (1) ,
 pubmed (2) ,
 ncbigi (3) } ,
 uid INTEGER }

END
```

ASN.1 Specification: seq.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI Sequence elements
-- by James Ostell, 1990
-- Version 3.0 - June 1994
--
--**********************************************************************

NCBI-Sequence DEFINITIONS ::=
BEGIN

EXPORTS Annotdesc, Annot-descr, Bioseq, GIBB-mol, Heterogen, MolInfo,
 Numbering, Pubdesc, Seq-annot, Seq-data, Seqdesc, Seq-descr, Seq-ext,
 Seq-hist, Seq-inst, Seq-literal, Seqdesc, Delta-ext, Seq-gap;

IMPORTS Date, Int-fuzz, Dbtag, Object-id, User-object FROM NCBI-General
 Seq-align FROM NCBI-Seqalign
 Seq-feat FROM NCBI-Seqfeat
 Seq-graph FROM NCBI-Seqres
 Pub-equiv FROM NCBI-Pub
 Org-ref FROM NCBI-Organism
 BioSource FROM NCBI-BioSource
 Seq-id, Seq-loc FROM NCBI-Seqloc
 GB-block FROM GenBank-General
 PIR-block FROM PIR-General
 EMBL-block FROM EMBL-General
 SP-block FROM SP-General
 PRF-block FROM PRF-General
 PDB-block FROM PDB-General
 Seq-table FROM NCBI-SeqTable;

--*** Sequence ********************************
--*

Bioseq ::= SEQUENCE {
 id SET OF Seq-id , -- equivalent identifiers
 descr Seq-descr OPTIONAL , -- descriptors
 inst Seq-inst , -- the sequence data
 annot SET OF Seq-annot OPTIONAL }

--*** Descriptors *****************************
--*

Seq-descr ::= SET OF Seqdesc

Seqdesc ::= CHOICE {
 mol-type GIBB-mol , -- type of molecule
 modif SET OF GIBB-mod , -- modifiers
 method GIBB-method , -- sequencing method
 name VisibleString , -- a name for this sequence
 title VisibleString , -- a title for this sequence
 org Org-ref , -- if all from one organism
 comment VisibleString , -- a more extensive comment
 num Numbering , -- a numbering system
 maploc Dbtag , -- map location of this sequence
 pir PIR-block , -- PIR specific info
 genbank GB-block , -- GenBank specific info
 pub Pubdesc , -- a reference to the publication
 region VisibleString , -- overall region (globin locus)
 user User-object , -- user defined object
 sp SP-block , -- SWISSPROT specific info
 dbxref Dbtag , -- xref to other databases
 embl EMBL-block , -- EMBL specific information
 create-date Date , -- date entry first created/released
 update-date Date , -- date of last update
 prf PRF-block , -- PRF specific information
 pdb PDB-block , -- PDB specific information
 het Heterogen , -- cofactor, etc associated but not bound
 source BioSource , -- source of materials, includes Org-ref
 molinfo MolInfo } -- info on the molecule and techniques

--******* NOTE:
--* mol-type, modif, method, and org are consolidated and expanded
--* in Org-ref, BioSource, and MolInfo in this specification. They
--* will be removed in later specifications. Do not use them in the
--* the future. Instead expect the new structures.
--*
--***************************
```

```
--**********************************************************************
--
-- MolInfo gives information on the
-- classification of the type and quality of the sequence
--
-- WARNING: this will replace GIBB-mol, GIBB-mod, GIBB-method
--
--**********************************************************************

MolInfo ::= SEQUENCE {
 biomol INTEGER {
 unknown (0) ,
 genomic (1) ,
 pre-RNA (2) , -- precursor RNA of any sort really
 mRNA (3) ,
 rRNA (4) ,
 tRNA (5) ,
 snRNA (6) ,
 scRNA (7) ,
 peptide (8) ,
 other-genetic (9) , -- other genetic material
 genomic-mRNA (10) , -- reported a mix of genomic and cdna sequence
 cRNA (11) , -- viral RNA genome copy intermediate
 snoRNA (12) , -- small nucleolar RNA
 transcribed-RNA (13) , -- transcribed RNA other than existing classes
 ncRNA (14) ,
 tmRNA (15) ,
 other (255) } DEFAULT unknown ,
 tech INTEGER {
 unknown (0) ,
 standard (1) , -- standard sequencing
 est (2) , -- Expressed Sequence Tag
 sts (3) , -- Sequence Tagged Site
 survey (4) , -- one-pass genomic sequence
 genemap (5) , -- from genetic mapping techniques
 physmap (6) , -- from physical mapping techniques
 derived (7) , -- derived from other data, not a primary entity
 concept-trans (8) , -- conceptual translation
 seq-pept (9) , -- peptide was sequenced
 both (10) , -- concept transl. w/ partial pept. seq.
 seq-pept-overlap (11) , -- sequenced peptide, ordered by overlap
 seq-pept-homol (12) , -- sequenced peptide, ordered by homology
 concept-trans-a (13) , -- conceptual transl. supplied by author
 htgs-1 (14) , -- unordered High Throughput sequence contig
 htgs-2 (15) , -- ordered High Throughput sequence contig
 htgs-3 (16) , -- finished High Throughput sequence
 fli-cdna (17) , -- full length insert cDNA
 htgs-0 (18) , -- single genomic reads for coordination
 htc (19) , -- high throughput cDNA
 wgs (20) , -- whole genome shotgun sequencing
 barcode (21) , -- barcode of life project
 composite-wgs-htgs (22) , -- composite of WGS and HTGS
 tsa (23) , -- transcriptome shotgun assembly
 other (255) } -- use Source.techexp
 DEFAULT unknown ,
 techexp VisibleString OPTIONAL , -- explanation if tech not enough
 --
 -- Completeness is not indicated in most records. For genomes, assume
 -- the sequences are incomplete unless specifically marked as complete.
 -- For mRNAs, assume the ends are not known exactly unless marked as
 -- having the left or right end.
 --
 completeness INTEGER {
 unknown (0) ,
 complete (1) , -- complete biological entity
 partial (2) , -- partial but no details given
 no-left (3) , -- missing 5' or NH3 end
 no-right (4) , -- missing 3' or COOH end
 no-ends (5) , -- missing both ends
 has-left (6) , -- 5' or NH3 end present
 has-right (7) , -- 3' or COOH end present
 other (255) } DEFAULT unknown ,
 gbmoltype VisibleString OPTIONAL } -- identifies particular ncRNA


GIBB-mol ::= ENUMERATED { -- type of molecule represented
 unknown (0) ,
 genomic (1) ,
 pre-mRNA (2) , -- precursor RNA of any sort really
 mRNA (3) ,
 rRNA (4) ,
 tRNA (5) ,
 snRNA (6) ,
 scRNA (7) ,
 peptide (8) ,
 other-genetic (9) , -- other genetic material
 genomic-mRNA (10) , -- reported a mix of genomic and cdna sequence
 other (255) }

GIBB-mod ::= ENUMERATED { -- GenInfo Backbone modifiers
 dna (0) ,
 rna (1) ,
 extrachrom (2) ,
 plasmid (3) ,
 mitochondrial (4) ,
 chloroplast (5) ,
 kinetoplast (6) ,
 cyanelle (7) ,
 synthetic (8) ,
 recombinant (9) ,
 partial (10) ,
 complete (11) ,
 mutagen (12) , -- subject of mutagenesis ?
 natmut (13) , -- natural mutant ?
 transposon (14) ,
 insertion-seq (15) ,
 no-left (16) , -- missing left end (5' for na, NH2 for aa)
 no-right (17) , -- missing right end (3' or COOH)
 macronuclear (18) ,
 proviral (19) ,
 est (20) , -- expressed sequence tag
 sts (21) , -- sequence tagged site
 survey (22) , -- one pass survey sequence
 chromoplast (23) ,
 genemap (24) , -- is a genetic map
 restmap (25) , -- is an ordered restriction map
 physmap (26) , -- is a physical map (not ordered restriction map)
 other (255) }

GIBB-method ::= ENUMERATED { -- sequencing methods
 concept-trans (1) , -- conceptual translation
 seq-pept (2) , -- peptide was sequenced
 both (3) , -- concept transl. w/ partial pept. seq.
 seq-pept-overlap (4) , -- sequenced peptide, ordered by overlap
 seq-pept-homol (5) , -- sequenced peptide, ordered by homology
 concept-trans-a (6) , -- conceptual transl. supplied by author
 other (255) }

Numbering ::= CHOICE { -- any display numbering system
 cont Num-cont , -- continuous numbering
 enum Num-enum , -- enumerated names for residues
 ref Num-ref , -- by reference to another sequence
 real Num-real } -- supports mapping to a float system

Num-cont ::= SEQUENCE { -- continuous display numbering system
 refnum INTEGER DEFAULT 1, -- number assigned to first residue
 has-zero BOOLEAN DEFAULT FALSE , -- 0 used?
 ascending BOOLEAN DEFAULT TRUE } -- ascending numbers?

Num-enum ::= SEQUENCE { -- any tags to residues
 num INTEGER , -- number of tags to follow
 names SEQUENCE OF VisibleString } -- the tags

Num-ref ::= SEQUENCE { -- by reference to other sequences
 type ENUMERATED { -- type of reference
 not-set (0) ,
 sources (1) , -- by segmented or const seq sources
 aligns (2) } , -- by alignments given below
 aligns Seq-align OPTIONAL }

Num-real ::= SEQUENCE { -- mapping to floating point system
 a REAL , -- from an integer system used by Bioseq
 b REAL , -- position = (a * int_position) + b
 units VisibleString OPTIONAL }

Pubdesc ::= SEQUENCE { -- how sequence presented in pub
 pub Pub-equiv , -- the citation(s)
 name VisibleString OPTIONAL , -- name used in paper
 fig VisibleString OPTIONAL , -- figure in paper
 num Numbering OPTIONAL , -- numbering from paper
 numexc BOOLEAN OPTIONAL , -- numbering problem with paper
 poly-a BOOLEAN OPTIONAL , -- poly A tail indicated in figure?
 maploc VisibleString OPTIONAL , -- map location reported in paper
 seq-raw StringStore OPTIONAL , -- original sequence from paper
 align-group INTEGER OPTIONAL , -- this seq aligned with others in paper
 comment VisibleString OPTIONAL, -- any comment on this pub in context
 reftype INTEGER { -- type of reference in a GenBank record
 seq (0) , -- refers to sequence
 sites (1) , -- refers to unspecified features
 feats (2) , -- refers to specified features
 no-target (3) } -- nothing specified (EMBL)
 DEFAULT seq }

Heterogen ::= VisibleString -- cofactor, prosthetic group, inhibitor, etc

--*** Instances of sequences *******************************
--*

Seq-inst ::= SEQUENCE { -- the sequence data itself
 repr ENUMERATED { -- representation class
 not-set (0) , -- empty
 virtual (1) , -- no seq data
 raw (2) , -- continuous sequence
 seg (3) , -- segmented sequence
 const (4) , -- constructed sequence
 ref (5) , -- reference to another sequence
 consen (6) , -- consensus sequence or pattern
 map (7) , -- ordered map of any kind
 delta (8) , -- sequence made by changes (delta) to others
 other (255) } ,
 mol ENUMERATED { -- molecule class in living organism
 not-set (0) , -- > cdna = rna
 dna (1) ,
 rna (2) ,
 aa (3) ,
 na (4) , -- just a nucleic acid
 other (255) } ,
 length INTEGER OPTIONAL , -- length of sequence in residues
 fuzz Int-fuzz OPTIONAL , -- length uncertainty
 topology ENUMERATED { -- topology of molecule
 not-set (0) ,
 linear (1) ,
 circular (2) ,
 tandem (3) , -- some part of tandem repeat
 other (255) } DEFAULT linear ,
 strand ENUMERATED { -- strandedness in living organism
 not-set (0) ,
 ss (1) , -- single strand
 ds (2) , -- double strand
 mixed (3) ,
 other (255) } OPTIONAL , -- default ds for DNA, ss for RNA, pept
 seq-data Seq-data OPTIONAL , -- the sequence
 ext Seq-ext OPTIONAL , -- extensions for special types
 hist Seq-hist OPTIONAL } -- sequence history

--*** Sequence Extensions **********************************
--* for representing more complex types
--* const type uses Seq-hist.assembly

Seq-ext ::= CHOICE {
 seg Seg-ext , -- segmented sequences
 ref Ref-ext , -- hot link to another sequence (a view)
 map Map-ext , -- ordered map of markers
 delta Delta-ext }

Seg-ext ::= SEQUENCE OF Seq-loc

Ref-ext ::= Seq-loc

Map-ext ::= SEQUENCE OF Seq-feat

Delta-ext ::= SEQUENCE OF Delta-seq

Delta-seq ::= CHOICE {
 loc Seq-loc , -- point to a sequence
 literal Seq-literal } -- a piece of sequence

Seq-literal ::= SEQUENCE {
 length INTEGER , -- must give a length in residues
 fuzz Int-fuzz OPTIONAL , -- could be unsure
 seq-data Seq-data OPTIONAL } -- may have the data

--*** Sequence History Record ***********************************
--** assembly = records how seq was assembled from others
--** replaces = records sequences made obsolete by this one
--** replaced-by = this seq is made obsolete by another(s)

Seq-hist ::= SEQUENCE {
 assembly SET OF Seq-align OPTIONAL ,-- how was this assembled?
 replaces Seq-hist-rec OPTIONAL , -- seq makes these seqs obsolete
 replaced-by Seq-hist-rec OPTIONAL , -- these seqs make this one obsolete
 deleted CHOICE {
 bool BOOLEAN ,
 date Date } OPTIONAL }
```

```
Seq-hist-rec ::= SEQUENCE {
 date Date OPTIONAL ,
 ids SET OF Seq-id }

--*** Various internal sequence representations ************
--* all are controlled, fixed length forms

Seq-data ::= CHOICE { -- sequence representations
 iupacna IUPACna , -- IUPAC 1 letter nuc acid code
 iupacaa IUPACaa , -- IUPAC 1 letter amino acid code
 ncbi2na NCBI2na , -- 2 bit nucleic acid code
 ncbi4na NCBI4na , -- 4 bit nucleic acid code
 ncbi8na NCBI8na , -- 8 bit extended nucleic acid code
 ncbipna NCBIpna , -- nucleic acid probabilities
 ncbi8aa NCBI8aa , -- 8 bit extended amino acid codes
 ncbieaa NCBIeaa , -- extended ASCII 1 letter aa codes
 ncbipaa NCBIpaa , -- amino acid probabilities
 ncbistdaa NCBIstdaa, -- consecutive codes for std aas
 gap Seq-gap -- gap types
}

Seq-gap ::= SEQUENCE {
 type INTEGER {
 unknown(0),
 fragment(1),
 clone(2),
 short-arm(3),
 heterochromatin(4),
 centromere(5),
 telomere(6),
 repeat(7),
 contig(8),
 other(255)
 },
 linkage INTEGER {
 unlinked(0),
 linked(1),
 other(255)
 } OPTIONAL
}

IUPACna ::= StringStore -- IUPAC 1 letter codes, no spaces
IUPACaa ::= StringStore -- IUPAC 1 letter codes, no spaces
NCBI2na ::= OCTET STRING -- 00=A, 01=C, 10=G, 11=T
NCBI4na ::= OCTET STRING -- 1 bit each for agct
 -- 0001=A, 0010=C, 0100=G, 1000=T/U
 -- 0101=Purine, 1010=Pyrimidine, etc
NCBI8na ::= OCTET STRING -- for modified nucleic acids
NCBIpna ::= OCTET STRING -- 5 octets/base, prob for a,c,g,t,n
 -- probabilities are coded 0-255 = 0.0-1.0
```

```
NCBI8aa ::= OCTET STRING -- for modified amino acids
NCBIeaa ::= StringStore -- ASCII extended 1 letter aa codes
 -- IUPAC codes + U=selenocysteine
NCBIpaa ::= OCTET STRING -- 25 octets/aa, prob for IUPAC aas in order:
 -- A-Y,B,Z,X,(ter),anything
 -- probabilities are coded 0-255 = 0.0-1.0
NCBIstdaa ::= OCTET STRING -- codes 0-25, 1 per byte

--*** Sequence Annotation *************************************
--*

-- This is a replica of Textseq-id
-- This is specific for annotations, and exists to maintain a semantic
-- difference between IDs assigned to annotations and IDs assigned to
-- sequences
Textannot-id ::= SEQUENCE {
 name VisibleString OPTIONAL ,
 accession VisibleString OPTIONAL ,
 release VisibleString OPTIONAL ,
 version INTEGER OPTIONAL
}

Annot-id ::= CHOICE {
 local Object-id ,
 ncbi INTEGER ,
 general Dbtag,
 other Textannot-id
}

Annot-descr ::= SET OF Annotdesc

Annotdesc ::= CHOICE {
 name VisibleString , -- a short name for this collection
 title VisibleString , -- a title for this collection
 comment VisibleString , -- a more extensive comment
 pub Pubdesc , -- a reference to the publication
 user User-object , -- user defined object
 create-date Date , -- date entry first created/released
 update-date Date , -- date of last update
 src Seq-id , -- source sequence from which annot came
 align Align-def, -- definition of the SeqAligns
 region Seq-loc } -- all contents cover this region

Align-def ::= SEQUENCE {
 align-type INTEGER { -- class of align Seq-annot
 ref (1) , -- set of alignments to the same sequence
 alt (2) , -- set of alternate alignments of the same seqs
 blocks (3) , -- set of aligned blocks in the same seqs
 other (255) } ,
 ids SET OF Seq-id OPTIONAL } -- used for the one ref seqid for now
```

```
Seq-annot ::= SEQUENCE {
 id SET OF Annot-id OPTIONAL ,
 db INTEGER { -- source of annotation
 genbank (1) ,
 embl (2) ,
 ddbj (3) ,
 pir (4) ,
 sp (5) ,
 bbone (6) ,
 pdb (7) ,
 other (255) } OPTIONAL ,
 name VisibleString OPTIONAL ,-- source if "other" above
 desc Annot-descr OPTIONAL , -- used only for stand alone Seq-annots
 data CHOICE {
 ftable SET OF Seq-feat ,
 align SET OF Seq-align ,
 graph SET OF Seq-graph ,
 ids SET OF Seq-id , -- used for communication between tools
 locs SET OF Seq-loc , -- used for communication between tools
 seq-table Seq-table } } -- features in table form

END
```

ASN.1 Specification: seqblock.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- 1990 - J.Ostell
-- Version 3.0 - June 1994
--
--**********************************************************************
--**********************************************************************
--
-- EMBL specific data
-- This block of specifications was developed by Reiner Fuchs of EMBL
-- Updated by J.Ostell, 1994
--
--**********************************************************************

EMBL-General DEFINITIONS ::=
BEGIN

EXPORTS EMBL-dbname, EMBL-xref, EMBL-block;

IMPORTS Date, Object-id FROM NCBI-General;

EMBL-dbname ::= CHOICE {
 code ENUMERATED {
 embl(0),
 genbank(1),
 ddbj(2),
 geninfo(3),
 medline(4),
 swissprot(5),
 pir(6),
 pdb(7),
 epd(8),
 ecd(9),
 tfd(10),
 flybase(11),
 prosite(12),
 enzyme(13),
 mim(14),
 ecoseq(15),
 hiv(16) ,
 other (255) } ,
 name VisibleString }

EMBL-xref ::= SEQUENCE {
 dbname EMBL-dbname,
 id SEQUENCE OF Object-id }

EMBL-block ::= SEQUENCE {
 class ENUMERATED {
 not-set(0),
 standard(1),
 unannotated(2),
 other(255) } DEFAULT standard,
 div ENUMERATED {
 fun(0),
 inv(1),
 mam(2),
 org(3),
 phg(4),
 pln(5),
 pri(6),
 pro(7),
 rod(8),
 syn(9),
 una(10),
 vrl(11),
 vrt(12),
 pat(13),
 est(14),
 sts(15),
 other (255) } OPTIONAL,
 creation-date Date,
 update-date Date,
 extra-acc SEQUENCE OF VisibleString OPTIONAL,
 keywords SEQUENCE OF VisibleString OPTIONAL,
 xref SEQUENCE OF EMBL-xref OPTIONAL }

END

--**********************************************************************
--
-- SWISSPROT specific data
-- This block of specifications was developed by Mark Cavanaugh of
-- NCBI working with Amos Bairoch of SWISSPROT
--
--**********************************************************************

SP-General DEFINITIONS ::=
BEGIN

EXPORTS SP-block;

IMPORTS Date, Dbtag FROM NCBI-General
 Seq-id FROM NCBI-Seqloc;

SP-block ::= SEQUENCE { -- SWISSPROT specific descriptions
 class ENUMERATED {
 not-set (0) ,
 standard (1) , -- conforms to all SWISSPROT checks
 prelim (2) , -- only seq and biblio checked
 other (255) } ,
 extra-acc SET OF VisibleString OPTIONAL , -- old SWISSPROT ids
 imeth BOOLEAN DEFAULT FALSE , -- seq known to start with Met
 plasnm SET OF VisibleString OPTIONAL, -- plasmid names carrying gene
 seqref SET OF Seq-id OPTIONAL, -- xref to other sequences
 dbref SET OF Dbtag OPTIONAL , -- xref to non-sequence dbases
 keywords SET OF VisibleString OPTIONAL , -- keywords
 created Date OPTIONAL , -- creation date
 sequpd Date OPTIONAL , -- sequence update
 annotupd Date OPTIONAL } -- annotation update

END

--**********************************************************************
--
-- PIR specific data
-- This block of specifications was developed by Jim Ostell of
-- NCBI
--
--**********************************************************************

PIR-General DEFINITIONS ::=
BEGIN

EXPORTS PIR-block;
```

```
IMPORTS Seq-id FROM NCBI-Seqloc;

PIR-block ::= SEQUENCE { -- PIR specific descriptions
 had-punct BOOLEAN OPTIONAL , -- had punctuation in sequence ?
 host VisibleString OPTIONAL ,
 source VisibleString OPTIONAL , -- source line
 summary VisibleString OPTIONAL ,
 genetic VisibleString OPTIONAL ,
 includes VisibleString OPTIONAL ,
 placement VisibleString OPTIONAL ,
 superfamily VisibleString OPTIONAL ,
 keywords SEQUENCE OF VisibleString OPTIONAL ,
 cross-reference VisibleString OPTIONAL ,
 date VisibleString OPTIONAL ,
 seq-raw VisibleString OPTIONAL , -- seq with punctuation
 seqref SET OF Seq-id OPTIONAL } -- xref to other sequences

END

--**********************************************************************
--
-- GenBank specific data
-- This block of specifications was developed by Jim Ostell of
-- NCBI
--
--**********************************************************************

GenBank-General DEFINITIONS ::=
BEGIN

EXPORTS GB-block;

IMPORTS Date FROM NCBI-General;

GB-block ::= SEQUENCE { -- GenBank specific descriptions
 extra-accessions SEQUENCE OF VisibleString OPTIONAL ,
 source VisibleString OPTIONAL , -- source line
 keywords SEQUENCE OF VisibleString OPTIONAL ,
 origin VisibleString OPTIONAL,
 date VisibleString OPTIONAL , -- OBSOLETE old form Entry Date
 entry-date Date OPTIONAL , -- replaces date
 div VisibleString OPTIONAL , -- GenBank division
 taxonomy VisibleString OPTIONAL } -- continuation line of organism

END

--***********************************************************************
-- PRF specific definition
-- PRF is a protein sequence database crated and maintained by
-- Protein Research Foundation, Minoo-city, Osaka, Japan.
--
```

```
-- Written by A.Ogiwara, Inst.Chem.Res. (Dr.Kanehisa's Lab),
-- Kyoto Univ., Japan
--
--**********************************************************************

PRF-General DEFINITIONS ::=
BEGIN

EXPORTS PRF-block;

PRF-block ::= SEQUENCE {
 extra-src PRF-ExtraSrc OPTIONAL,
 keywords SEQUENCE OF VisibleString OPTIONAL
}

PRF-ExtraSrc ::= SEQUENCE {
 host VisibleString OPTIONAL,
 part VisibleString OPTIONAL,
 state VisibleString OPTIONAL,
 strain VisibleString OPTIONAL,
 taxon VisibleString OPTIONAL
}

END

--*********************************************************************
--
-- PDB specific data
-- This block of specifications was developed by Jim Ostell and
-- Steve Bryant of NCBI
--
--*********************************************************************

PDB-General DEFINITIONS ::=
BEGIN

EXPORTS PDB-block;

IMPORTS Date FROM NCBI-General;

PDB-block ::= SEQUENCE { -- PDB specific descriptions
 deposition Date , -- deposition date month,year
 class VisibleString ,
 compound SEQUENCE OF VisibleString ,
 source SEQUENCE OF VisibleString ,
 exp-method VisibleString OPTIONAL , -- present if NOT X-ray diffraction
 replace PDB-replace OPTIONAL } -- replacement history

PDB-replace ::= SEQUENCE {
 date Date ,
 ids SEQUENCE OF VisibleString } -- entry ids replace by this one
```

```
END
```

ASN.1 Specification: seqcode.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
-- **********************************************************************
--
-- These are code and conversion tables for NCBI sequence codes
-- ASN.1 for the sequences themselves are define in seq.asn
--
-- Seq-map-table and Seq-code-table REQUIRE that codes start with 0
-- and increase continuously. So IUPAC codes, which are upper case
-- letters will always have 65 0 cells before the codes begin. This
-- allows all codes to do indexed lookups for things
--
-- Valid names for code tables are:
-- IUPACna
-- IUPACaa
-- IUPACeaa
-- IUPACaa3 3 letter amino acid codes : parallels IUPACeaa
-- display only, not a data exchange type
-- NCBI2na
-- NCBI4na
-- NCBI8na
-- NCBI8aa
-- NCBIstdaa
-- probability types map to IUPAC types for display as characters

NCBI-SeqCode DEFINITIONS ::=
BEGIN

EXPORTS Seq-code-table, Seq-map-table, Seq-code-set;

Seq-code-type ::= ENUMERATED { -- sequence representations
 iupacna (1) , -- IUPAC 1 letter nuc acid code
 iupacaa (2) , -- IUPAC 1 letter amino acid code
 ncbi2na (3) , -- 2 bit nucleic acid code
 ncbi4na (4) , -- 4 bit nucleic acid code
 ncbi8na (5) , -- 8 bit extended nucleic acid code
 ncbipna (6) , -- nucleic acid probabilities
 ncbi8aa (7) , -- 8 bit extended amino acid codes
 ncbieaa (8) , -- extended ASCII 1 letter aa codes
 ncbipaa (9) , -- amino acid probabilities
 iupacaa3 (10) , -- 3 letter code only for display
 ncbistdaa (11) } -- consecutive codes for std aas, 0-25

Seq-map-table ::= SEQUENCE { -- for tables of sequence mappings
 from Seq-code-type , -- code to map from
 to Seq-code-type , -- code to map to
 num INTEGER , -- number of rows in table
 start-at INTEGER DEFAULT 0 , -- index offset of first element
 table SEQUENCE OF INTEGER } -- table of values, in from-to order

Seq-code-table ::= SEQUENCE { -- for names of coded values
 code Seq-code-type , -- name of code
 num INTEGER , -- number of rows in table
 one-letter BOOLEAN , -- symbol is ALWAYS 1 letter?
 start-at INTEGER DEFAULT 0 , -- index offset of first element
 table SEQUENCE OF
 SEQUENCE {
 symbol VisibleString , -- the printed symbol or letter
 name VisibleString } , -- an explanatory name or string
 comps SEQUENCE OF INTEGER OPTIONAL } -- pointers to complement nuc acid

Seq-code-set ::= SEQUENCE { -- for distribution
 codes SET OF Seq-code-table OPTIONAL ,
 maps SET OF Seq-map-table OPTIONAL }

END
```

ASN.1 Specification: seqset.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI Sequence Collections
-- by James Ostell, 1990
--
-- Version 3.0 - 1994
--
--**********************************************************************

NCBI-Seqset DEFINITIONS ::=
BEGIN

EXPORTS Bioseq-set, Seq-entry;

IMPORTS Bioseq, Seq-annot, Seq-descr FROM NCBI-Sequence
 Object-id, Dbtag, Date FROM NCBI-General;

--*** Sequence Collections ********************************
--*

Bioseq-set ::= SEQUENCE { -- just a collection
 id Object-id OPTIONAL ,
 coll Dbtag OPTIONAL , -- to identify a collection
 level INTEGER OPTIONAL , -- nesting level
 class ENUMERATED {
 not-set (0) ,
 nuc-prot (1) , -- nuc acid and coded proteins
 segset (2) , -- segmented sequence + parts
 conset (3) , -- constructed sequence + parts
 parts (4) , -- parts for 2 or 3
 gibb (5) , -- geninfo backbone
 gi (6) , -- geninfo
 genbank (7) , -- converted genbank
 pir (8) , -- converted pir
 pub-set (9) , -- all the seqs from a single publication
 equiv (10) , -- a set of equivalent maps or seqs
 swissprot (11) , -- converted SWISSPROT
 pdb-entry (12) , -- a complete PDB entry
 mut-set (13) , -- set of mutations
 pop-set (14) , -- population study
 phy-set (15) , -- phylogenetic study
 eco-set (16) , -- ecological sample study
 gen-prod-set (17) , -- genomic products, chrom+mRNA+protein
 wgs-set (18) , -- whole genome shotgun project
 named-annot (19) , -- named annotation set
 named-annot-prod (20) , -- with instantiated mRNA+protein
 read-set (21) , -- set from a single read
 paired-end-reads (22) , -- paired sequences within a read-set
 other (255) } DEFAULT not-set ,
 release VisibleString OPTIONAL ,
 date Date OPTIONAL ,
 descr Seq-descr OPTIONAL ,
 seq-set SEQUENCE OF Seq-entry ,
 annot SET OF Seq-annot OPTIONAL }

Seq-entry ::= CHOICE {
 seq Bioseq ,
 set Bioseq-set }

END
```

ASN.1 Specification: seqloc.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI Sequence location and identifier elements
-- by James Ostell, 1990
--
-- Version 3.0 - 1994
--
--**********************************************************************

NCBI-Seqloc DEFINITIONS ::=
BEGIN
```

```
EXPORTS Seq-id, Seq-loc, Seq-interval, Packed-seqint, Seq-point, Packed-
seqpnt,
 Na-strand, Giimport-id;

IMPORTS Object-id, Int-fuzz, Dbtag, Date FROM NCBI-General
 Id-pat FROM NCBI-Biblio
 Feat-id FROM NCBI-Seqfeat;

--*** Sequence identifiers ********************************
--*

Seq-id ::= CHOICE {
 local Object-id , -- local use
 gibbsq INTEGER , -- Geninfo backbone seqid
 gibbmt INTEGER , -- Geninfo backbone moltype
 giim Giimport-id , -- Geninfo import id
 genbank Textseq-id ,
 embl Textseq-id ,
 pir Textseq-id ,
 swissprot Textseq-id ,
 patent Patent-seq-id ,
 other Textseq-id , -- for historical reasons, 'other' = 'refseq'
 general Dbtag , -- for other databases
 gi INTEGER , -- GenInfo Integrated Database
 ddbj Textseq-id , -- DDBJ
 prf Textseq-id , -- PRF SEQDB
 pdb PDB-seq-id , -- PDB sequence
 tpg Textseq-id , -- Third Party Annot/Seq Genbank
 tpe Textseq-id , -- Third Party Annot/Seq EMBL
 tpd Textseq-id , -- Third Party Annot/Seq DDBJ
 gpipe Textseq-id , -- Internal NCBI genome pipeline processing ID
 named-annot-track Textseq-id -- Internal named annotation tracking ID
}

Seq-id-set ::= SET OF Seq-id


Patent-seq-id ::= SEQUENCE {
 seqid INTEGER , -- number of sequence in patent
 cit Id-pat } -- patent citation

Textseq-id ::= SEQUENCE {
 name VisibleString OPTIONAL ,
 accession VisibleString OPTIONAL ,
 release VisibleString OPTIONAL ,
 version INTEGER OPTIONAL }

Giimport-id ::= SEQUENCE {
 id INTEGER , -- the id to use here
 db VisibleString OPTIONAL , -- dbase used in
 release VisibleString OPTIONAL } -- the release
```

```
PDB-seq-id ::= SEQUENCE {
 mol PDB-mol-id , -- the molecule name
 chain INTEGER DEFAULT 32 , -- a single ASCII character, chain id
 rel Date OPTIONAL } -- release date, month and year

PDB-mol-id ::= VisibleString -- name of mol, 4 chars

--*** Sequence locations **********************************
--*

Seq-loc ::= CHOICE {
 null NULL , -- not placed
 empty Seq-id , -- to NULL one Seq-id in a collection
 whole Seq-id , -- whole sequence
 int Seq-interval , -- from to
 packed-int Packed-seqint ,
 pnt Seq-point ,
 packed-pnt Packed-seqpnt ,
 mix Seq-loc-mix ,
 equiv Seq-loc-equiv , -- equivalent sets of locations
 bond Seq-bond ,
 feat Feat-id } -- indirect, through a Seq-feat


Seq-interval ::= SEQUENCE {
 from INTEGER ,
 to INTEGER ,
 strand Na-strand OPTIONAL ,
 id Seq-id , -- WARNING: this used to be optional
 fuzz-from Int-fuzz OPTIONAL ,
 fuzz-to Int-fuzz OPTIONAL }

Packed-seqint ::= SEQUENCE OF Seq-interval

Seq-point ::= SEQUENCE {
 point INTEGER ,
 strand Na-strand OPTIONAL ,
 id Seq-id , -- WARNING: this used to be optional
 fuzz Int-fuzz OPTIONAL }

Packed-seqpnt ::= SEQUENCE {
 strand Na-strand OPTIONAL ,
 id Seq-id ,
 fuzz Int-fuzz OPTIONAL ,
 points SEQUENCE OF INTEGER }

Na-strand ::= ENUMERATED { -- strand of nucleic acid
 unknown (0) ,
 plus (1) ,
 minus (2) ,
 both (3) , -- in forward orientation
 both-rev (4) , -- in reverse orientation
 other (255) }

Seq-bond ::= SEQUENCE { -- bond between residues
 a Seq-point , -- connection to a least one residue
 b Seq-point OPTIONAL } -- other end may not be available

Seq-loc-mix ::= SEQUENCE OF Seq-loc -- this will hold anything

Seq-loc-equiv ::= SET OF Seq-loc -- for a set of equivalent locations

END
```

ASN.1 Specification: seqfeat.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI Sequence Feature elements
-- by James Ostell, 1990
-- Version 3.0 - June 1994
--
--**********************************************************************

NCBI-Seqfeat DEFINITIONS ::=
BEGIN

EXPORTS Seq-feat, Feat-id, Genetic-code;

IMPORTS Gene-ref FROM NCBI-Gene
 Prot-ref FROM NCBI-Protein
 Org-ref FROM NCBI-Organism
 Variation-ref FROM NCBI-Variation
 BioSource FROM NCBI-BioSource
 RNA-ref FROM NCBI-RNA
 Seq-loc, Giimport-id FROM NCBI-Seqloc
 Pubdesc, Numbering, Heterogen FROM NCBI-Sequence
 Rsite-ref FROM NCBI-Rsite
 Txinit FROM NCBI-TxInit
 Pub-set FROM NCBI-Pub
 Object-id, Dbtag, User-object FROM NCBI-General;

--*** Feature identifiers ********************************
--*

Feat-id ::= CHOICE {
 gibb INTEGER , -- geninfo backbone
 giim Giimport-id , -- geninfo import
 local Object-id , -- for local software use
 general Dbtag } -- for use by various databases

--*** Seq-feat *******************************************
--* sequence feature generalization

Seq-feat ::= SEQUENCE {
 id Feat-id OPTIONAL ,
 data SeqFeatData , -- the specific data
 partial BOOLEAN OPTIONAL , -- incomplete in some way?
 except BOOLEAN OPTIONAL , -- something funny about this?
 comment VisibleString OPTIONAL ,
 product Seq-loc OPTIONAL , -- product of process
 location Seq-loc , -- feature made from
 qual SEQUENCE OF Gb-qual OPTIONAL , -- qualifiers
 title VisibleString OPTIONAL , -- for user defined label
 ext User-object OPTIONAL , -- user defined structure extension
 cit Pub-set OPTIONAL , -- citations for this feature
 exp-ev ENUMERATED { -- evidence for existence of feature
 experimental (1) , -- any reasonable experimental check
 not-experimental (2) } OPTIONAL , -- similarity, pattern, etc
 xref SET OF SeqFeatXref OPTIONAL , -- cite other relevant features
 dbxref SET OF Dbtag OPTIONAL , -- support for xref to other databases
 pseudo BOOLEAN OPTIONAL , -- annotated on pseudogene?
 except-text VisibleString OPTIONAL , -- explain if except=TRUE
 ids SET OF Feat-id OPTIONAL , -- set of Ids; will replace 'id' field
 exts SET OF User-object OPTIONAL } -- set of extensions; will replace 'ext'
field

SeqFeatData ::= CHOICE {
 gene Gene-ref ,
 org Org-ref ,
 cdregion Cdregion ,
 prot Prot-ref ,
 rna RNA-ref ,
 pub Pubdesc , -- publication applies to this seq
 seq Seq-loc , -- to annotate origin from another seq
 imp Imp-feat ,
 region VisibleString, -- named region (globin locus)
 comment NULL , -- just a comment
 bond ENUMERATED {
 disulfide (1) ,
 thiolester (2) ,
 xlink (3) ,
 thioether (4) ,
 other (255) } ,
 site ENUMERATED {
 active (1) ,
 binding (2) ,
 cleavage (3) ,
 inhibit (4) ,
 modified (5),
 glycosylation (6) ,
 myristoylation (7) ,
 mutagenized (8) ,
 metal-binding (9) ,
 phosphorylation (10) ,
 acetylation (11) ,
 amidation (12) ,
 methylation (13) ,
 hydroxylation (14) ,
 sulfatation (15) ,
 oxidative-deamination (16) ,
 pyrrolidone-carboxylic-acid (17) ,
 gamma-carboxyglutamic-acid (18) ,
 blocked (19) ,
 lipid-binding (20) ,
 np-binding (21) ,
 dna-binding (22) ,
 signal-peptide (23) ,
 transit-peptide (24) ,
 transmembrane-region (25) ,
 nitrosylation (26) ,
 other (255) } ,
 rsite Rsite-ref , -- restriction site (for maps really)
 user User-object , -- user defined structure
 txinit Txinit , -- transcription initiation
 num Numbering , -- a numbering system
 psec-str ENUMERATED { -- protein secondary structure
 helix (1) , -- any helix
 sheet (2) , -- beta sheet
 turn (3) } , -- beta or gamma turn
 non-std-residue VisibleString , -- non-standard residue here in seq
 het Heterogen , -- cofactor, prosthetic grp, etc, bound to seq
 biosrc BioSource,
 clone Clone-ref,
 variation Variation-ref
}

SeqFeatXref ::= SEQUENCE { -- both optional because can have one or both
 id Feat-id OPTIONAL , -- the feature copied
 data SeqFeatData OPTIONAL } -- the specific data

--*** CdRegion ***********************************************
--*
--* Instructions to translate from a nucleic acid to a peptide
--* conflict means it's supposed to translate but doesn't
--*


Cdregion ::= SEQUENCE {
 orf BOOLEAN OPTIONAL , -- just an ORF ?
 frame ENUMERATED {
```

```
 not-set (0) , -- not set, code uses one
 one (1) ,
 two (2) ,
 three (3) } DEFAULT not-set , -- reading frame
 conflict BOOLEAN OPTIONAL , -- conflict
 gaps INTEGER OPTIONAL , -- number of gaps on conflict/except
 mismatch INTEGER OPTIONAL , -- number of mismatches on above
 code Genetic-code OPTIONAL , -- genetic code used
 code-break SEQUENCE OF Code-break OPTIONAL , -- individual exceptions
 stops INTEGER OPTIONAL } -- number of stop codons on above

 -- each code is 64 cells long, in the order where
 -- T=0,C=1,A=2,G=3, TTT=0, TTC=1, TCA=4, etc
 -- NOTE: this order does NOT correspond to a Seq-data
 -- encoding. It is "natural" to codon usage instead.
 -- the value in each cell is the AA coded for
 -- start= AA coded only if first in peptide
 -- in start array, if codon is not a legitimate start
 -- codon, that cell will have the "gap" symbol for
 -- that alphabet. Otherwise it will have the AA
 -- encoded when that codon is used at the start.

Genetic-code ::= SET OF CHOICE {
 name VisibleString , -- name of a code
 id INTEGER , -- id in dbase
 ncbieaa VisibleString , -- indexed to IUPAC extended
 ncbi8aa OCTET STRING , -- indexed to NCBI8aa
 ncbistdaa OCTET STRING , -- indexed to NCBIstdaa
 sncbieaa VisibleString , -- start, indexed to IUPAC extended
 sncbi8aa OCTET STRING , -- start, indexed to NCBI8aa
 sncbistdaa OCTET STRING } -- start, indexed to NCBIstdaa

Code-break ::= SEQUENCE { -- specific codon exceptions
 loc Seq-loc , -- location of exception
 aa CHOICE { -- the amino acid
 ncbieaa INTEGER , -- ASCII value of NCBIeaa code
 ncbi8aa INTEGER , -- NCBI8aa code
 ncbistdaa INTEGER } } -- NCBIstdaa code

Genetic-code-table ::= SET OF Genetic-code -- table of genetic codes

--*** Import ***********************************************
--*
--* Features imported from other databases
--*

Imp-feat ::= SEQUENCE {
 key VisibleString ,
 loc VisibleString OPTIONAL , -- original location string
 descr VisibleString OPTIONAL } -- text description
```

```
Gb-qual ::= SEQUENCE {
 qual VisibleString ,
 val VisibleString }


--*** Clone-ref ***********************************************
--*
--* Specification of clone features
--*

Clone-ref ::= SEQUENCE {
 name VisibleString, -- Official clone symbol
 library VisibleString OPTIONAL, -- Library name

 concordant BOOLEAN DEFAULT FALSE, -- OPTIONAL?
 unique BOOLEAN DEFAULT FALSE, -- OPTIONAL?
 placement-method INTEGER {
 end-seq (0), -- Clone placed by end sequence
 insert-alignment (1), -- Clone placed by insert alignment
 sts (2), -- Clone placed by STS
 fish (3),
 fingerprint (4),
 other (255)
 } OPTIONAL,
 clone-seq Clone-seq-set OPTIONAL
}

Clone-seq-set ::= SET OF Clone-seq


Clone-seq ::= SEQUENCE {
 type INTEGER {
 insert (0),
 end (1),
 other (255)
 },
 confidence INTEGER {
 multiple (0), -- Multiple hits
 na (1), -- Unspecified
 nohit-rep (2), -- No hits, repetitive
 nohitnorep (3), -- No hits, not repetitive
 other-chrm (4), -- Hit on different chromosome
 unique (5),
 virtual (6), -- Virtual (hasn't been sequenced)
 other (255)
 } OPTIONAL,
 location Seq-loc, -- location on sequence
 seq Seq-loc OPTIONAL, -- clone sequence location
 align-id Dbtag OPTIONAL
}
```

```
END


--*** Variation-ref ***********************************************
--*
--* Specification of variation features
--*

NCBI-Variation DEFINITIONS ::=
BEGIN

EXPORTS Variation-ref, Variation-inst;

IMPORTS Int-fuzz, User-object, Object-id, Dbtag FROM NCBI-General
 Seq-literal FROM NCBI-Sequence
 Seq-loc FROM NCBI-Seqloc
 Pub FROM NCBI-Pub;

-- ---------------------------------------------------------------------------
-- Historically, the dbSNP definitions document data structures used in the
-- processing and annotation of variations by the dbSNP group. The intention
-- is to provide information to clients that reflect internal information
-- produced during the mapping of SNPs
-- ---------------------------------------------------------------------------

VariantProperties ::= SEQUENCE {
 version INTEGER,

 -- NOTE:
 -- The format for each of these values is as an integer
 -- Unless otherwise noted, these integers represent a bitwise OR of the
 -- possible values, and as such, these values represent the specific bit
 -- flags that may be set for each of the possible attributes here.

 resource-link INTEGER {
 precious (1), -- Clinical, Pubmed, Cited, (0x01)
 provisional (2), -- Provisional Third Party Annotations (0x02)
 has3D (4), -- Has 3D strcture SNP3D table (0x04)
 submitterLinkout (8), -- SNP->SubSNP->Batch link_out (0x08)
 clinical (16), -- Clinical if LSDB, OMIM, TPA, Diagnostic
 genotypeKit (32) -- Marker exists on high density genotyping kit
 } OPTIONAL,

 gene-function INTEGER {
 no-change (0), -- known to cause no functional changes
 -- since 0 does not combine with any other bit
 -- value, 'no-change' specifically implies that
 -- there are no consequences
 in-gene (1), -- Sequence intervals covered by a gene ID but not
 -- having an aligned transcript (0x01)
 in-gene-5 (2), -- In Gene near 5' (0x02)
```

```
in-gene-3 (4), -- In Gene near 3' (0x04)
intron (8), -- In Intron (0x08)
donor (16), -- In donor splice-site (0x10)
acceptor (32), -- In acceptor splice-site (0x20)
utr-5 (64), -- In 5' UTR (0x40)
utr-3 (128), -- In 3' UTR (0x80)
synonymous (256), -- one allele in the set does not change the encoded
-- amino acid (0x100)
nonsense (512), -- one allele in the set changes to STOP codon
-- (TER). (0x200)
missense (1024), -- one allele in the set changes protein peptide
-- (0x400)
frameshift (2048), -- one allele in the set changes all downstream
-- amino acids (0x800)

in-start-codon(4096), -- the variant is observed in a start codon (0x1000)
up-regulator(8192), -- the variant causes increased transcription
-- (0x2000)
down-regulator(16384) -- the variant causes decreased transcription
-- (0x4000)
} OPTIONAL,

mapping INTEGER {
has-other-snp (1), -- Another SNP has the same mapped positions
-- on reference assembly (0x01)
has-assembly-conflict (2), -- Weight 1 or 2 SNPs that map to different
-- chromosomes on different assemblies (0x02)
is-assembly-specific (4) -- Only maps to 1 assembly (0x04)
} OPTIONAL,

-- This is *NOT* a bitfield
weight INTEGER {
is-uniquely-placed(1),
placed-twice-on-same-chrom(2),
placed-twice-on-diff-chrom(3),
many-placements(10)
} OPTIONAL,

allele-freq INTEGER {
is-mutation (1), -- low frequency variation that is cited in journal
-- and other reputable sources (0x01)
above-5pct-all (2), -- >5% minor allele freq in each and all
-- populations (0x02)
above-5pct-1plus (4), -- >5% minor allele freq in 1+ populations (0x04)
validated (8) -- Bit is set if the variant has 2+ minor allele
-- count based on freq or genotype data
} OPTIONAL,

genotype INTEGER {
in-haplotype-set (1), -- Exists in a haplotype tagging set (0x01)
has-genotypes (2) -- SNP has individual genotype (0x02)
```

```
 } OPTIONAL,

 hapmap INTEGER {
 phase1-genotyped (1), -- Phase 1 genotyped; filtered, non-redundant
 -- (0x01)
 phase2-genotyped (2), -- Phase 2 genotyped; filtered, non-redundant
 -- (0x02)
 phase3-genotyped (4) -- Phase 3 genotyped; filtered, non-redundant
 -- (0x04)
 } OPTIONAL,

 quality-check INTEGER {
 contig-allele-missing (1), -- Reference sequence allele at the mapped
 -- position is not present in the SNP
 -- allele list, adjusted for orientation
 -- (0x01)
 withdrawn-by-submitter (2), -- One member SS is withdrawn by submitter
 -- (0x02)
 non-overlapping-alleles (4), -- RS set has 2+ alleles from different
 -- submissions and these sets share no
 -- alleles in common (0x04)
 strain-specific (8), -- Straing specific fixed difference (0x08)
 genotype-conflict (16) -- Has Genotype Conflict (0x10)
 } OPTIONAL
}

Phenotype ::= SEQUENCE {
 source VisibleString OPTIONAL,
 term VisibleString OPTIONAL,
 xref SET OF Dbtag OPTIONAL,

 -- does this variant have known clinical significance?
 clinical-significance INTEGER {
 unknown (0),
 untested (1),
 non-pathogenic (2),
 probable-non-pathogenic (3),
 probable-pathogenic (4),
 pathogenic (5),
 other (255)
 } OPTIONAL
}

Population-data ::= SEQUENCE {
 -- assayed population (e.g. HAPMAP-CEU)
 population VisibleString,
 genotype-frequency REAL OPTIONAL,
 chromosomes-tested INTEGER OPTIONAL,
 sample-ids SET OF Object-id OPTIONAL
}
```

```
Ext-loc ::= SEQUENCE {
 id Object-id,
 location Seq-loc
}

Variation-ref ::= SEQUENCE {
 -- ids (i.e., SNP rsid / ssid, dbVar nsv/nssv)
 -- expected values include 'dbSNP|rs12334', 'dbSNP|ss12345', 'dbVar|nsv1'
 --
 -- we relate three kinds of IDs here:
 -- - our current object's id
 -- - the id of this object's parent, if it exists
 -- - the sample ID that this item originates from
 id Dbtag OPTIONAL,
 parent-id Dbtag OPTIONAL,
 sample-id Object-id OPTIONAL,
 other-ids SET OF Dbtag OPTIONAL,

 -- names and synonyms
 -- some variants have well-known canonical names and possible accepted
 -- synonyms
 name VisibleString OPTIONAL,
 synonyms SET OF VisibleString OPTIONAL,

 -- tag for comment and descriptions
 description VisibleString OPTIONAL,

 -- phenotype
 phenotype SET OF Phenotype OPTIONAL,

 -- sequencing / acuisition method
 method SET OF INTEGER {
 unknown (0),
 bac-acgh (1),
 computational (2),
 curated (3),
 digital-array (4),
 expression-array (5),
 fish (6),
 flanking-sequence (7),
 maph (8),
 mcd-analysis (9),
 mlpa (10),
 oea-assembly (11),
 oligo-acgh (12),
 paired-end (13),
 pcr (14),
 qpcr (15),
 read-depth (16),
 roma (17),
 rt-pcr (18),
 sage (19),
 sequence-alignment (20),
 sequencing (21),
 snp-array (22),
 snp-genoytyping (23),
 southern (24),
 western (25),

 other (255)
 } OPTIONAL,

 -- Note about SNP representation and pretinent fields: allele-frequency,
 -- population, quality-codes:
 -- The case of multiple alleles for a SNP would be described by
 -- parent-feature of type Variation-set.diff-alleles, where the child
 -- features of type Variation-inst, all at the same location, would
 -- describe individual alleles.

 -- population data
 population-data SET OF Population-data OPTIONAL,

 -- variant properties bit fields
 variant-prop VariantProperties OPTIONAL,

 -- has this variant been validated?
 validated BOOLEAN OPTIONAL,

 -- link-outs to GeneTests database
 clinical-test SET OF Dbtag OPTIONAL,

 -- origin of this allele, if known
 allele-origin INTEGER {
 unknown (0),
 germline (1),
 somatic (2),
 inherited (3),
 paternal (4),
 maternal (5),
 de-novo (6),
 biparental (7),
 uniparental (8),
 not-tested (9),
 tested-inconclusive (10),

 other (255)
 } OPTIONAL,

 -- observed allele state, if known
 allele-state INTEGER {
 unknown (0),
 homozygous (1),
 heterozygous (2),
 hemizygous (3),
 nullizygous (4),
 other (255)
 } OPTIONAL,

 allele-frequency REAL OPTIONAL,

 -- is this variant the ancestral allele?
 is-ancestral-allele BOOLEAN OPTIONAL,

 -- publication support.
 -- Note: made this pub instead of pub-equiv, since
 -- Pub can be pub-equiv and pub-equiv is a set of pubs, but it looks like
 -- Pub is more often used as top-level container
 pub Pub OPTIONAL,

 data CHOICE {
 unknown NULL,
 note VisibleString, --free-form
 uniparental-disomy NULL,

 -- actual sequence-edit at feat.location
 instance Variation-inst,

 -- Set of related Variations.
 -- Location of the set equals to the union of member locations
 set SEQUENCE {
 type INTEGER {
 unknown (0),
 compound (1), -- complex change at the same location on the
 -- same molecule
 products (2), -- different products arising from the same
 -- variation in a precursor, e.g. r.[13g>a,
 -- 13_88del]
 haplotype (3), -- changes on the same allele, e.g
 -- r.[13g>a;15u>c]
 alleles (4), -- changes on different alleles in the same
 -- genotype, e.g. g.[476C>T]+[476C>T]
 mosaic (5), -- different genotypes in the same individual
 individual (6), -- same organism; allele relationship unknown,
 -- e.g. g.[476C>T(+)183G>C]
 population (7), -- population
 other (255)
 },
 variations SET OF Variation-ref,
 name VisibleString OPTIONAL
 }
 },

 consequence SET OF CHOICE {
 unknown NULL,
 splicing NULL, --some effect on splicing
 note VisibleString, --freeform

 -- Describe resulting variation in the product, e.g. missense,
 -- nonsense, silent, neutral, etc in a protein, that arises from
 -- THIS variation.
 variation Variation-ref,

 -- see http://www.hgvs.org/mutnomen/recs-prot.html
 frameshift SEQUENCE {
 phase INTEGER OPTIONAL,
 x-length INTEGER OPTIONAL
 },

 loss-of-heterozygosity SEQUENCE {
 -- In germline comparison, it will be reference genome assembly
 -- (default) or reference/normal population. In somatic mutation,
 -- it will be a name of the normal tissue.
 reference VisibleString OPTIONAL,

 -- Name of the testing subject type or the testing tissue.
 test VisibleString OPTIONAL
 }
 } OPTIONAL,

 -- Observed location, if different from the parent set or feature.location.
 location Seq-loc OPTIONAL,

 -- reference other locs, e.g. mapped source
 ext-locs SET OF Ext-loc OPTIONAL,

 ext User-object OPTIONAL

}

Delta-item ::= SEQUENCE {
 seq CHOICE {
 literal Seq-literal,
 loc Seq-loc,
 this NULL --same location as variation-ref itself
 },

 -- Multiplier allows representing a tandem, e.g. ATATAT as AT*3
 -- This allows describing CNV/SSR where delta=self with a
 -- multiplier which specifies the count of the repeat unit.

 multiplier INTEGER OPTIONAL, --assumed 1 if not specified.
 multiplier-fuzz Int-fuzz OPTIONAL
}
```

```
-- Variation instance
Variation-inst ::= SEQUENCE {
 type INTEGER {
 unknown (0),
 identity (1), -- delta = this
 inv (2), -- inversion: delta =
 -- reverse-comp(feat.location)
 snp (3), -- delins where len(del) = len(ins) = 1
 mnp (4), -- delins where len(del) = len(ins) > 1
 delins (5), -- delins where len(del) != len(ins)
 del (6), -- deltaseq is empty
 ins (7), -- deltaseq contains [this, ins] or [ins, this]
 microsatellite (8), -- location describes tandem sequence;
 -- delta is the repeat-unit with a multiplier
 transposon (9), -- delta refers to equivalent sequence in
 -- another location.
 -- ext-loc describes donor sequence, if known
 -- (could be location itself)
 cnv (10), -- general CNV class, indicating "local"
 -- rearrangement

 -- Below are four possible copy configurations,
 -- where delta is a seq-loc on the same sequence.
 -- If the repeat is represented on the sequence, it is
 -- described like a transposon; if it is a de-novo
 -- repeat, it is described like an insertion.
 direct-copy (11), -- delta sequence is located upstream, same
 -- strand
 rev-direct-copy (12), -- delta sequence is downstream, same strand
 inverted-copy (13), -- delta sequence is upstream, opposite strand
 everted-copy (14), -- delta sequence is downstream, opposite strand

 translocation (15), -- feat.location is swapped with delta (i.e.
 -- reciprocal transposon)
 prot-missense (16),
 prot-nonsense (17),
 prot-neutral (18),
 prot-silent (19),
 prot-other (20),

 other (255)
 },

 -- Sequence that replaces the location, in biological order.
 delta SEQUENCE OF Delta-item
}

END


--**********************************************************************
```

```
--
-- NCBI Restriction Sites
-- by James Ostell, 1990
-- version 0.8
--
--**********************************************************************

NCBI-Rsite DEFINITIONS ::=
BEGIN

EXPORTS Rsite-ref;

IMPORTS Dbtag FROM NCBI-General;

Rsite-ref ::= CHOICE {
 str VisibleString , -- may be unparsable
 db Dbtag } -- pointer to a restriction site database

END

--**********************************************************************
--
-- NCBI RNAs
-- by James Ostell, 1990
-- version 0.8
--
--**********************************************************************

NCBI-RNA DEFINITIONS ::=
BEGIN

EXPORTS RNA-ref, Trna-ext, RNA-gen, RNA-qual, RNA-qual-set;

IMPORTS Seq-loc FROM NCBI-Seqloc;

--*** rnas **********************************************
--*
--* various rnas
--*
 -- minimal RNA sequence
RNA-ref ::= SEQUENCE {
 type ENUMERATED { -- type of RNA feature
 unknown (0) ,
 premsg (1) ,
 mRNA (2) ,
 tRNA (3) ,
 rRNA (4) ,
 snRNA (5) , -- will become ncRNA, with RNA-gen.class = snRNA
 scRNA (6) , -- will become ncRNA, with RNA-gen.class = scRNA
 snoRNA (7) , -- will become ncRNA, with RNA-gen.class = snoRNA
 ncRNA (8) , -- non-coding RNA; subsumes snRNA, scRNA, snoRNA
 tmRNA (9) ,
 miscRNA (10) ,
 other (255) } ,
 pseudo BOOLEAN OPTIONAL ,
 ext CHOICE {
 name VisibleString , -- for naming "other" type
 tRNA Trna-ext , -- for tRNAs
 gen RNA-gen } OPTIONAL -- generic fields for ncRNA, tmRNA, miscRNA
 }

Trna-ext ::= SEQUENCE { -- tRNA feature extensions
 aa CHOICE { -- aa this carries
 iupacaa INTEGER ,
 ncbieaa INTEGER ,
 ncbi8aa INTEGER ,
 ncbistdaa INTEGER } OPTIONAL ,
 codon SET OF INTEGER OPTIONAL , -- codon(s) as in Genetic-code
 anticodon Seq-loc OPTIONAL } -- location of anticodon

RNA-gen ::= SEQUENCE {
 class VisibleString OPTIONAL , -- for ncRNAs, the class of non-coding RNA:
 -- examples: antisense_RNA, guide_RNA, snRNA
 product VisibleString OPTIONAL ,
 quals RNA-qual-set OPTIONAL -- e.g., tag_peptide qualifier for tmRNAs
}

RNA-qual ::= SEQUENCE { -- Additional data values for RNA-gen,
 qual VisibleString , -- in a tag (qual), value (val) format
 val VisibleString }

RNA-qual-set ::= SEQUENCE OF RNA-qual

END

--**********************************************************************
--
-- NCBI Genes
-- by James Ostell, 1990
-- version 0.8
--
--**********************************************************************

NCBI-Gene DEFINITIONS ::=
BEGIN

EXPORTS Gene-ref, Gene-nomenclature;

IMPORTS Dbtag FROM NCBI-General;

--*** Gene ***********************************************
--*
```

```
--* reference to a gene
--*

Gene-ref ::= SEQUENCE {
 locus VisibleString OPTIONAL , -- Official gene symbol
 allele VisibleString OPTIONAL , -- Official allele designation
 desc VisibleString OPTIONAL , -- descriptive name
 maploc VisibleString OPTIONAL , -- descriptive map location
 pseudo BOOLEAN DEFAULT FALSE , -- pseudogene
 db SET OF Dbtag OPTIONAL , -- ids in other dbases
 syn SET OF VisibleString OPTIONAL , -- synonyms for locus
 locus-tag VisibleString OPTIONAL , -- systematic gene name (e.g., MI0001,
ORF0069)
 formal-name Gene-nomenclature OPTIONAL
}

Gene-nomenclature ::= SEQUENCE {
 status ENUMERATED {
 unknown (0) ,
 official (1) ,
 interim (2)
 } ,
 symbol VisibleString OPTIONAL ,
 name VisibleString OPTIONAL ,
 source Dbtag OPTIONAL
}

END


--**********************************************************************
--
-- NCBI Organism
-- by James Ostell, 1994
-- version 3.0
--
--**********************************************************************

NCBI-Organism DEFINITIONS ::=
BEGIN

EXPORTS Org-ref;

IMPORTS Dbtag FROM NCBI-General;

--*** Org-ref ***********************************************
--*
--* Reference to an organism
--* defines only the organism.. lower levels of detail for biological
--* molecules are provided by the Source object
--*
```

```
Org-ref ::= SEQUENCE {
 taxname VisibleString OPTIONAL , -- preferred formal name
 common VisibleString OPTIONAL , -- common name
 mod SET OF VisibleString OPTIONAL , -- unstructured modifiers
 db SET OF Dbtag OPTIONAL , -- ids in taxonomic or culture dbases
 syn SET OF VisibleString OPTIONAL , -- synonyms for taxname or common
 orgname OrgName OPTIONAL }


OrgName ::= SEQUENCE {
 name CHOICE {
 binomial BinomialOrgName , -- genus/species type name
 virus VisibleString , -- virus names are different
 hybrid MultiOrgName , -- hybrid between organisms
 namedhybrid BinomialOrgName , -- some hybrids have genus x species name
 partial PartialOrgName } OPTIONAL , -- when genus not known
 attrib VisibleString OPTIONAL , -- attribution of name
 mod SEQUENCE OF OrgMod OPTIONAL ,
 lineage VisibleString OPTIONAL , -- lineage with semicolon separators
 gcode INTEGER OPTIONAL , -- genetic code (see CdRegion)
 mgcode INTEGER OPTIONAL , -- mitochondrial genetic code
 div VisibleString OPTIONAL } -- GenBank division code


OrgMod ::= SEQUENCE {
 subtype INTEGER {
 strain (2) ,
 substrain (3) ,
 type (4) ,
 subtype (5) ,
 variety (6) ,
 serotype (7) ,
 serogroup (8) ,
 serovar (9) ,
 cultivar (10) ,
 pathovar (11) ,
 chemovar (12) ,
 biovar (13) ,
 biotype (14) ,
 group (15) ,
 subgroup (16) ,
 isolate (17) ,
 common (18) ,
 acronym (19) ,
 dosage (20) , -- chromosome dosage of hybrid
 nat-host (21) , -- natural host of this specimen
 sub-species (22) ,
 specimen-voucher (23) ,
 authority (24) ,
 forma (25) ,
 forma-specialis (26) ,
 ecotype (27) ,
 synonym (28) ,
 anamorph (29) ,
 teleomorph (30) ,
 breed (31) ,
 gb-acronym (32) , -- used by taxonomy database
 gb-anamorph (33) , -- used by taxonomy database
 gb-synonym (34) , -- used by taxonomy database
 culture-collection (35) ,
 bio-material (36) ,
 metagenome-source (37) ,
 old-lineage (253) ,
 old-name (254) ,
 other (255) } , -- ASN5: old-name (254) will be added to next spec
 subname VisibleString ,
 attrib VisibleString OPTIONAL } -- attribution/source of name

BinomialOrgName ::= SEQUENCE {
 genus VisibleString , -- required
 species VisibleString OPTIONAL , -- species required if subspecies used
 subspecies VisibleString OPTIONAL }

MultiOrgName ::= SEQUENCE OF OrgName -- the first will be used to assign
division

PartialOrgName ::= SEQUENCE OF TaxElement -- when we don't know the genus

TaxElement ::= SEQUENCE {
 fixed-level INTEGER {
 other (0) , -- level must be set in string
 family (1) ,
 order (2) ,
 class (3) } ,
 level VisibleString OPTIONAL ,
 name VisibleString }

END


--**********************************************************************
--
-- NCBI BioSource
-- by James Ostell, 1994
-- version 3.0
--
--**********************************************************************

NCBI-BioSource DEFINITIONS ::=
BEGIN
```

```
EXPORTS BioSource;

IMPORTS Org-ref FROM NCBI-Organism;

--*********************************************************************
--
-- BioSource gives the source of the biological material
-- for sequences
--
--*********************************************************************

BioSource ::= SEQUENCE {
 genome INTEGER { -- biological context
 unknown (0) ,
 genomic (1) ,
 chloroplast (2) ,
 chromoplast (3) ,
 kinetoplast (4) ,
 mitochondrion (5) ,
 plastid (6) ,
 macronuclear (7) ,
 extrachrom (8) ,
 plasmid (9) ,
 transposon (10) ,
 insertion-seq (11) ,
 cyanelle (12) ,
 proviral (13) ,
 virion (14) ,
 nucleomorph (15) ,
 apicoplast (16) ,
 leucoplast (17) ,
 proplastid (18) ,
 endogenous-virus (19) ,
 hydrogenosome (20) ,
 chromosome (21) ,
 chromatophore (22)
 } DEFAULT unknown ,
 origin INTEGER {
 unknown (0) ,
 natural (1) , -- normal biological entity
 natmut (2) , -- naturally occurring mutant
 mut (3) , -- artificially mutagenized
 artificial (4) , -- artificially engineered
 synthetic (5) , -- purely synthetic
 other (255)
 } DEFAULT unknown ,
 org Org-ref ,
 subtype SEQUENCE OF SubSource OPTIONAL ,
 is-focus NULL OPTIONAL , -- to distinguish biological focus
 pcr-primers PCRReactionSet OPTIONAL }
```

```
PCRReactionSet ::= SET OF PCRReaction

PCRReaction ::= SEQUENCE {
 forward PCRPrimerSet OPTIONAL ,
 reverse PCRPrimerSet OPTIONAL }

PCRPrimerSet ::= SET OF PCRPrimer

PCRPrimer ::= SEQUENCE {
 seq PCRPrimerSeq OPTIONAL ,
 name PCRPrimerName OPTIONAL }

PCRPrimerSeq ::= VisibleString

PCRPrimerName ::= VisibleString

SubSource ::= SEQUENCE {
 subtype INTEGER {
 chromosome (1) ,
 map (2) ,
 clone (3) ,
 subclone (4) ,
 haplotype (5) ,
 genotype (6) ,
 sex (7) ,
 cell-line (8) ,
 cell-type (9) ,
 tissue-type (10) ,
 clone-lib (11) ,
 dev-stage (12) ,
 frequency (13) ,
 germline (14) ,
 rearranged (15) ,
 lab-host (16) ,
 pop-variant (17) ,
 tissue-lib (18) ,
 plasmid-name (19) ,
 transposon-name (20) ,
 insertion-seq-name (21) ,
 plastid-name (22) ,
 country (23) ,
 segment (24) ,
 endogenous-virus-name (25) ,
 transgenic (26) ,
 environmental-sample (27) ,
 isolation-source (28) ,
 lat-lon (29) , -- +/- decimal degrees
 collection-date (30) , -- DD-MMM-YYYY format
 collected-by (31) , -- name of person who collected the sample
 identified-by (32) , -- name of person who identified the sample
 fwd-primer-seq (33) , -- sequence (possibly more than one; semicolon-
separated)
 rev-primer-seq (34) , -- sequence (possibly more than one; semicolon-
separated)
 fwd-primer-name (35) ,
 rev-primer-name (36) ,
 metagenomic (37) ,
 mating-type (38) ,
 linkage-group (39) ,
 haplogroup (40) ,
 other (255) } ,
 name VisibleString ,
 attrib VisibleString OPTIONAL } -- attribution/source of this name

END

--**********************************************************************
--
-- NCBI Protein
-- by James Ostell, 1990
-- version 0.8
--
--**********************************************************************

NCBI-Protein DEFINITIONS ::=
BEGIN

EXPORTS Prot-ref;

IMPORTS Dbtag FROM NCBI-General;

--*** Prot-ref ***********************************************
--*
--* Reference to a protein name
--*

Prot-ref ::= SEQUENCE {
 name SET OF VisibleString OPTIONAL , -- protein name
 desc VisibleString OPTIONAL , -- description (instead of name)
 ec SET OF VisibleString OPTIONAL , -- E.C. number(s)
 activity SET OF VisibleString OPTIONAL , -- activities
 db SET OF Dbtag OPTIONAL , -- ids in other dbases
 processed ENUMERATED { -- processing status
 not-set (0) ,
 preprotein (1) ,
 mature (2) ,
 signal-peptide (3) ,
 transit-peptide (4) } DEFAULT not-set }

END
--********************************************************************
--
```

```
-- Transcription Initiation Site Feature Data Block
-- James Ostell, 1991
-- Philip Bucher, David Ghosh
-- version 1.1
--
--
--
--**********************************************************************

NCBI-TxInit DEFINITIONS ::=
BEGIN

EXPORTS Txinit;

IMPORTS Gene-ref FROM NCBI-Gene
 Prot-ref FROM NCBI-Protein
 Org-ref FROM NCBI-Organism;

Txinit ::= SEQUENCE {
 name VisibleString , -- descriptive name of initiation site
 syn SEQUENCE OF VisibleString OPTIONAL , -- synonyms
 gene SEQUENCE OF Gene-ref OPTIONAL , -- gene(s) transcribed
 protein SEQUENCE OF Prot-ref OPTIONAL , -- protein(s) produced
 rna SEQUENCE OF VisibleString OPTIONAL , -- rna(s) produced
 expression VisibleString OPTIONAL , -- tissue/time of expression
 txsystem ENUMERATED { -- transcription apparatus used at this site
 unknown (0) ,
 pol1 (1) , -- eukaryotic Pol I
 pol2 (2) , -- eukaryotic Pol II
 pol3 (3) , -- eukaryotic Pol III
 bacterial (4) ,
 viral (5) ,
 rna (6) , -- RNA replicase
 organelle (7) ,
 other (255) } ,
 txdescr VisibleString OPTIONAL , -- modifiers on txsystem
 txorg Org-ref OPTIONAL , -- organism supplying transcription apparatus
 mapping-precise BOOLEAN DEFAULT FALSE , -- mapping precise or approx
 location-accurate BOOLEAN DEFAULT FALSE , -- does Seq-loc reflect mapping
 inittype ENUMERATED {
 unknown (0) ,
 single (1) ,
 multiple (2) ,
 region (3) } OPTIONAL ,
 evidence SET OF Tx-evidence OPTIONAL }

Tx-evidence ::= SEQUENCE {
 exp-code ENUMERATED {
 unknown (0) ,
 rna-seq (1) , -- direct RNA sequencing
 rna-size (2) , -- RNA length measurement
 np-map (3) , -- nuclease protection mapping with homologous sequence ladder
 np-size (4) , -- nuclease protected fragment length measurement
 pe-seq (5) , -- dideoxy RNA sequencing
 cDNA-seq (6) , -- full-length cDNA sequencing
 pe-map (7) , -- primer extension mapping with homologous sequence ladder
 pe-size (8) , -- primer extension product length measurement
 pseudo-seq (9) , -- full-length processed pseudogene sequencing
 rev-pe-map (10) , -- see NOTE (1) below
 other (255) } ,
 expression-system ENUMERATED {
 unknown (0) ,
 physiological (1) ,
 in-vitro (2) ,
 oocyte (3) ,
 transfection (4) ,
 transgenic (5) ,
 other (255) } DEFAULT physiological ,
 low-prec-data BOOLEAN DEFAULT FALSE ,
 from-homolog BOOLEAN DEFAULT FALSE } -- experiment actually done on
 -- close homolog

 -- NOTE (1) length measurement of a reverse direction primer-extension
 -- product (blocked by RNA 5'end) by comparison with
 -- homologous sequence ladder (J. Mol. Biol. 199, 587)

END
```

ASN.1 Specification: seqalign.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI Sequence Alignment elements
-- by James Ostell, 1990
--
--**********************************************************************

NCBI-Seqalign DEFINITIONS ::=
BEGIN

EXPORTS Seq-align, Score, Score-set, Seq-align-set;

IMPORTS Seq-id, Seq-loc , Na-strand FROM NCBI-Seqloc
 User-object, Object-id FROM NCBI-General;

--*** Sequence Alignment ********************************
--*

Seq-align-set ::= SET OF Seq-align
```

```
Seq-align ::= SEQUENCE {
 type ENUMERATED {
 not-set (0) ,
 global (1) ,
 diags (2) , -- unbroken, but not ordered, diagonals
 partial (3) , -- mapping pieces together
 disc (4) , -- discontinuous alignment
 other (255) } ,
 dim INTEGER OPTIONAL , -- dimensionality
 score SET OF Score OPTIONAL , -- for whole alignment
 segs CHOICE { -- alignment data
 dendiag SEQUENCE OF Dense-diag ,
 denseg Dense-seg ,
 std SEQUENCE OF Std-seg ,
 packed Packed-seg ,
 disc Seq-align-set,
 spliced Spliced-seg,
 sparse Sparse-seg
 } ,

 -- regions of sequence over which align
 -- was computed
 bounds SET OF Seq-loc OPTIONAL,

 -- alignment id
 id SEQUENCE OF Object-id OPTIONAL,

 --extra info
 ext SEQUENCE OF User-object OPTIONAL
}

Dense-diag ::= SEQUENCE { -- for (multiway) diagonals
 dim INTEGER DEFAULT 2 , -- dimensionality
 ids SEQUENCE OF Seq-id , -- sequences in order
 starts SEQUENCE OF INTEGER , -- start OFFSETS in ids order
 len INTEGER , -- len of aligned segments
 strands SEQUENCE OF Na-strand OPTIONAL ,
 scores SET OF Score OPTIONAL }

 -- Dense-seg: the densist packing for sequence alignments only.
 -- a start of -1 indicates a gap for that sequence of
 -- length lens.
 --
 -- id=100 AAGGCCTTTTAGAGATGATGATGATGATGA
 -- id=200 AAGGCCTTTTAG.......GATGATGATGA
 -- id=300 ....CCTTTTAGAGATGATGAT....ATGA
 --
 -- dim = 3, numseg = 6, ids = { 100, 200, 300 }
 -- starts = { 0,0,-1, 4,4,0, 12,-1,8, 19,12,15, 22,15,-1, 26,19,18 }
 -- lens = { 4, 8, 7, 3, 4, 4 }
 --
```

```
Dense-seg ::= SEQUENCE { -- for (multiway) global or partial alignments
 dim INTEGER DEFAULT 2 , -- dimensionality
 numseg INTEGER , -- number of segments here
 ids SEQUENCE OF Seq-id , -- sequences in order
 starts SEQUENCE OF INTEGER , -- start OFFSETS in ids order within segs
 lens SEQUENCE OF INTEGER , -- lengths in ids order within segs
 strands SEQUENCE OF Na-strand OPTIONAL ,
 scores SEQUENCE OF Score OPTIONAL } -- score for each seg

Packed-seg ::= SEQUENCE { -- for (multiway) global or partial alignments
 dim INTEGER DEFAULT 2 , -- dimensionality
 numseg INTEGER , -- number of segments here
 ids SEQUENCE OF Seq-id , -- sequences in order
 starts SEQUENCE OF INTEGER , -- start OFFSETS in ids order for whole
alignment
 present OCTET STRING , -- Boolean if each sequence present or absent in
 -- each segment
 lens SEQUENCE OF INTEGER , -- length of each segment
 strands SEQUENCE OF Na-strand OPTIONAL ,
 scores SEQUENCE OF Score OPTIONAL } -- score for each segment

Std-seg ::= SEQUENCE {
 dim INTEGER DEFAULT 2 , -- dimensionality
 ids SEQUENCE OF Seq-id OPTIONAL ,
 loc SEQUENCE OF Seq-loc ,
 scores SET OF Score OPTIONAL }


Spliced-seg ::= SEQUENCE {
 -- product is either protein or transcript (cDNA)
 product-id Seq-id OPTIONAL,
 genomic-id Seq-id OPTIONAL,

 -- should be 'plus' or 'minus'
 product-strand Na-strand OPTIONAL ,
 genomic-strand Na-strand OPTIONAL ,

 product-type ENUMERATED {
 transcript(0),
 protein(1)
 },

 -- set of segments involved
 -- each segment corresponds to one exon
 -- exons are always in biological order
 exons SEQUENCE OF Spliced-exon ,

 -- start of poly(A) tail on the transcript
 -- For sense transcripts:
 -- aligned product positions < poly-a <= product-length
 -- poly-a == product-length indicates inferred poly(A) tail at transcript's
end
 -- For antisense transcripts:
 -- -1 <= poly-a < aligned product positions
 -- poly-a == -1 indicates inferred poly(A) tail at transcript's start
 poly-a INTEGER OPTIONAL,

 -- length of the product, in bases/residues
 -- from this (or from poly-a if present), a 3' unaligned length can be
extracted
 product-length INTEGER OPTIONAL,

 -- alignment descriptors / modifiers
 -- this provides us a set for extension
 modifiers SET OF Spliced-seg-modifier OPTIONAL
}

Spliced-seg-modifier ::= CHOICE {
 -- protein aligns from the start and the first codon
 -- on both product and genomic is start codon
 start-codon-found BOOLEAN,

 -- protein aligns to it's end and there is stop codon
 -- on the genomic right after the alignment
 stop-codon-found BOOLEAN
}


-- complete or partial exon
-- two consecutive Spliced-exons may belong to one exon
Spliced-exon ::= SEQUENCE {
 -- product-end >= product-start
 product-start Product-pos ,
 product-end Product-pos ,

 -- genomic-end >= genomic-start
 genomic-start INTEGER ,
 genomic-end INTEGER ,

 -- product is either protein or transcript (cDNA)
 product-id Seq-id OPTIONAL ,
 genomic-id Seq-id OPTIONAL ,

 -- should be 'plus' or 'minus'
 product-strand Na-strand OPTIONAL ,

 -- genomic-strand represents the strand of translation
 genomic-strand Na-strand OPTIONAL ,

 -- basic seqments always are in biologic order
 parts SEQUENCE OF Spliced-exon-chunk OPTIONAL ,
 -- scores for this exon
 scores Score-set OPTIONAL ,

 -- splice sites
 acceptor-before-exon Splice-site OPTIONAL,
 donor-after-exon Splice-site OPTIONAL,

 -- flag: is this exon complete or partial?
 partial BOOLEAN OPTIONAL,

 --extra info
 ext SEQUENCE OF User-object OPTIONAL
}


Product-pos ::= CHOICE {
 nucpos INTEGER,
 protpos Prot-pos
}


-- codon based position on protein (1/3 of aminoacid)
Prot-pos ::= SEQUENCE {
 -- standard protein position
 amin INTEGER ,

 -- 0, 1, 2, or 3 as for Cdregion
 -- 0 = not set
 -- 1, 2, 3 = actual frame
 frame INTEGER DEFAULT 0
}


-- Spliced-exon-chunk: piece of an exon
-- lengths are given in nucleotide bases (1/3 of aminoacid when product is a
-- protein)
Spliced-exon-chunk ::= CHOICE {
 -- both sequences represented, product and genomic sequences match
 match INTEGER ,

 -- both sequences represented, product and genomic sequences do not match
 mismatch INTEGER ,

 -- both sequences are represented, there is sufficient similarity
 -- between product and genomic sequences. Can be used to replace stretches
 -- of matches and mismatches, mostly for protein to genomic where
 -- definition of match or mismatch depends on translation table
 diag INTEGER ,

 -- insertion in product sequence (i.e. gap in the genomic sequence)
 product-ins INTEGER ,

 -- insertion in genomic sequence (i.e. gap in the product sequence)
 genomic-ins INTEGER
}


-- site involved in splice
Splice-site ::= SEQUENCE {
 -- typically two bases in the intronic region, always
 -- in IUPAC format
 bases VisibleString
}


-- ==========================================================================
--
-- Sparse-seg follows the semantics of dense-seg and is more optimal for
-- representing sparse multiple alignments
--
-- ==========================================================================


Sparse-seg ::= SEQUENCE {
 master-id Seq-id OPTIONAL,

 -- pairwise alignments constituting this multiple alignment
 rows SET OF Sparse-align,

 -- per-row scores
 row-scores SET OF Score OPTIONAL,

 -- index of extra items
 ext SET OF Sparse-seg-ext OPTIONAL
}

Sparse-align ::= SEQUENCE {
 first-id Seq-id,
 second-id Seq-id,

 numseg INTEGER, --number of segments
 first-starts SEQUENCE OF INTEGER , --starts on the first sequence [numseg]
 second-starts SEQUENCE OF INTEGER , --starts on the second sequence [numseg]
 lens SEQUENCE OF INTEGER , --lengths of segments [numseg]
 second-strands SEQUENCE OF Na-strand OPTIONAL ,

 -- per-segment scores
 seg-scores SET OF Score OPTIONAL
}

Sparse-seg-ext ::= SEQUENCE {
 --seg-ext SET OF {
 -- index INTEGER,
 -- data User-field
 -- }
 index INTEGER
}



-- use of Score is discouraged for external ASN.1 specifications
Score ::= SEQUENCE {
 id Object-id OPTIONAL ,
 value CHOICE {
 real REAL ,
 int INTEGER
 }
}

-- use of Score-set is encouraged for external ASN.1 specifications
Score-set ::= SET OF Score

END
```

ASN.1 Specification: seqres.asn

See also the online-version of this specification, which may be more up-to-date.

```
--$Revision$
--**********************************************************************
--
-- NCBI Sequence Analysis Results (other than alignments)
-- by James Ostell, 1990
--
--**********************************************************************

NCBI-Seqres DEFINITIONS ::=
BEGIN

EXPORTS Seq-graph;

IMPORTS Seq-loc FROM NCBI-Seqloc;

--*** Sequence Graph ********************************
--*
--* for values mapped by residue or range to sequence
--*

Seq-graph ::= SEQUENCE {
 title VisibleString OPTIONAL ,
 comment VisibleString OPTIONAL ,
 loc Seq-loc , -- region this applies to
 title-x VisibleString OPTIONAL , -- title for x-axis
 title-y VisibleString OPTIONAL ,
 comp INTEGER OPTIONAL , -- compression (residues/value)
 a REAL OPTIONAL , -- for scaling values
 b REAL OPTIONAL , -- display = (a x value) + b
 numval INTEGER , -- number of values in graph
 graph CHOICE {
 real Real-graph ,
 int Int-graph ,
 byte Byte-graph } }

Real-graph ::= SEQUENCE {
 max REAL , -- top of graph
 min REAL , -- bottom of graph
 axis REAL , -- value to draw axis on
 values SEQUENCE OF REAL }

Int-graph ::= SEQUENCE {
 max INTEGER ,
 min INTEGER ,
 axis INTEGER ,
 values SEQUENCE OF INTEGER }

Byte-graph ::= SEQUENCE { -- integer from 0-255
 max INTEGER ,
 min INTEGER ,
 axis INTEGER ,
 values OCTET STRING }

END
```

15: Biological Object Manager

Last Update: July 30, 2013.

Overview

Introduction

The Object Manager[Library xobjmgr: include | src]

The Object Manager is a library, working in conjunction with the serializable object classes (see above) to facilitate access to biological sequence data. The Object Manager has been designed to coordinate the use of these objects, particularly the management of the details of loading data from one or more potentially heterogeneous data sources. The goal is to present a consistent, flexible interface to users that minimizes their exposure to the details of interacting with biological databases and their underlying data structures.

Most of the major classes in this library have a short definition in addition to the descriptions and links below. Handles are the primary mechanism through which users access data; details of the retrieval are managed transparently by the Object Manager.

See the usage page to begin working with the Object Manager. An example and sample project have been created to further assist new users and serve as a template for new projects. We have also compiled a list of common problems encountered when using the Object Manager.

Object Manager [include/objmgr | src/objmgr]

- Top-Level Object Manager Classes
  - CObjectManager Class: Manage Serializable Data Objects object_manager [.hpp | .cpp]
  - Scope Definition for Bio-Sequence Data scope[.hpp | .cpp]
  - Data loader Base Class data_loader[.hpp | .cpp]
- Handles
  - Seq_id Handle (now located outside of the Object Manager) seq_id_handle [.hpp | .cpp]
  - Bioseq handle bioseq_handle[.hpp | .cpp]
  - Bioseq-set handle bioseq_set_handle[.hpp | .cpp]
  - Seq-entry handle seq_entry_handle[.hpp | .cpp]
  - Seq-annot handle seq_annot_handle[.hpp | .cpp]
  - Seq-feat handle seq_feat_handle[.hpp | .cpp]
  - Seq-align handle seq_align_handle[.hpp | .cpp]
  - Seq-graph handle seq_graph_handle[.hpp | .cpp]
- Accessing Sequence Data
  - Sequence Map seq_map[.hpp | .cpp]
  - Representation of/Random Access to the Letters of a Bioseq seq_vector[.hpp | .cpp]
- Iterators
  - Tree structure iterators

Chapter Outline

The following is an outline of the topics presented in this chapter:

Preface

Molecular biology is generating a vast multitude of data referring to our understanding of the processes which underlie all living things. This data is being accumulated and analyzed in thousands of laboratories all over the world. Its raw volume is growing at an astonishing rate.

In these circumstances the problem of storing, searching, retrieving and exchanging molecular biology data cannot be underestimated. NCBI maintains several databases for storing biomedical information. While the amount of information stored in these databases grows at an exponential rate, it becomes more and more important to optimize and improve the data retrieval software tools. Object Manager is a tool specifically designed to facilitate data retrieval.

The NCBI databases and software tools are designed around a particular model of biological sequence data. The nature of this data is not yet fully understood, and its fundamental properties and relationships are constantly being revised. So, the data model must be very flexible. NCBI uses Abstract Syntax Notation One (ASN.1) as a formal language to describe biological sequence data and its associated information.

Requirements

Clients must be able to analyze biological sequence data, which come from multiple heterogeneous data sources. As for 'standard' databases, we mean only NCBI GenBank. 'Nonstandard' data sources may include but are not limited to reading data from files or constructing bio sequences 'manually'.

A biologist's goal could be to investigate different combinations of data. The system should provide for transparent merging of different pieces of data, as well as various combinations of it. It is Important to note that such combinations may be incorrect or ambiguous. It is one of the possible goals of the client to discover such ambiguities.

The bio sequence data may be huge. Querying this vast amount of data from a remote database may impose severe requirements on communication lines and computer resources - both client and server. The system should provide for partial data acquisition. In other words, the system should only transmit data that is really needed, not all of it at once. At the same time this technology should not impose additional (or too much) restrictions on a client system. The process, from a client point of view, should be as transparent as possible. When and if the client needs more information, it should be retrieved 'automatically'.

Different biological sequences can refer to each other. One example of such a reference may be in the form 'the sequence of amino acids here is the same as the sequence of amino acids there' (the meaning of here and there is a separate question). The data retrieval system should be able to resolve such references automatically answering what amino acids (or nucleic acids) are actually here. At the same time, at the client's request, such automatic resolution may be turned off. Probably, the client's purpose is to investigate such references.

Biological sequences are identified by Seq-id, which may have different forms. Information about specific sequence stored in the database can be modified at any time. Sometimes, if changes are minor, this only results in creating a new submission of an existing bio sequence and assigning a new *revision* number to it. In the case of more substantial changes, a new *version* number can be assigned. From the client's point of view, the system should remain consistent when data change. Possible scenarios include:

- Database changes during client's session. Client starts working and retrieves some data from the database, the data in database then change. When client then asks for an additional data, the system should retrieve original bio sequence submission data, not the most recent one.
- Database changes between client's sessions. Client retrieves some data and ends work session. Next time the most recent submission data is retrieved, unless the client asks for a specific version number.

The system must support multithreading. It should be possible to work with bio sequence data from multiple threads.

Use cases

Biological sequence data and its associated information are specified in the NCBI data model using Abstract Syntax Notation One (ASN.1). There is a tool which, based on these specifications, generates corresponding data objects. The Object Manager manipulates these objects, so they are referenced in this document without further explanation.

The most general container object of bio sequence data, as defined in the NCBI data model, is Seq-entry. In general, Seq-entry is defined recursively as a tree of Seq-entry's (one entry refers to another one etc), where each node contains either a Bioseq or a list of other Seq-entry's plus some additional data like sequence description, sequence annotations etc. Naturally, in any such tree there is only one top-level Seq-entry (TSE).

The client must be able to define a scope of visibility and reference resolution. Such a scope is defined by the sources of data - the system uses only 'allowed' sources to look for data. Such scopes may, for instance, contain several variants of the same bio sequence (Seq-entry). Since sequences refer to each other, the scopes practically always intersect. In this case changing some data in one scope should be somehow reflected in all other scopes, which 'look' at the same data - there is a need for some sort of communication between scopes.

A scope may contain multiple top-level Seq-entry's and multiple sources of data.

Once a scope is created, a client should be able to:

- Add an externally created top-level Seq-entry to it.
- Add a data loader to it. A data loader is a link between an out-of-process source of bio sequence data and the scope; it loads data when and if necessary.
- Edit objects retrieved from the scope. Data fetched from external sources through loaders can not be modified directly. Instead, an object may be detached from its original source and the new copy provided for editing. Editing includes:
  - moving existing data from one object to another;
  - adding new data to an object; and
  - removing data from an object.

Once the scope is populated with data, a client should be able to:

- Find a Bioseq with a given Seq_id, loading the Seq-entry if necessary.
- Find a top-level Seq-entry for a sequence with a given Seq_id.

- Retrieve general information about the sequence (type, length etc., without fetching sequence data) by Seq_id.
- Obtain sequence data - actual sequence data (by Seq_id) in a specified encoding.
- Enumerate sequence descriptions and sequence annotation data, namely: features, graphs, and alignments. The annotation iterators may be fine-tuned to restrict annotation types, locations, depth of search, etc.

Multithreading. There are two scenarios:

- Several threads work with the same scope simultaneously. The scope is given to them from the outside, so this external controller is responsible for waiting for thread termination and deleting the scope.
- Different threads create their own scopes to work with the same data source. That is, the data source is a shared resource.

Classes

Definition

Here we define Object Manager's key classes and their behavior:

- Object manager
- Scope
- Data loader
- Data source
- Handles
- Seq-map
- Seq-vector
- Seq-annot
- Iterators
- CFeatTree

Object manager

Object manager manages data objects, provides them to Scopes when needed. It knows all existing Data sources and Data loaders. When a Scope needs one, it receives a data object from the Object Manager. This enables sharing and reusing of all relevant data between different Scopes. Another function of the Object Manager is letting Scopes know each other, letting Scopes to communicate. This is a barely visible entity.

Scope

Scope is a top-level object available to a client. Its purpose is to define a scope of visibility and reference resolution and provide access to the bio sequence data.

Data loader

Data loader is a link between in-process data storage and remote, out-of process data source. Its purpose is to communicate with a remote data source, receive data from there, and understand what is already received and what is missing, and pass data to the local storage (Data source). Data loader maintains its own index of what data is loaded already and references that data in the Data source.

Data source

Data source stores bio sequence data locally. Scope communicates with this object to obtain any sequence data. Data source creates and maintains internal indices to facilitate information search. Data source may contain data of several top-level Seq-entry's. In case client pushes an externally constructed Seq-entry object in the Scope, such object is stored in a separate Data source. In this case, Data source has only one top-level Seq-entry. From the other side, when Data source is linked to a Data loader, it will contain all top-level Seq-entry's retrieved by that loader.

Handles

Most objects received from the Object Manager are accessed through handles. One of the most important of them is Bioseq handle, a proxy for CBioseq. Its purpose is to facilitate access to Bioseq data. When client wants to access particular biological sequence, it requests a Bioseq handle from the Scope. Another important class is Seq-id handle which is used in many places to optimize data indexing. Other handles used in the Object Manager are:

- Bioseq-set handle
- Seq-entry handle
- Seq-annot handle
- Seq-feat handle
- Seq-align handle
- Seq-graph handle

Most handles have two versions: simple read-only handle and edit handle, which may be used to modify the data.

Seq-map

Seq-map contains general information about the sequence structure: location of data, references gaps etc.

Seq-vector

Seq-vector provides sequence data in the selected coding.

Seq-annot

A Seq-annot is a self-contained package of sequence annotations, or information that refers to specific locations on specific Bioseqs. It may contain a feature table, a set of sequence alignments, or a set of graphs of attributes along the sequence. A Bioseq may have many Seq-annot's.

See the Seq-annot section in the data model chapter for more information.

Iterators

Many objects in the Object Manager can be enumerated using iterators. Some of the iterators behave like usual container iterators (e.g. Seq-vector iterator), others have more complicated behavior depending on different arguments and flags.

Description iterators traverse bio sequence descriptions (Seq-descr and Seqdesc) in the Seq-entry. They start with the description(s) of the requested Bioseq or Seq-entry and then retrieve all descriptions iterating through the tree nodes up to the top-level Seq-entry. Starting Bioseq is defined by a Bioseq handle. Descriptions do not contain information about what Bioseq they describe, so the only way to figure it out is by description location on the tree.

Annotation iterators are utility classes for traversing sequence annotation data. Each annotation contains a reference to one or more regions on one or more sequences (Bioseq). From one point of view this is good, because we can always say which sequences are related to the given annotation. On the other hand, this creates many problems, since an annotation referencing a sequence may be stored in another sequence/Seq-entry/tree. The annotation iterators attempt to find all objects related to the given location in all Data sources from the current Scope. Data sources create indexes for all annotations by their locations. Another useful feature of the annotation iterators is location mapping: for segmented sequences the iterators can collect annotations defined on segments and adjust their locations to point to the master sequence.

There are several annotation iterator classes; some specialized for particular annotation types:

- Seq-annot iterator - traverses Seq-annot objects starting from a given Seq-entry/Bioseq up to the top-level Seq-entry (The same way as Descriptor iterators do) or down to each leaf Seq-entry.
- Annot iterator -traverses Seq-annot objects (Seq-annot) rather than individual annotations.
- Feature iterator - traverses sequence features (Seq-feat).
- Alignment iterator - traverses sequence alignments descriptions (Seq-align).
- Graph iterator - traverses sequence graphs (Seq-graph).

Tree iterators include Bioseq iterator and Seq-entry iterator, which may be used to visit leafs and nodes of a Seq-entry tree.

Seq-map iterator iterates over parts of a Bioseq. It is used mostly with segmented sequences to enumerate their segments and check their type without fetching complete sequence data.

Seq-vector iterator is used to access individual sequence characters in a selected coding.

CFeatTree

The CFeatTree class builds a parent-child feature tree in a more efficient way than repeatedly calling GetParentFeature() for each feature. The algorithm of a parent search is the same as the one used by GetParentFeature().

The class CFeatTree works with a set of features specified by calling AddFeature() or AddFeatures(). The actual tree is built the first time method GetParent() or GetChildren() is called after adding new features. Features can be added later, but the parent information is cached and will not change if parents were found already. However, features with no parent will be processed again in attempt to find parents from the newly added features.

Here's a sample code snippet that constructs a CFeatTree based on selected features:

```
// Construct the Seq-loc to get features for.
CSeq_loc seq_loc;
seq_loc.SetWhole().SetGi(src.gi);

// Make a selector to limit features to those of interest.
SAnnotSelector sel;
sel.SetResolveAll();
sel.SetAdaptiveDepth(true);
sel.IncludeFeatType(CSeqFeatData::e_Gene)
 .IncludeFeatType(CSeqFeatData::e_Cdregion)
 .IncludeFeatType(CSeqFeatData::e_Rna);

// Exclude SNP's and STS's since they won't add anything interesting
// but could significantly degrade performance.
sel.ExcludeNamedAnnots("SNP");
sel.ExcludeNamedAnnots("STS");

// Use a CFeat_CI iterator to iterate through all selected features.
CFeat_CI feat_it(CFeat_CI(*gscope, seq_loc, sel));

// Create the feature tree and add to it the features found by the
// feature iterator.
feature::CFeatTree feat_tree;
feat_tree.AddFeatures(feat_it);
```

The CFeatTree class can also improve the performance of the feature::GetBestXxxForYyy() functions, such as GetBestGeneForMrna(). Simply create the CFeatTree and pass it to the GetBestXxxForYyy() functions.

Note: There are "old" and "new" GetBestXxxForYyy() functions. The "new" functions are in the feature namespace, are located in include/objmgr/util/feature.hpp, and should be used for new development, as they are more efficient. The "old" functions are in the sequence namespace and are located in include/objmgr/util/sequence.hpp.

Attributes and Operations

- Object manager
- Scope
- Data loader
  - Interaction with the Object Manager
- Handles:
  - Bioseq handle
  - Bioseq-set handle
  - Seq-entry handle
  - Seq-annot handle
  - Seq-feat handle
  - Seq-align handle
  - Seq-graph handle
- Seq-map
- Seq-vector
- Seq-annot
  - Interaction with the Object Manager
- Iterators:
  - Bioseq iterator

- Seq-entry iterator
- Seq-descr iterator
- Seqdesc iterator
- Seq-annot iterator
- Annot iterator
- Feature iterator
- Alignment iterator
- Graph iterator
- Seq-map iterator
- Seq-vector iterator

Object manager

Before being able to use any Scopes, a client must create and initialize the Object Manager (CObjectManager). Initialization functions include registration of Data loaders, some of which may be declared as default ones. All default Data loaders are added to a Scope when the latter asks for them. All Data loaders are named, so Scopes may refer to them by name. Another kind of data object is CSeq_entry - it does not require any data loader, but also may be registered with the Object Manager. Seq-entry may not be a default data object.

CObjectManager is a multi-thread-safe singleton class, which means that only one instance of the class will be created, and it will be safely accessible from all threads. This object gets created in the first call to CObjectManager::GetInstance(void) and does not get destroyed until the program terminates (even if all references to it are destroyed), so all calls to GetInstance() will return the same object. Therefore you can either save the CRef returned by GetInstance() or call GetInstance() again for subsequent use.

Most other CObjectManager methods are used to manage Data loaders.

CObjectManager important methods

- GetInstance - returns the object manager singleton (creating it if necessary). This method can be called multiple times and/or the returned CRef can be saved.
- RegisterDataLoader - creates and registers data loader specified by driver name using plugin manager.
- FindDataLoader - finds data loader by its name. Returns pointer to the loader or null if no loader was found.
- GetRegisteredNames - fills vector of strings with the names of all registered data loaders.
- void SetLoaderOptions - allows to modify options (default flag and priority) of a registered data loader.
- bool RevokeDataLoader - revokes a registered data loader by pointer or name. Returns false if the loader is still in use. Throws exception if the loader is not registered.

See the CObjectManager API reference for an up-to-date list of all methods.

Scope

The Scope class (CScope) is designed to be a lightweight object, which could be easily created and destroyed. Scope may even be created on the stack - as an automatic object. Scope is populated with data by adding data loaders or already created Seq-entry's to it. Data loaders can only be added by name, which means it must be registered with the Object Manager beforehand. Once an externally created Seq-entry is added to a Scope, it should not be modified any more.

The main task of a scope is to cache resolved data references. Any resolved data chunk will be locked by the scope through which it was fetched. For this reason retrieving a lot of unrelated data through the same scope may consume a lot of memory. To clean a scope's cache and release the memory you can use ResetHistory or just destroy the scope and create a new one. Note: When a scope is destroyed or cleaned any handles retrieved from the scope become invalid.

CScope important methods

- AddDefaults - adds all loaders registered as default in the object manager.
- AddDataLoader - adds a data loader to the scope using the loader's name.
- AddScope - adds all loaders attached to another scope.
- AddTopLevelSeqEntry - adds a TSE to the scope. If the TSE has been already added to some scope, the data and indices will be re-used.
- AddBioseq - adds a Bioseq object wrapping it to a new Seq-entry.
- AddSeq_annot - adds a Seq-annot object to the scope.
- GetBioseqHandle - returns a Bioseq handle for the requested Bioseq. There are several versions of this function accepting different arguments. A bioseqs can be found by its Seq-id, Seq-id handle or Seq-loc. There are special flags which control data loading while resolving a Bioseq (e.g. you may want to check if a Bioseq has been already loaded by any scope or resolved in this particular scope).
- GetBioseqHandleFromTSE - allows getting a Bioseq handle restricting the search to a single top-level Seq-entry.
- GetSynonyms - returns a set of synonyms for a given Bioseq. Synonyms returned by a scope may differ from the Seq-id set stored in Bioseq object. The returned set includes all ids which are resolved to the Bioseq in this scope. An id may be hidden if it has been resolved to another Bioseq. Several modifications of the same id may appear as synonyms (e.g. accession.version and accession-only may be synonyms).
- GetAllTSEs - fills a vector of Seq-entry handles with all resolved TSEs.
- GetIds - fetches complete list of IDs for a given Seq-id without fetching the Bioseq (if supported by loader).

See the CScope API reference for an up-to-date list of all methods.

All data sources (data loaders and explicitly added data) have priorities. For example, if you call AddScope() and specify a non-default priority, the scope scans data sources in order of increasing priority to find the sequence you've requested. By default, explicitly added data have priority 9 and data loaders have priority 99, so the scope will first look in explicit data, then in data loaders. If you have conflicting data or loaders (e.g. GenBank and BLAST), you may need different priorities to make the scope first look, for example, in BLAST, and then in GenBank if the sequence is not found.

Note: the priority you've specified for a data loader at registration time (RegisterInObjectManager()) is a new default for it, and can be overridden when you add the data loader to a scope.

Data loader

The Data loader base class (CDataLoader) is almost never used by a client application directly. The specific data loaders (like GenBank data loader) have several static methods which should be used to register loaders in the Object Manager. Each of RegisterInObjectManager methods constructs a loader name depending on the arguments, checks if a loader with this name is already registered, creates and registers the loader if necessary. GetLoaderNameFromArgs methods may be used to get a potential loader's name from a set of arguments. RegisterInObjectManager returns a simple structure with two methods: IsCreated, indicating if the loader was just created or a registered loader with the same name was found, and GetLoader, returning pointer to the loader. The pointer may be null if the RegisterInObjectManager function fails or if the type of the already registered loader can not be casted to the type requested.

Interaction with the Object Manager

By default, the Object Manager will use registered data loaders to fetch basic information about all referenced Seq-entry's and annotations. For example, even if a Seq-entry contains no external references and is added to the scope using CScope::AddTopLevelSeqEntry(), the Object Manager will still use the data loader to fetch basic information about that Seq-entry and its annotations.

If the Object Manager finds a difference between a Seq-entry loaded by a data loader and an in-memory Seq-entry (having the same Seq-id) loaded with AddTopLevelSeqEntry(), the in-memory data will be used instead of the data from the data loader.

Bioseq handle

When a client wants to access a Bioseq data, it asks the Scope for a Bioseq handle (CBioseq_Handle). The Bioseq handle is a proxy to access the Bioseq data; it may be used to iterate over annotations and descriptors related to the Bioseq etc. Bioseq handle also takes care of loading any necessary data when requested. E.g. to get a sequence of characters for a segmented Bioseq it will load all segments and put their data in the right places.

Most methods of CBioseq for checking and getting object members are mirrored in the Bioseq handle's interface. Other methods are described below.

CBioseq_Handle important methods

- GetSeqId - returns Seq-id which was used to obtain the handle or null (if the handle was obtained in a way not requiring Seq-id).
- GetSeq_id_Handle - returns Seq-id handle corresponding to the id used to obtain the handle.
- IsSynonym - returns true if the id resolves to the same handle.
- GetSynonyms - returns a list of all Bioseq synonyms.
- GetParentEntry - returns a handle for the parent Seq-entry of the Bioseq.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetBioseqCore - returns TBioseqCore, which is CConstRef<CBioseq>. The Bioseq object is guaranteed to have basic information loaded (the list of Seq-ids,

Bioseq length, type etc.). Some information in the Bioseq (descriptors, annotations, sequence data) may be not loaded yet.

- GetCompleteBioseq - returns the complete Bioseq object. Any missing data will be loaded and put in the Bioseq members.
- GetComplexityLevel and GetExactComplexityLevel - allow finding a parent Seq-entry of a specified class (e.g. nuc-prot). The first method is more flexible since it considers some Seq-entry classes as equivalent.
- GetBioseqMolType - returns molecule type of the Bioseq.
- GetSeqMap - returns Seq-map object for the Bioseq.
- GetSeqVector - returns Seq-vector with the selected coding and strand.
- GetSequenceView - creates a Seq-vector for a part of the Bioseq. Depending on the flags the resulting Seq-vector may show all intervals (merged or not) on the Bioseq specified by Seq-loc, or all parts of the Bioseq not included in the Seq-loc.
- GetSeqMapByLocation - returns Seq-map constructed from a Seq-loc. The method uses the same flags as GetSequenceView.
- MapLocation - maps a Seq-loc from the Bioseq's segment to the Bioseq.

See the CBioseq_Handle API reference for an up-to-date list of all methods.

Bioseq-set handle

The Bioseq-set handle class (CBioseq_set_Handle) is a proxy class for Bioseq-set objects. Like in Bioseq handle, most of its methods allow read-only access to the members of CBioseq_set object. Some other methods are similar to the Bioseq handle's interface.

CBioseq_set_Handle important methods

- GetParentEntry - returns a handle for the parent Seq-entry of the Bioseq.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetBioseq_setCore - returns core data for the Bioseq-set. The object is guaranteed to have basic information loaded. Some information may be not loaded yet.
- GetCompleteBioseq_set - returns the complete Bioseq-set object. Any missing data will be loaded and put in the Bioseq members.
- GetComplexityLevel and GetExactComplexityLevel - allow finding a parent Seq-entry of a specified class (e.g. nuc-prot). The first method is more flexible since it considers some Seq-entry classes as equivalent.

See the CBioseq_set_Handle API reference for an up-to-date list of all methods.

Seq-entry handle

The Seq-entry handle class (CSeq_entry_Handle) is a proxy class for Seq-entry objects. Most of its methods allow read-only access to the members of Seq-entry object. Other methods may be used to navigate the Seq-entry tree.

CSeq_entry_Handle important methods

- GetParentBioseq_set - returns a handle for the parent Bioseq-set if any.
- GetParentEntry - returns a handle for the parent Seq-entry.

- GetSingleSubEntry - checks that the Seq-entry contains a Bioseq-set of just one child Seq-entry and returns a handle for this entry, otherwise throws exception.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetSeq_entryCore - returns core data for the Seq-entry. Some information may be not loaded yet.
- GetCompleteSeq_entry - returns the complete Seq-entry object. Any missing data will be loaded and put in the Bioseq members.

See the CSeq_entry_Handle API reference for an up-to-date list of all methods.

Seq-annot handle

The Seq-annot handle class (CSeq_annot_Handle) is a simple proxy for Seq-annot objects.

CSeq_annot_Handle important methods

- GetParentEntry - returns a handle for the parent Seq-entry.
- GetTopLevelEntry - returns a handle for the top-level Seq-entry.
- GetCompleteSeq_annot - returns the complete Seq-annot object. Any data stubs are resolved and loaded.

See the CSeq_annot_Handle API reference for an up-to-date list of all methods.

Seq-feat handle

The Seq-feat handle class (CSeq_feat_Handle) is a read-only proxy to Seq-feat objects data. It also simplifies and optimizes access to methods of SNP features.

Seq-align handle

The Seq-align handle class (CSeq_align_Handle) is a read-only proxy to Seq-align objects data. Most of its methods are simply mapped to the CSeq_align methods.

Seq-graph handle

The Seq-graph handle class (CSeq_graph_Handle) is a read-only proxy to Seq-graph objects data. Most of its methods are simply mapped to the CSeq_graph methods.

Seq-map

The Seq-map class (CSeqMap) object gives a general description of a biological sequence: the location and type of each segment, without the actual sequence data. It provides the overall structure of a Bioseq, or can be constructed from a Seq-loc, representing a set of locations rather than a real Bioseq. Seq-map is typically used with Seq-map iterator, which enumerates individual segments. Special flags allow selecting the types of segments to be iterated and the maximum depth of resolved references.

CSeqMap important methods

- GetSegmentsCount - returns the number of segments in the Seq-map.
- GetLength - returns the length of the whole Seq-map.
- GetMol - returns the molecule type for real bioseqs.
- begin, Begin, end, End, FindSegment - methods for normal Seq-map iteration (lower case names added for compatibility with STL).

- BeginResolved, FindResolved, EndResolved - force resolving references in the Seq-map. Optional arguments allow controlling types of segments to be shown and resolution depth.
- ResolvedRangeIterator - starts iterator over the specified range and strand only.
- CanResolveRange - checks if necessary data is available to resolve all segments in the specified range.

See the CSeqMap API reference for an up-to-date list of all methods.

Seq-vector

The Seq-vector class (CSeqVector) is a convenient representation of sequence data. It uses interface similar to the STL vector but data retrieval is optimized for better performance on big sequences. Individual characters may be accessed through operator[], but better performance may be achieved with Seq-vector iterator. Seq-vector can be obtained from a Bioseq handle, or constructed from a Seq-map or Seq-loc.

CSeqVector important methods

- size - returns length of the whole Seq-vector.
- begin, end - STL-style methods for iterating over Seq-vector.
- operator[] - provides access to individual character at a given position.
- GetSeqData - copy characters from a specified range to a string.
- GetSequenceType, IsProtein, IsNucleotide - check sequence type.
- SetCoding, SetIupacCoding, SetNcbiCoding - control coding used by Seq-vector. These methods allow selecting Iupac or Ncbi coding without checking the exact sequence type - correct coding will be selected by the Seq-vector automatically.
- GetGapChar - returns character used in the current coding to indicate gaps in the sequence.
- CanGetRange - check if sequence data for the specified range is available.
- SetRandomizeAmbiguities, SetNoAmbiguities - control randomization of ambiguities in ncbi2na coding. If set, ambiguities will be represented with random characters with distribution corresponding to the ambiguity symbol at each position. Once assigned, the same character will be returned every time for the same position.

See the CSeqVector API reference for an up-to-date list of all methods.

Seq-annot

The Seq-annot class (CSeq_annot) serves primarily as a container for annotation data. However, depending on the nature of the contained data, it may affect the behavior of the Object Manager.

CSeq_annot important methods

- SetNameDesc - set a description of type name for the Seq-annot.
- SetTitleDesc - set a description of type title for the Seq-annot.
- AddComment - set a description of type comment for the Seq-annot.
- SetCreateDate - set the Seq-annot's time of creation.
- SetUpdateDate - set the Seq-annot's time of last update.

- AddUserObject - this enables adding custom attributes to an annotation.

See the CSeq_annot API reference for an up-to-date list of all methods.

Interaction with the Object Manager

An external annotation is one residing in a TSE other than the TSE containing the Bioseq object that it annotates. This definition applies whether the TSE containing the Bioseq was loaded by a data loader or by calling CScope::AddTopLevelSeqEntry().

If a Seq-annot references external annotations, and if a data loader has been added to the scope, then by default the Object Manager will read the external annotations.

This behavior can be modified by passing an appropriate SAnnotSelector to a CFeat_CI feature iterator constructor. By default, SAnnotSelector will not exclude externals; however, calling SetExcludeExternal() on the selector will instruct the Object Manager to omit external annotations for this SAnnotSelector.

In addition you can disable/enable annotations by name or type using other methods of SAnnotSelector. Selection by name is useful for GenBank external annotations like SNPs because their names are fixed - "SNP", "CDD", etc.

Tree structure iterators

Bioseq iterator

The Bioseq iterator class (CBioseq_CI) enumerates bioseqs in a given Seq-entry. Optional filters may be used to restrict types of bioseqs to iterate.

Seq-entry iterator

The Seq-entry iterator (CSeq_entry_CI) enumerates Seq-entry's in a given parent Seq-entry or a Bioseq-set. Note that the iterator enumerates sub-entries for only one tree level. It **does not** go down the tree if it finds a sub-entry of type 'set'.

Descriptor iterators

Seq-descr iterator

The Seq-descr iterator (CSeq_descr_CI) enumerates CSeq_descr objects from a Bioseq or Seq-entry handle. The iterator starts from the specified point in the tree and goes up to the top-level Seq-entry. This provides sets of descriptors more closely related to the Bioseq/Seq-entry requested to be returned first, followed by descriptors that are more generic. To enumerate individual descriptors CSeqdesc_CI iterator should be used.

Seqdesc iterator

Another type of descriptor iterator is CSeqdesc_CI. It enumerates individual descriptors (CSeqdesc) rather than sets of them. Optional flags allow selecting type of descriptors to be included and depth of the search. The iteration starts from the requested Seq-entry or Bioseq and proceeds to the top-level Seq-entry or stops after going selected number of Seq-entry's up the tree.

Annotation iterators

Seq-annot iterator

The Seq-annot iterator (CSeq_annot_CI) may be used to enumerate CSeq_annot objects - packs of annotations (features, graphs, alignments etc.). The iterator can work in two directions:

starting from a Bioseq and going up to the top-level Seq-entry, or going down the tree from the selected Seq-entry.

Annot iterator

Although returning CSeq_annot objects, CAnnot_CI searches individual features, alignments and graphs related to the specified Bioseq or Seq-loc. It enumerates all Seq-annot's containing the requested annotations. The search parameters may be fine-tuned using SAnnotSelector for feature, alignment, or graph iterators.

SAnnotSelector is a helper class which may be used to fine-tune annotation iterator's settings. It is used with CAnnot_CI, CFeat_CI, CAlign_CI and CGraph_CI iterators. Below is the brief explanation of the class methods. Some methods have several modifications to simplify the selector usage. E.g. one can find SetOverlapIntervals() more convenient than SetOverlapType (SAnnotSelector::eOverlap_Intervals).

- SetAnnotType - selects type of annotations to search for (features, alignments or graphs). Type-specific iterators set this type automatically.
- SetFeatType - selects type of features to search for. Ignored when used with alignment or graph iterator.
- SetFeatSubtype - selects feature subtype and corresponding type.
- SetByProduct - sets flag to search features by product rather than by location.
- SetOverlapType - select type of location matching during the search. If overlap type is set to intervals, the annotation should have at least one interval intersecting with the requested ranges to be included in the results. If overlap type is set to total range, the annotation will be found even if its location has a gap intersecting with the requested range. The default value is intervals. Total ranges are calculated for each referenced Bioseq individually, even if an annotation is located on several bioseqs, which are segments of the same parent sequence.
- SetSortOrder - selects sorting of annotations: normal, reverse or none. The default value is normal.
- SetResolveMethod - defines method of resolving references in segmented bioseqs. Default value is TSE, meaning that annotations should only be searched on segments located in the same top-level Seq-entry. Other available options are none (to ignore annotations on segments) and all (to search on all segments regardless of their location). Resolving all references may produce a huge number of annotations for big bioseqs, this option should be used with care.
- SetResolveDepth - limits the depth of resolving references in segmented bioseqs. By default the search depth is not limited (set to kMax_Int).
- SetAdaptiveDepth, SetAdaptiveTrigger - set search depth limit using a trigger type/subtype. The search stops when an annotation of the trigger type is found on some level.
- SetMaxSize - limits total number of annotations to find.
- SetLimitNone, SetLimitTSE, SetLimitSeqEntry, SetLimitSeqAnnot - limits the search to a single TSE, Seq-entry or Seq-annot object.
- SetUnresolvedFlag, SetIgnoreUnresolved, SetSearchUnresolved, SetFailUnresolved - define how the iterators should behave if a reference in a sequence can not be resolved. Ignore (default) will ignore missing parts, Fail will throw CAnnotException. Search may be used to search by known ID on missing parts, but will work only if limit object is also set, since the iterator needs to know where to look for the annotations.

- SetSearchExternal - sets all flags to search for external annotations. Such annotations are packed with special bioseqs, (e.g. gnl|Annot:CDD|6 references gi 6 and contains CDD features for the gi). If SetSearchSpecial is called with the Bioseq handle for this special sequence or its TSE handle, only external CDD features from this TSE will be found. The method calls SetResolveTSE, sets limit object to the same TSE and sets SearchUnresolved flag.
- SetNoMapping - prevents the iterator from mapping locations to the top-level Bioseq. This option can dramatically increase iterators' performance when searching annotations on a segmented Bioseq.

Feature iterator

The Feature iterator (CFeat_CI) is a kind of annotation iterator. It enumerates CSeq_feat objects related to a Bioseq, Seq-loc, or contained in a particular Seq-entry or Seq-annot regardless of the referenced locations. The search parameters may be set using SAnnotSelector (preferred method) or using constructors with different arguments. The iterator returns CMappedFeat object rather than CSeq_feat. This allows accessing both the original feature (e.g. loaded from a database) and the mapped one, with its location adjusted according to the search parameters. Most methods of CMappedFeat are just proxies for the original feature members and are not listed here.

CMappedFeat important methods

- GetOriginalFeature - returns the original feature.
- GetSeq_feat_Handle - returns handle for the original feature object.
- GetMappedFeature - returns a copy of the original feature with its location/product adjusted according to the search parameters (e.g. id and ranges changed from a segment to the parent Bioseq). The mapped feature is not created unless requested. This allows improving the iterator's performance.
- GetLocation - although present in CSeq_feat class, this method does not always return the original feature's location, but first checks if the feature should be mapped, creates the mapped location if necessary and returns it. To get the unmapped location use GetOriginalFeature().GetLocation() instead.
- GetAnnot - returns handle for the Seq-annot object, containing the original feature.

See the CMappedFeat API reference for an up-to-date list of all methods.

Alignment iterator

The Alignment iterator (CAlign_CI) enumerates CSeq_align objects related to the specified Bioseq or Seq-loc. It behaves much like CFeat_CI. operator* and operator-> return a mapped CSeq_align object. To get the original alignment you can use GetOriginalSeq_align or GetSeq_align_Handle methods. The objects iterated over may be selected by using SAnnotSelector in the constructor.

Graph iterator

The Graph iterator (CGraph_CI) enumerates CSeq_graph objects related to a specific Bioseq or Seq-loc. It behaves much like CFeat_CI, returning CMappedGraph object which imitates the interface of CSeq_graph and has additional methods to access both original and mapped graphs. The objects iterated over may be selected by using SAnnotSelector in the constructor.

Note: Quality Graphs for cSRA data are not iterated by default. To include them, set the following configuration parameter:

```
[csra_loader]
quality_graphs=true
```

Seq-map iterator

The Seq-map iterator (CSeqMap_CI) is used to enumerate Seq-map segments. The segments to be iterated are selected through a SSeqMapSelector.

CSeqMap_CI important methods

- GetType - returns type of the current segment. The allowed types are eSeqGap, eSeqData, eSubMap, eSeqRef, and eSeqEnd, and eSeqChunk.
- GetPosition - returns start position of the current segment.
- GetLength - returns length of the current segment.
- IsUnknownLength - returns whether the length of the current segment is known.
- GetEndPosition - returns end position (exclusive) of the current segment.
- GetData - returns sequence data (CSeq_data). The current segment type must be eSeqData.
- GetRefSeqId - returns referenced Seq-id for segments of type eSeqRef.
- GetRefData - returns sequence data for any segment which can be resolved to a real sequence. The real position, length and strand of the data should be checked using other methods.
- GetRefPosition - returns start position on the referenced Bioseq for segments of type eSeqRef.
- GetRefEndPosition - returns end position (exclusive) on the referenced Bioseq for segments of type eSeqRef.
- GetRefMinusStrand - returns true if referenced Bioseq's strand should be reversed. If there are several levels of references for the current segment, the method checks strands on each level.

See the CSeqMap_CI API reference for an up-to-date list of all methods.

Note: Some methods will throw exceptions if called inappropriately, so you should either check for the appropriate conditions before calling these methods or catch the exceptions. The methods that throw and the appropriate conditions for calling them are:

Method	**Calling Condition**
GetData	Type must be eSeqGap or eSeqData. If type is eSeqData then GetRefPosition must return zero and GetRefMinusStrand must return false. If the data must be modified (e.g. for a delta sequence) then GetRefData should be called rather than GetData.
GetRefSeqid	Type must be eSeqRef.
GetRefData	Type must be eSeqGap or eSeqData.

Note: Some other methods will not throw exceptions if called inappropriately, and will instead return invalid data. Therefore you must check for the appropriate conditions before calling these methods or using their data:

Method	Calling Condition
GetLength	IsUnknownLength must return false.
GetEndPosition	IsUnknownLength must return false.
GetRefEndPosition	Type must be eSeqRef and IsUnknownLength must return false.

SSeqMapSelector

SSeqMapSelector is a helper class which may be used to fine-tune the Seq-map iterator's settings. Below is a brief description of its main class methods.

SSeqMapSelector important methods

- SSeqMapSelector - there is a constructor that takes flags (CSeqMap::Tflags) and a resolve count. The flags can determine which types of segments are included, while the resolve count determines how many levels over which references are resolved.
- SetPosition - selects segments containing this position.
- SetRange - selects segments within this range.
- SetStrand - selects segments matching a strand constraint.
- SetResolveCount - limits the depth of resolved references.
- SetLinkUsedTSE - limits the TSE to resolve references.
- SetFlags - selects segments matching these flags.
- SetByFeaturePolicy - a convenience method equivalent to SetFlags (my_selector.GetFlags() | CSeqMap::fByFeaturePolicy).

See the SSeqMapSelector API reference for an up-to-date list of all methods.

Here is some code that illustrates:

- iterating over data, gaps, and references;
- resolving up to 3 levels of references;
- avoiding exceptions and invalid data; and
- calling various API methods on the iterated segments.

```
// Create a new scope ("attached" to our OM).
// Add default loaders to the scope.
CScope scope(*m_ObjMgr);
scope.AddDefaults();

// Create a Seq-id.
CSeq_id seq_id;
seq_id.SetGi(123456);

// Create a bioseq handle for this seqid.
CBioseq_Handle handle = scope.GetBioseqHandle(seq_id);

// Create the selector, resolving up to 3 levels of references.
SSeqMapSelector sel(CSeqMap::fFindAny, 3);
```

```
// Iterate over the segments, printing relevant data:
for ( CSeqMap_CI map_it(handle, sel); map_it; ++map_it ) {
 CSeqMap::ESegmentType segtype = map_it.GetType();

 bool getData = ( ( segtype == CSeqMap::eSeqGap ) ||
 ( segtype == CSeqMap::eSeqData &&
 map_it.GetRefPosition() == 0 &&
 ! map_it.GetRefMinusStrand() ) );
 bool getPos = true;
 bool getLen = ( ! map_it.IsUnknownLength() );
 bool getEndPos = ( ! map_it.IsUnknownLength() );
 bool getRefSeqid = ( segtype == CSeqMap::eSeqRef );
 bool getRefData = ( segtype == CSeqMap::eSeqGap ||
 segtype == CSeqMap::eSeqData );
 bool getRefPos = ( segtype == CSeqMap::eSeqRef );
 bool getRefEndPos = ( segtype == CSeqMap::eSeqRef &&
 ! map_it.IsUnknownLength() );
 bool getRefMinus = ( segtype == CSeqMap::eSeqRef );

 cout << "Type=" << segtype;
 if ( getData ) {
 cout << " Data=";
 if ( map_it.IsSetData() ) {
 if ( segtype == CSeqMap::eSeqGap ) {
 cout << "gap";
 } else {
 const CSeq_data& data(map_it.GetData());
 cout << data.SelectionName(data.Which());
 }
 } else {
 cout << "(not set)";
 }
 }
 if ( getPos ) cout << " Pos=" << map_it.GetPosition();
 if ( getLen ) cout << " Length=" << map_it.GetLength();
 if ( getEndPos ) cout << " EndPos=" << map_it.GetEndPosition();
 if ( getRefSeqid ) cout << " Seqid=" << map_it.GetRefSeqid();
 if ( getRefData ) {
 cout << " RefData=";
 if ( segtype == CSeqMap::eSeqGap ) {
 cout << "gap";
 } else {
 const CSeq_data& refdata(map_it.GetRefData());
 cout << refdata.SelectionName(refdata.Which());
 }
 }
 if ( getRefPos ) cout << " RefPos=" << map_it.GetRefPosition();
 if ( getRefEndPos ) cout << " RefEndPos=" << map_it.GetRefEndPosition();
 if ( getRefMinus ) cout << " RefMinus=" << map_it.GetRefMinusStrand();
 cout << endl;
}
```

Seq-vector iterator

The Seq-vector iterator (CSeqVector_CI) is used to access individual characters from a Seq-vector. It has better performance than CSeqVector::operator[] when used for sequential access to the data.

CSeqVector_CI important methods

- GetSeqData - copy characters from a specified range to a string.
- GetPos, SetPos - control current position of the iterator.
- GetCoding, SetCoding - control character coding.
- SetRandomizeAmbiguities, SetNoAmbiguities - control randomization of ambiguities in ncbi2na coding. If set, ambiguities will be represented with random characters with distribution corresponding to the ambiguity symbol at each position. Once assigned, the same character will be returned every time for the same position.

See the CSeqVector_CI API reference for an up-to-date list of all methods.

Request history and conflict resolution

There are several points of potential ambiguity:

1. the client request may be incomplete;
2. the data in the database may be ambiguous;
3. the data stored by the Object Manager in the local cache may be out of date (in case the database has been updated during the client session);
4. the history of requests may create conflicts (when the Object Manager is unable to decide what exactly is the meaning of the request).

Incomplete Seq-id

Biological sequence id (Seq-id) gives a lot of freedom in defining what sequence the client is interested in. It can be a Gi - a simple integer assigned to a sequence by the NCBI 'ID' database, which in most cases is unique and univocal (Gi does not change if only annotations are changed), but it also can be an accession string only (without version number or release specification). It can specify in what database the sequence data is stored, or this information could be missing.

The Object Manager's interpretation of such requests is kind of arbitrary (yet reasonable, e.g. only the latest version of a given accession is chosen). That is, the sequence could probably be found, but only one sequence, not the list of 'matching' ones. At this point the initially incomplete Seq-id has been resolved into a complete one. That is, the client asked the Scope for a BioseqHandle providing an incomplete Seq-id as the input. The Scope resolved it into a specific complete Seq-id and returned a handle. The client may now ask the handle about its Seq-id. The returned Seq-id differs from the one provided initially by the client.

History of requests

Once the Seq-id has been resolved into a specific Seq-entry, the Object Manager keeps track of all data requests to this sequence in order to maintain consistency. That is, it is perfectly possible that few minutes later this same Seq-id could be resolved into another Seq-entry (the data in the database may change). Still, from the client point of view, as long as this is the same session, nothing should happen - the data should not change.

By 'session' we mean here the same Scope of resolution. That is, as long as the data are requested through the same Scope, it is consistent. In another Scope the data could potentially be different. The Scope can be made to forget about previous requests by calling its ResetHistory() method.

Ambiguous requests

It is possible that there are several Seq-entry's which contain requested information. In this case the processing depends on what is actually requested: sequence data or sequence annotations. The Bioseq may be taken from only one source, while annotations - from several Seq-entry's.

Request for Bioseq

Scopes use several rules when searching for the best Bioseq for each requested Seq-id. These rules are listed below in the order they are applied:

1. Check if the requested Seq-id has been already resolved to a Seq-entry within this scope. This guarantees the same Bioseq will be returned for the same Seq-id.
2. If the Seq-id requested is not resolved yet, request it from Data sources starting from the highest priority sources. Do not check lower-priority sources if something was found in the higher-priority ones.
3. If more than one Data source of the same priority contain the Bioseq or there is one Data source with several versions of the same Seq-id, ask the Data source to resolve the conflict. The Data source may take into account whether the Bioseq is most recent or not, what Seq-entry's have been already used by the Scope (preferred Seq-entry's), etc.

Request for annotations

Annotation iterators start with examining all Data Sources in the Scope to find all top-level Seq-entry's that contain annotations pointing to the given Seq-id. The rules for filtering annotations are slightly different than for resolving Bioseqs. First of all, the scope resolves the requested Seq-id and takes all annotations related to the Seq-id from its top-level Seq-entry. TSEs containing both sequence and annotations with the same Seq-id are ignored, since any other Bioseq with the same id is considered an old version of the resolved one. If there are external annotations in TSEs not containing a Bioseq with the requested Seq-id, they are also collected.

GenBank data loader configuration

Application configuration is stored in a file with the same name as application, and extension .ini. The file will be found either in the executable or in the user's home directory.

GenBank data loader looks for parameters in section [genbank] and its subsections.

Main GenBank data loader configuration section [genbank]

```
[genbank]

; loader_method lists GenBank readers - interfaces to GenBank server.
; They are checked by GenBank loader in the order of appearance in this list.
; For example the value "cache;id2" directs GenBank loader to look in cache
; reader first, then to look for information in id2 reader from GenBank
```

```
servers.
; Available readers are: id1, id2, pubseqos, pubseqos2, and cache.
loader_method = cache;id2

; preopen can be set to false to postpone GenBank connection until needed,
; or to true to open connections in all readers at GenBank construction time.
; By default, each reader opens its connection depending on reader settings.
preopen = true
```

GenBank readers configuration

Readers id1& id2
section [genbank/id1] or [genbank/id2]

```
[genbank/id1]
; no_conn means maximum number of simultaneous connections to ID server.
; By default it's 3 in multi-threaded application, and 1 in single-threaded.
no_conn = 2
; max_number_of_connections is a synonym for no_conn, e.g.:
; max_number_of_connections = 2

; If preopen is not set in [genbank] section, local setting of preopen
; will be used to determine when to open ID connection.
; If preopen is set to false, ID reader will open connection only when
needed.
; If the value is true the connection will be opened at GenBank
; construction time.
preopen = false

; ID1/ID2 service name, (default: ID1 or ID2 correspondingly)
service = ID1_TEST

; ID1/ID2 connection timeout in seconds, (default: 20 for ID1 and ID2)
timeout = 10

; ID1/ID2 connection timeout in seconds while opening and initializing,
(default: 5 for ID1 and ID2)
open_timeout = 5

; number of connection retries in case of error (default: 5)
retry = 3
```

Readers pubseqos & pubseqos2
section [genbank/pubseqos] or [genbank/pubseqos2]

```
[genbank/pubseqos]

; no_conn means maximum number of simultaneous connections to PubSeqOS
server.
; By default it's 2 in multi-threaded application, and 1 in single-threaded.
no_conn = 1

; If preopen is not set in [genbank] section, local setting of preopen will
be used
```

```
; to determine when to open PubSeqOS connection.
; If preopen is set to false, PubSeqOS reader will open connection only when
needed.
; If the value is true the connection will be opened at GenBank construction
time.
preopen = false

; PubSeqOS server name, (default: PUBSEQ_OS)
server = PUBSEQ_OS_PUBLIC

; PubSeqOS connection login name, (default: myusername)
user = myusername

; PubSeqOS connection password, (default: mypassword)
password = mypassword

; number of connection retries in case of error (default: 3)
retry = 3
```

Reader cache section [gebank/cache]

GenBank loader cache consists of two parts, **id_cache** for storing small information, and **blob_cache** for storing large sequence data. Parameters of those caches are similar and stored in two sections, **[genbank/cache/id_cache]** and **[genbank/cache/blob_cache].**

The only parameter in those sections is **driver**, which can have values: **bdb** for a cache in a local BerkeleyDB database, **netcache** for a cache in netcached. Then parameters of corresponding **ICache** plugins are stored either in **netcache** or in **bdb** subsections.

Usually, both caches use the same interface with the same parameters, so it makes sense to put interface parameters in one section and include it in both places.

For example:

```
[genbank/cache/id_cache]

driver=netcache


[genbank/cache/id_cache/netcache]

.Include = netcache


[genbank/cache/blob_cache]

driver=netcache


[genbank/cache/blob_cache/netcache]
```

```
.Include = netcache


[netcache]

; Section with parameters of netcache interface.
; Name or IP of the computer where netcached is running.
host = localhost

; Port of netcached service.
port = 9000

; Display name of this application for use by netcached in its logs and
diagnostics.
client = objmgr_demo
```

Configuring NetCached to cache GenBank data

NetCached configuration is stored in netcached.ini file either in the executable or in the user's home directory.

Configuration parameters in the file are grouped in several sections.

Section **[server]** describes parameters of the server not related to storage.

Section **[bdb]** describes parameters of BerkeleyDB database for main NetCache storage.

One or more **[icache_???]** sections describe parameters of ICache instances used by GenBank loader.

Server configuration section [server]

```
[server]

; port number server responds on
port=9000

; maximum number of clients(threads) can be served simultaneously
max_threads=25

; initial number of threads created for incoming requests
init_threads=5

; directory where server creates access log and error log
log_path=

; Server side logging
log=false

; Use name instead of IP address in keys, false by default
;use_hostname=false
```

```
; Size of thread local buffer (65536 should be fine)
tls_size=65536

; when true, if database cannot be open (corrupted) server
; automatically drops the db directory (works only for BDB)
; and creates the database from scratch
; (the content is going to be lost)
; Directory reinitialization can be forced by 'netcached -reinit'
drop_db=true

; Network inactivity timeout in seconds
network_timeout=20

; Switch for session management API
; when turned on if the last customer disconnects server shutdowns
; after waiting for 'session_shutdown_timeout'
session_mng=false

; application shuts itself down if no new sessions arrive in the
; specified time
session_shutdown_timeout=30
```

Main BerkeleyDB database configuration section [bdb]

```
[bdb]

; directory to keep the database. It is important that this
; directory resides on local drive (not NFS)
;
; WARNING: the database directory sometimes can be recursively deleted
;(when netcached started with -reinit).
;DO NOT keep any of your files(besides the database) in it.
path=e:/netcached_data

; Path to transaction log storage. By default transaction logs are stored
; at the same location as main database, but to improve performance it's
; best to put it to a dedicated fast hard drive (split I/O load)
;
transaction_log_path=

; cache name
name=nccache

; use syncronous or asyncromous writes (used with transactions)
write_sync=false

; Direct IO for database files
direct_db=false

; Direct IO for transaction logs
direct_log=false
```

```
; when 'true' the database is transaction protected
use_transactions=true

; BLOB expiration timeout in seconds
timeout=3600

; onread - update BLOB time stamp on every read
;(otherwise only creation time will taken into account)
; purge_on_startup - delete all deprecated BLOBs when startind netcached
; (may significantly slow down startup propcess)
; check_expiration - check if BLOB has expired (on read) and if it is
; return 'not found'. Otherwise BLOB lives until
; it is deleted by the internal garbage collector
timestamp=onread
# purge_on_startup check_expiration

; do not change this
keep_versions=all

; Run background cleaning thread
; (Pretty much mandatory parameter, turn it off only if you want
; to keep absolutely everything in the database)
purge_thread=true

; Delay (seconds) between purge(garbage collector) runs.
purge_thread_delay=30

; maintanance thread sleeps for specified number of milliseconds after
; each batch. By changing this parameter you can adjust the purge
; thread priority
purge_batch_sleep=100

; maintanance thread processes database records by chunks of specified
; number. If you increase this number it also increases the performance
; of purge process (at the expense of the online connections)
purge_batch_size=70

; amount of memory allocated by BerkeleyDB for the database cache
; Berkeley DB page cache) (More is better)
mem_size=50M

; when non 0 transaction LOG will be placed to memory for better performance
; as a result transactions become non-durable and there is a risk of
; loosing the data if server fails
; (set to at least 100M if planned to have bulk transactions)
;
log_mem_size=0

; Maximum size of the transaction log file
log_file_max=200M
```

```
; Percent of pages NC tries to keep available for read
; 0 - means no background dirty page write
;
memp_trickle=10

; Number of times Berkeley DB mutex spins before sleeping
; for some reason values beyond 75 somehow disable memp_trickle
;
tas_spins=200

; Specifies how often cache should remove the Berkeley DB LOG files
; Removal is triggered by the purge thread. Value of 2 means LOG is
; cleaned every second purge
purge_clean_log=2

; Call transaction checkpoint every 'checkpoint_bytes' of stored data
checkpoint_bytes=10M

; BLOBs < 10M stored in database
overflow_limit=10M

; This parameter regulates BLOB expiration. If client constantly reads
; the BLOB and you do not want it to stuck in the database forever
; (timestamp=onread), set this parameter.
; If timeout is 3600 and ttl_prolong is 2, maximum possible timeout for
; the BLOB becomes 3600 * 2 = 7200 seconds.
ttl_prolong=3

; Maximum allowed BLOB size (for a single BLOB). 0 - no restriction
max_blob_size=0

; Number of round robin volumes. 0 - no rotation
; Cache opens approx 7 files per RR volume.
rr_volumes=3
```

ICache instances configuration sections [icache_*]

Each ICache instance has an interface name which is used by clients to select the instance.

The name of the section with the ICache instance's configuration is a concatenation of the string **icache_** and the name of the instance.

For example, the parameters of an ICache instance named **ids** are stored in the section **[icache_ids]**.

The parameters inside the section are the same as the parameters in the **[bdb]** section with some exceptions.

If the **path** parameter has the same value as **path** in main **[bdb]** section, then both databases will be stored in the same directory and share the same BerkeleyDB environment.

As a result, all parameters of the BerkeleyDB environment have no meaning in an ICache section and are taken from the [**bdb**] section instead. To avoid a database conflict, all sections with the same **path** parameter must have different **name** parameters.

The GenBank data loader requires two cache instances with slightly different parameters. The first, named **ids** by default, is used for small Seq-id resolution information. The second, named **blobs** by default, is used for large Seq-entry information. The names of those caches can be changed in the client program configuration.

Similarly, NetCached configuration should describe two instances of ICache with names matching to the names on client (**ids** and **blobs** by default).

For example:

```
[icache_ids]
name=ids
path=e:/netcached_data
write_sync=false
use_transactions=true
timeout=3600
timestamp=subkey check_expiration
keep_versions=all
purge_thread=true
purge_thread_delay=3600
purge_batch_sleep=5000
purge_batch_size=10
mem_size=0
purge_clean_log=10
checkpoint_bytes=10M
overflow_limit=1M
ttl_prolong=3
page_size=small

[icache_blobs]
name=blobs
path=e:/netcached_data
write_sync=false
use_transactions=true
timeout=3600
timestamp=subkey onread check_expiration
keep_versions=all
purge_thread=true
purge_thread_delay=3600
purge_batch_sleep=5000
purge_batch_size=10
mem_size=0
purge_clean_log=10
checkpoint_bytes=10M
overflow_limit=1M
ttl_prolong
```

Use of Local Data Storage (LDS) by Object Manager

Serializable object data can be stored locally in an SQLite database for efficient access from the Object Manager.

The required libraries are:

UNIX	LIB = ncbi_xloader_lds2 lds2 xobjread id2 id1 seqsplit sqlitewrapp creaders $(COMPRESS_LIBS) $(SOBJMGR_LIBS)
	LIBS = $(SQLITE3_LIBS) $(CMPRS_LIBS) $(DL_LIBS) $(ORIG_LIBS)

Windows	id1.lib, id2.lib, lds2.lib, sqlitewrapp.lib, sqlite3.lib, ncbi_xloader_lds2.lib, xobjread.lib

A demonstration program is available: SVN | LXR

Registering the LDS loader with the Object Manager

The CLDS2_Manager class creates (or updates) an SQLite database at the path specified in its constructor. Data files that it should manage can be specified with the AddDataFile() and/or AddDataDir() methods. AddDataFile() adds a single data file; AddDataDir() adds all data files in the specified directory and its subdirectories (by default). Recursion into the subdirectories can be disabled by passing CLDS2_Manager::eDir_NoRecurse as the second argument to the AddDataDir() call. UpdateData() synchronizes the database with all the added data files. Source data files can be in ASN.1 text, ASN.1 binary, XML, or FASTA format.

For example, the following code creates an LDS database, populates it with data, registers it with the Object Manager, and adds the LDS data loader to the scope.

```
// Create/update LDS db at given path, based on data in given directory.
CRef<CLDS2_Manager> mgr(new CLDS2_Manager(db_path));
mgr->AddDataDir(data_dir);
mgr->UpdateData();

// Register LDS with Object Manager.
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path);

// Explicitly add LDS to scope.
scope.AddDataLoader(CLDS2_DataLoader::GetLoaderNameFromArgs(db_path));
```

Using both the LDS and GenBank loaders

The previous example adds the LDS data loader to the scope without adding any default loaders, including GenBank. To add both the LDS and GenBank loaders (but no other default loaders) to the scope:

```
// Create/update LDS db at given path, based on data in given directory.
CRef<CLDS2_Manager> mgr(new CLDS2_Manager(db_path));
mgr->AddDataDir(data_dir);
mgr->UpdateData();

// Register LDS with Object Manager - as first priority.
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path, -1,
 CObjectManager::eNonDefault, 1);

// Explicitly add LDS to scope.
```

```
scope.AddDataLoader(CLDS2_DataLoader::GetLoaderNameFromArgs(db_path));

// Register GenBank with Object Manager - as second priority.
CGBDataLoader::RegisterInObjectManager(*object_manager, 0,
 CObjectManager::eNonDefault, 2);

// Explicitly add GenBank to scope.
scope.AddDataLoader(CGBDataLoader::GetLoaderNameFromArgs());
```

The scope will now include just LDS and GenBank.

CObjectManager::eNonDefault was passed to the RegisterInObjectManager() method in this example simply because it is the default value for that argument, and some value was necessary so that the next argument could be specified. It could equally well have been CObjectManager::eDefault.

The last argument to RegisterInObjectManager() is the priority. Here it was set to 1 for LDS and 2 for GenBank so the Object Manager would attempt to load data via LDS first, and only if that failed would it resort to GenBank.

In the above example, the loaders were explicitly added to the scope to ensure that they were the only loaders in the scope.

To add the LDS data loader and any other default loaders to the scope:

```
// Create/update LDS db at given path, based on data in given directory.
CRef<CLDS2_Manager> mgr(new CLDS2_Manager(db_path));
mgr->AddDataDir(data_dir);
mgr->UpdateData();

// Register LDS with Object Manager - as first priority.
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path, -1,
 CObjectManager::eDefault, 1);

// Register GenBank with Object Manager - as second priority.
CGBDataLoader::RegisterInObjectManager(*object_manager, 0,
 CObjectManager::eDefault, 2);

// Add default loaders to scope.
scope.AddDefaults();
```

By registering with eDefault, the LDS data loader will be added to the scope along with the default data loaders.

Known gotchas

Resolving Data References

Multiple factors determine whether data references can be resolved or not. For example, imagine that a local data store has been created from a collection of simple annotations. References between annotations might not be resolved, unless the GenBank loader is also registered with the Object Manager, or unless a flag has been set to search unresolved annotations, as in:

```
SAnnotSelector sel;
sel.SetUnresolvedFlag(SAnnotSelector::eSearchUnresolved);
```

For more information about resolving data references, see the section on SAnnot_Selector and the associated header documentation.

Setting Loader Priority

It is the caller's responsibility to ensure that the priorities are different for different loaders – or that the same sequence won't be found by both loaders. If multiple loaders are registered with the same priority, or if they are registered without specifying a priority (which results in them both getting the default priority), and if both loaders can fetch the same data, then an exception may be thrown.

In-Memory Caching in the Object Manager and Data Loaders

The following table summarizes the classes that perform short-term, in-memory caching for various objects. A custom class must be written for short-term caching of other objects or long-term caching of any objects.

Object(s)	**Caching done by**
master TSE blob	CObjectManager
id, gi, label, taxid	CGBDataLoader
blob id	CGBDataLoader

If you want in-memory caching for objects other than those listed in the table, you can implement a cache in a CDataLoader subclass. For an example implementation, see the CGBDataLoader class. CGBDataLoader actually has two Seq-id caches - one for blob id's and the other for the other small objects listed in the table. The size for both of these caches is controlled through the [GENBANK] ID_GC_SIZE configuration parameter (i.e. their sizes can't be set independently). Subclasses of CGBDataLoader can access their configuration using the CParam methods.

Short-term caching, as applied to the Object Manager and Data Loaders, means keeping data for "a little while" in a FIFO before deleting. Long-term caching means keeping objects for "a long while" – i.e. longer than they would be kept using a short-term cache. Here, "a while" is relative to the rate at which objects are discarded, not relative to elapsed time. So short-term caching means keeping at most a given number of objects, rather than keeping objects for a given amount of time.

A CDataSource object inside the Object Manager automatically performs short-term caching of blobs for master TSEs. To set the Object Manager's blob cache size, use the [OBJMGR] BLOB_CACHE configuration parameter. This configuration parameter is created by the CParam declaration "NCBI_PARAM_DECL(unsigned, OBJMGR, BLOB_CACHE)" in src/objmgr/data_source.cpp and can be set via the environment, the registry, or manipulated via the CParam API. Note: External annotation TSEs and TSEs with Delta segments are linked to one of the master TSEs.

Short-term caching is done automatically for CGBDataLoader, but not for other data loaders. If you want short-term caching for some other data loader, you'll have to add it, possibly using CGBDataLoader as an example.

Long-term caching is not done by either the Object Manager or the GenBank data loader, so to get it you will have to implement your own mechanism. Simply keeping a handle on the objects you wish to cache long-term will prevent them from being put into the short-term cache. When you no longer need the objects to be cached, just delete the handles. Note that some system of prioritization must be used to limit the number of handles kept, since keeping handles on all object would be essentially the same as increasing the short-term cache size, which presumably failed if you're trying long-term caching. You may want to see if the CSyncQueue__priority__queue class will meet your needs.

How to use it

1. Start working with the Object Manager
2. Add externally created top-level Seq-entry to the Scope
3. Add a data loader to the Scope
4. Start working with a Bioseq
5. Access sequence data
6. Edit sequence data
7. Enumerate sequence descriptions
8. Enumerate sequence annotations
9. Use the CPrefetchManager class

Start working with the Object Manager

Include the necessary headers:

```
#include <objmgr/object_manager.hpp>
#include <objmgr/scope.hpp>
#include <objmgr/bioseq_handle.hpp>
#include <objmgr/seq_vector.hpp>
#include <objmgr/desc_ci.hpp>
#include <objmgr/feat_ci.hpp>
#include <objmgr/align_ci.hpp>
#include <objmgr/graph_ci.hpp>
```

Request an instance of the CObjectManager and store as CRef:

```
CRef<CObjectManager> obj_mgr = CObjectManager::GetInstance();
```

Create a CScope. The Scope may be created as an object on the stack, or on the heap:

```
CRef<CScope> scope1 = new CScope(*obj_mgr);
CScope scope2(*obj_mgr);
```

Add externally created top-level Seq-entry to the Scope

Once there is a Seq-entry created somehow, it can be added to the Scope using the following code:

```
CRef<CSeq_entry> entry(new CSeq_entry);
... // Populate or load the Seq-entry in some way
scope.AddTopLevelSeqEntry(*entry);
```

Add a data loader to the Scope

The data loader is designed to be a replaceable object. There can be a variety of data loaders, each of which would load data from different databases, flat files, etc. Each data loader must be registered with the Object Manager. One distinguishes them later by their names. One of the most popular data loaders is the one that loads data from GenBank - CGBDataLoader. Each loader has at least one RegisterInObjectManager() static method, the first argument is usually a reference to the Object Manager:

```
#include <objtools/data_loaders/genbank/gbloader.hpp>
...
CGBDataLoader::RegisterInObjectManager(*obj_mgr);
```

A data loader may be registered as a default or non-default loader. The GenBank loader is automatically registered as default if you don't override it explicitly. For other loaders you may need to specify additional arguments to set their priority or make them default (usually this can be done through the last two arguments of the RegisterInObjectManager() method). A Scope can request data loaders from the Object Manager one at a time - by name. In this case you will need to know the loader's name. You can get it from the loader using its GetName() method, or if you don't have a loader instance, you can use the desired loader's static method GetLoaderNameFromArgs():

```
scope.AddDataLoader(my_loader.GetName()); // with loader instance
scope.AddDataLoader(CGBDataLoader::GetLoaderNameFromArgs()); // without a
loader
```

A more convenient way to add data loaders to a Scope is to register them with the Object Manager as default and then add all the default loaders to the scope, for example:

```
CLDS2_DataLoader::RegisterInObjectManager(*object_manager, db_path, -1,
 CObjectManager::eDefault, 1);
scope.AddDefaults();
```

Start working with a Bioseq

In order to be able to access a Bioseq, one has to obtain a Bioseq handle from the Scope, based on a known Seq_id. It's always a good idea to check if the operation was successful:

```
CSeq_id seq_id;
seq_id.SetGi(3);
CBioseq_Handle handle = scope.GetBioseqHandle(seq_id);
if ( !handle ) {
 ... // Failed to get the bioseq handle
}
```

Access sequence data

The access to the sequence data is provided through the Seq-vector object, which can be obtained from a Bioseq handle. The vector may be used together with a Seq-vector iterator to enumerate the sequence characters:

```
CSeqVector seq_vec = handle.GetSeqVector(CBioseq_Handle::eCoding_Iupac);
for (CSeqVector_CI it = seq_vec.begin(); it; ++it) {
 NcbiCout << *it;
}
```

The CSeqVector class provides much more than the plain data storage - in particular, it "knows where to find" the data. As a result of a query, it may initiate a reference-resolution process, send requests to the source database for more data, etc.

A sequence map is another useful object that describes sequence data. It is a collection of segments, which describe sequence parts in general - location and type only - without providing any real data. To obtain a sequence map from a Bioseq handle:

```
CConstRef<CSeqMap> seqmap(&handle.GetSeqMap());
```

It is possible then to enumerate all the segments in the map asking their type, length or position. Note that in this example the iterator is obtained using the begin() method and will enumerate only top level segments of the Seq-map:

```
int len = 0;
for (CSeqMap::const_iterator seg = seqmap->begin() ; seg; ++seg) {
 switch (seg.GetType()) {
 case CSeqMap::eSeqData:
 len += seg.GetLength();
 break;
 case CSeqMap::eSeqRef:
 len += seg.GetLength();
 break;
 case CSeqMap::eSeqGap:
 len += seg.GetLength();
 break;
 default:
 break;
 }
}
```

Edit sequence data

Seq-entry's can be edited, but editing a Seq-entry in one scope must not affect a corresponding Seq-entry in another scope. For example, if a Seq-entry is loaded from GenBank into one scope and edited, and if the original Seq-entry is subsequently loaded into a second scope, then the Seq-entry loaded in the second scope must be the original unedited Seq-entry. Therefore, to ensure that shared Seq-entry's remain unchanged, only local copies can be edited.

Top-level Seq-entry's are thus either shared and not editable or local and editable. You can find out if a TSE is editable - for example:

```
bool editable = scope.GetTSE_Handle().CanBeEdited();
```

A TSE can be added to a scope using Scope::AddTopLevelSeqEntry(), passing either a const or a non-const CSeq_entry reference. If a non-const reference is passed then the TSE wil be local and editable; if a const reference is passed then the TSI will be shared and not editable.

Seq-entry's can also be added to a scope by using a data loader - such Seq-entry's are initially shared and not editable.

A shared TSE can be made editable by gettng an edit handle for any object in the TSE, regardless of how the Seq-entry was added to the scope. For example:

```
CBioseq_EditHandle beh = scope.GetBioseqHandle(m_SeqId).GetEditHandle();
```

Note: A CBioseq_EditHandle can also be constructed from a CBioseq_Handle - but only if the TSE is already editable. If the TSE is not editable then the CBioseq_EditHandle constructor will throw an exception to prevent accidental editing.

Once a TSE is editable, any object within it can be edited with either a CBioseq_EditHandle or a CBioseq_Handle.

Enumerate sequence descriptions

Descriptor iterators may be initialized with a Bioseq handle or Seq-entry handle. This makes it possible to enumerate all CSeqdesc objects the Bioseq or the Seq-entry refers to:

```
for (CSeqdesc_CI desc_it(handle); desc_it; ++desc_it) {
 const CSeqdesc& desc = *desc_it;
 ... // your code here
}
```

Another type of descriptor iterator iterates over sets of descriptors rather than individual objects:

```
for (CSeq_descr_CI descr_it(handle); descr_it; ++descr_it) {
 const CSeq_descr& descr = *descr_it;
 ... // your code here
}
```

Enumerate sequence annotations

Annotation iterators may be used to enumerate annotations (features, alignments and graphs) related to a Bioseq or a Seq-loc. They are very flexible and can be fine-tuned through Annot-selector structure:

```
// Search all TSEs in the Scope for gene features
SAnnotSelector sel;
sel.SetFeatType(CSeqFeatData::e_Gene);
/// both start and stop are 0 - iterate the whole bioseq
CFeat_CI feat_it(handle, 0, 0, sel);
for (; feat_it; ++feat_it) {
 const CSeq_loc& loc = feat_it->GetLocation();
 ... // your code here
}
```

The next example shows slightly more complicated settings for the feature iterator. The selector forces resolution of all references, both near (to Bioseqs located in the same TSE) and far. The features will be collected from all segments resolved. Since this may result in loading a lot of external Bioseqs, the selector is set to restrict the depth of references to 2 levels:

```
SAnnotSelector sel;
sel.SetFeatType(CSeqFeatData::e_Gene)
 .SetReaolveAll()
 .SetResolveDepth(2);
CFeat_CI feat_it(handle, 0, 0, sel);
for (; feat_it; ++feat_it) {
 const CSeq_loc& loc = feat_it->GetLocation();
 ... // your code here
}
```

Usage of alignment and graph iterators is similar to the feature iterator:

```
CAlign_CI align_it(handle, 0, 0);
...
CGraph_CI graph_it(handle, 0, 0);
...
```

All the above examples iterate annotations in a continuous interval on a Bioseq. To specify more complicated locations a Seq-loc may be used instead of the Bioseq handle. The Seq-loc may even reference different ranges on several Bioseqs:

```
CSeq_loc loc;
CSeq_loc_mix& mix = loc.SetMix();
... // fill the mixed location
for (CFeat_CI feat_it(scope, loc); feat_it; ++feat_it) {
 const CSeq_loc& feat_loc = feat_it->GetLocation();
 ... // your code here
}
```

Use the cSRA data loader

To access cSRA data, use the cSRA data loader, for example:

```
CRef<CObjectManager> om(CObjectManager::GetInstance());
CCSRADataLoader::RegisterInObjectManager(*om, CObjectManager::eDefault, 0);
CGBDataLoader::RegisterInObjectManager(*om);
```

Note that to minimize the quantity of data transferred, quality graphs are not returned by default. If you want them, you'll need to set a configuration parameter, for example:

```
[csra_loader]
quality_graphs=true
```

Also, the returned data will be cilpped to exclude poor quality reads. If you want all data, including poor quality, you'll need to set another configuration parameter, for example:

```
[csra]
clip_by_quality=false
```

Use the CPrefetchManager class

Suppose you want to retrieve all the features for several hundred thousand protein sequences. Features don't consume much memory and protein sequences typically have a small number of features, so it should be feasible to simultaneously load all the features into memory.

The CPrefetchManager class was designed to improve the efficiency of this type of data retrieval, as illustrated here:

```
// Set up all the object manager stuff.
m_ObjMgr = CObjectManager::GetInstance();
CGBDataLoader::RegisterInObjectManager(*m_ObjMgr);
CScope scope(*m_ObjMgr);
scope.AddDefaults();
SAnnotSelector sel(CSeqFeatData::e_not_set);
sel.SetResolveMethod(sel.eResolve_All);

// Create a vector for IDs.
vector<CSeq_id_Handle> m_Ids;
PopulateTheIdVectorSomehow(&m_Ids);

// Create the prefetch manager.
m_PrefMgr = new CPrefetchManager();

// Create a prefetch sequence, using the prefetch manager and based on a
// feature iterator action source (in turn based on the scope, IDs, and
// feature selector).
// Note: CPrefetchSequence is designed to take ownership of the action
// source, so do not delete it or use auto_ptr etc to manage it.
CRef<CPrefetchSequence> prefetch;
prefetch = new CPrefetchSequence(*m_PrefMgr,
new CPrefetchFeat_CIActionSource(CScopeSource::New(scope),
m_Ids, sel));

// Fetch data for each ID.
for ( size_t i = 0; i < m_Ids.size(); ++i ) {

// Get a feature iterator that uses the prefetch.
CRef<CPrefetchRequest> token = prefetch->GetNextToken();
CFeat_CI it = CStdPrefetch::GetFeat_CI(token);

// Do something with the fetched features.
for ( ; it; ++it ) {
DoSomethingInterestingWithTheFeature(it->GetOriginalFeature());
}
}
```

Note: Error handling was removed from the above code for clarity - please see the Object Manager test code for examples of appropriate error handling.

Educational Exercises

Setup the framework for the C++ Object Manager learning task

Starting point

To jump-start your first project utilizing the new C++ Object Manager in the C++ Toolkit framework on a UNIX platform, we suggest using the new_project shell script, which creates a sample application and a makefile:

1 Create a new project called task in the folder task using the new_project shell script (this will create the folder, the source file and the makefile):

```
new_project task app/objmgr
```

2 Build the sample project and run the application:

```
cd task
make -f Makefile.task_app
./task -gi 333
```

The output should look like this:

```
First ID: emb|CAA23443.1|
Sequence: length=263, data=MARFLGLCTW
# of descriptions: 6
# of features:
[whole] Any: 2
[whole] Genes: 0
[0..9] Any: 2
[0..999, TSE] Any: 1
# of alignments:
[whole] Any: 0
Done
```

3 Now you can go ahead and convert the sample code in the task.cpp into the code that performs your learning task.

The new_project script can also be used to create a new project on Windows, and the usage is the same as on UNIX.

How to convert the test application into CGI one?

In order to convert your application into CGI one:

1 Create copy of the source (task.cpp) and makefile (Makefile.task_app)

```
cp task.cpp task_cgi.cpp
cp Makefile.task_app Makefile.task_cgiapp
```

2 Edit the makefile for the CGI application (Makefile.task_cgiapp): change application name, name of the source file, add cgi libraries:

```
APP = task.cgi
SRC = task_cgi

LIB = xobjmgr id1 seqset $(SEQ_LIBS) pub medline biblio general \
xser xhtml xcgi xutil xconnect xncbi
```

LIBS = $(NCBI_C_LIBPATH) $(NCBI_C_ncbi) $(FASTCGI_LIBS) \
$(NETWORK_LIBS) $(ORIG_LIBS)

3. Build the project (at this time it is not a CGI application yet):

   make -f Makefile.task_cgiapp
4. Convert task_cgi.cpp into a CGI application.

Please also see the section on FCGI Redirection and Debugging CGI Programs for more information.

Convert CGI application into Fast-CGI one

In the LIB=... section of Makefile.task_cgiapp, just replace xcgi library by xfcgi:

```
LIB = xobjmgr id1 seqset $(SEQ_LIBS) pub medline biblio general \
 xser xhtml xfcgi xutil xconnect xncbi
```

Task Description

We have compiled here a list of teaching examples to help you start working with the C++ Object Manager. Completing them, getting your comments and investigating the problems encountered would let us give warnings of issues to deal with in the nearest future, better understand what modifications should be made to this software system.

The main idea here is to build one task on the top of another, in growing level of complexity:

1. having a Seq-id (GI), get the Bioseq;
2. print the Bioseq's title descriptor;
3. print the Bioseq's length;
4. dump the Seg-map structure;
5. print the total number of cd-region features on the Bioseq;
6. calculate percentage of 'G' and 'C' symbols in the whole sequence;
7. calculate percentage of 'G' and 'C' symbols within cd-regions;
8. calculate percentage of 'G' and 'C' symbols for regions outside any cd-region feature;
9. convert the application into a CGI one;
10. convert the application into a FCGI one.

Test Bioseqs

Below is the list of example sequences to use with the C++ toolkit training course. It starts with one Teaching Example that has one genomic nucleic acid sequence and one protein with a cd-region. Following that is the list of Test Examples. Once the code is functioning on the Teaching Example, we suggest running it through these. They include a bunch of different conditions: short sequence with one cd-region, longer with 6 cd-regions, a protein record (this is an error, and code should recover), segmented sequence, 8 megabase genomic contig, a popset member, and a draft sequence with no cd-regions.

Teaching example

IDs and description of the sequence to be used as a simple teaching example is shown in Table 1.

The application should produce the following results for the above Bioseq:

```
ID: emb|AJ438945.1|HSA438945 + gi|19584253
Homo sapiens SLC16A1 gene for monocarboxylate transporter isoform 1, exons
2-5
Sequence length: 17312
Sequence map:
 Segment: pos=0, length=17312, type=DATA
Total: 40.29%
 cdr0: 46.4405%
Cdreg: 46.4405%
Non-Cdreg: 39.7052%
```

Test examples

More complicated test Bioseqs are listed in Table 2.

Correct Results

Below are shown the correct results for each of the test Bioseqs. You can use them as reference to make sure your application works correctly.

```
ID: gb|J01066.1|DROADH + gi|156787
D.melanogaster alcohol dehydrogenase gene, complete cds.
Sequence length: 2126
Sequence map:
 Segment: pos=0, length=2126, type=DATA
Total: 45.8137%
 cdr0: 57.847%
Cdreg: 57.847%
Non-Cdreg: 38.9668%

ID: gb|U01317.1|HUMHBB + gi|455025
Human beta globin region on chromosome 11.
Sequence length: 73308
Sequence map:
 Segment: pos=0, length=73308, type=DATA
Total: 39.465%
 cdr0: 52.9279%
 cdr1: 53.6036%
 cdr2: 53.6036%
 cdr3: 49.2099%
 cdr4: 54.5045%
 cdr5: 56.3063%
 cdr6: 56.7568%
Cdreg: 53.2811%
Non-Cdreg: 38.9403%

ID: emb|AJ293577.1|HSA293577 + gi|14971422
Homo sapiens partial MOCS1 gene, exon 1 and joined CDS
Sequence length: 913
Sequence map:
 Segment: pos=0, length=913, type=DATA
Total: 54.655%
 cdr0: 58.3765%
```

```
Cdreg: 58.3765%
Non-Cdreg: 51.5837%

ID: gb|AH011004.1|SEG_Y043402S + gi|19550966
Mus musculus light ear protein (le) gene, complete cds.
Sequence length: 5571
Sequence map:
 Segment: pos=0, length=255, type=DATA
 Segment: pos=255, length=0, type=GAP
 Segment: pos=255, length=306, type=DATA
 Segment: pos=561, length=0, type=GAP
 Segment: pos=561, length=309, type=DATA
 Segment: pos=870, length=0, type=GAP
 Segment: pos=870, length=339, type=DATA
 Segment: pos=1209, length=0, type=GAP
 Segment: pos=1209, length=404, type=DATA
 Segment: pos=1613, length=0, type=GAP
 Segment: pos=1613, length=349, type=DATA
 Segment: pos=1962, length=0, type=GAP
 Segment: pos=1962, length=361, type=DATA
 Segment: pos=2323, length=0, type=GAP
 Segment: pos=2323, length=369, type=DATA
 Segment: pos=2692, length=0, type=GAP
 Segment: pos=2692, length=347, type=DATA
 Segment: pos=3039, length=0, type=GAP
 Segment: pos=3039, length=1066, type=DATA
 Segment: pos=4105, length=0, type=GAP
 Segment: pos=4105, length=465, type=DATA
 Segment: pos=4570, length=0, type=GAP
 Segment: pos=4570, length=417, type=DATA
 Segment: pos=4987, length=0, type=GAP
 Segment: pos=4987, length=584, type=DATA
Total: 57.2305%
 cdr0: 59.5734%
Cdreg: 59.5734%
Non-Cdreg: 55.8899%

ID: ref|NT_017168.8|HS7_17324 + gi|18565551
Homo sapiens chromosome 7 working draft sequence segment
Sequence length: 8470605
Sequence map:
 Segment: pos=0, length=29884, type=DATA
 Segment: pos=29884, length=100, type=GAP
 Segment: pos=29984, length=20739, type=DATA
 Segment: pos=50723, length=100, type=GAP
 Segment: pos=50823, length=157624, type=DATA
 Segment: pos=208447, length=29098, type=DATA
 Segment: pos=237545, length=115321, type=DATA
 Segment: pos=352866, length=25743, type=DATA
 Segment: pos=378609, length=116266, type=DATA
 Segment: pos=494875, length=144935, type=DATA
```

```
Segment: pos=639810, length=108678, type=DATA
Segment: pos=748488, length=102398, type=DATA
Segment: pos=850886, length=149564, type=DATA
Segment: pos=1000450, length=120030, type=DATA
Segment: pos=1120480, length=89411, type=DATA
Segment: pos=1209891, length=51161, type=DATA
Segment: pos=1261052, length=131072, type=DATA
Segment: pos=1392124, length=118395, type=DATA
Segment: pos=1510519, length=70119, type=DATA
Segment: pos=1580638, length=59919, type=DATA
Segment: pos=1640557, length=131072, type=DATA
Segment: pos=1771629, length=41711, type=DATA
Segment: pos=1813340, length=131072, type=DATA
Segment: pos=1944412, length=56095, type=DATA
Segment: pos=2000507, length=93704, type=DATA
Segment: pos=2094211, length=82061, type=DATA
Segment: pos=2176272, length=73699, type=DATA
Segment: pos=2249971, length=148994, type=DATA
Segment: pos=2398965, length=37272, type=DATA
Segment: pos=2436237, length=96425, type=DATA
Segment: pos=2532662, length=142196, type=DATA
Segment: pos=2674858, length=58905, type=DATA
Segment: pos=2733763, length=94760, type=DATA
Segment: pos=2828523, length=110194, type=DATA
Segment: pos=2938717, length=84638, type=DATA
Segment: pos=3023355, length=94120, type=DATA
Segment: pos=3117475, length=46219, type=DATA
Segment: pos=3163694, length=7249, type=DATA
Segment: pos=3170943, length=118946, type=DATA
Segment: pos=3289889, length=127808, type=DATA
Segment: pos=3417697, length=51783, type=DATA
Segment: pos=3469480, length=127727, type=DATA
Segment: pos=3597207, length=76631, type=DATA
Segment: pos=3673838, length=81832, type=DATA
Segment: pos=3755670, length=21142, type=DATA
Segment: pos=3776812, length=156640, type=DATA
Segment: pos=3933452, length=117754, type=DATA
Segment: pos=4051206, length=107098, type=DATA
Segment: pos=4158304, length=15499, type=DATA
Segment: pos=4173803, length=156199, type=DATA
Segment: pos=4330002, length=89478, type=DATA
Segment: pos=4419480, length=156014, type=DATA
Segment: pos=4575494, length=105047, type=DATA
Segment: pos=4680541, length=120711, type=DATA
Segment: pos=4801252, length=119796, type=DATA
Segment: pos=4921048, length=35711, type=DATA
Segment: pos=4956759, length=131072, type=DATA
Segment: pos=5087831, length=1747, type=DATA
Segment: pos=5089578, length=38864, type=DATA
Segment: pos=5128442, length=131072, type=DATA
Segment: pos=5259514, length=97493, type=DATA
```

```
 Segment: pos=5357007, length=125390, type=DATA
 Segment: pos=5482397, length=96758, type=DATA
 Segment: pos=5579155, length=1822, type=DATA
 Segment: pos=5580977, length=144039, type=DATA
 Segment: pos=5725016, length=58445, type=DATA
 Segment: pos=5783461, length=158094, type=DATA
 Segment: pos=5941555, length=4191, type=DATA
 Segment: pos=5945746, length=143965, type=DATA
 Segment: pos=6089711, length=107230, type=DATA
 Segment: pos=6196941, length=158337, type=DATA
 Segment: pos=6355278, length=25906, type=DATA
 Segment: pos=6381184, length=71810, type=DATA
 Segment: pos=6452994, length=118113, type=DATA
 Segment: pos=6571107, length=118134, type=DATA
 Segment: pos=6689241, length=92669, type=DATA
 Segment: pos=6781910, length=123131, type=DATA
 Segment: pos=6905041, length=136624, type=DATA
 Segment: pos=7041665, length=177180, type=DATA
 Segment: pos=7218845, length=98272, type=DATA
 Segment: pos=7317117, length=22979, type=DATA
 Segment: pos=7340096, length=123747, type=DATA
 Segment: pos=7463843, length=13134, type=DATA
 Segment: pos=7476977, length=156146, type=DATA
 Segment: pos=7633123, length=59501, type=DATA
 Segment: pos=7692624, length=107689, type=DATA
 Segment: pos=7800313, length=29779, type=DATA
 Segment: pos=7830092, length=135950, type=DATA
 Segment: pos=7966042, length=71035, type=DATA
 Segment: pos=8037077, length=129637, type=DATA
 Segment: pos=8166714, length=80331, type=DATA
 Segment: pos=8247045, length=49125, type=DATA
 Segment: pos=8296170, length=131072, type=DATA
 Segment: pos=8427242, length=25426, type=DATA
 Segment: pos=8452668, length=100, type=GAP
 Segment: pos=8452768, length=16014, type=DATA
 Segment: pos=8468782, length=100, type=GAP
 Segment: pos=8468882, length=1723, type=DATA
Total: 37.2259%
 cdr0: 39.6135%
 cdr1: 38.9474%
 cdr2: 57.362%
 cdr3: 59.144%
 cdr4: 45.4338%
 cdr5: 37.6812%
 cdr6: 58.9856%
 cdr7: 61.1408%
 cdr8: 51.2472%
 cdr9: 44.2105%
 cdr10: 49.1071%
 cdr11: 43.6508%
 cdr12: 38.3754%
```

```
cdr13: 39.1892%
cdr14: 42.2222%
cdr15: 49.5763%
cdr16: 44.4034%
cdr17: 42.9907%
cdr18: 47.619%
cdr19: 47.3684%
cdr20: 47.973%
cdr21: 38.6544%
cdr22: 45.3052%
cdr23: 37.7115%
cdr24: 36.1331%
cdr25: 61.4583%
cdr26: 51.9878%
cdr27: 47.6667%
cdr28: 45.3608%
cdr29: 38.7387%
cdr30: 37.415%
cdr31: 40.5405%
cdr32: 41.1819%
cdr33: 42.6791%
cdr34: 43.7352%
cdr35: 44.9235%
cdr36: 38.218%
cdr37: 34.4928%
cdr38: 44.3137%
cdr39: 37.9734%
cdr40: 37.0717%
cdr41: 48.6772%
cdr42: 38.25%
cdr43: 48.8701%
cdr44: 46.201%
cdr45: 46.7803%
cdr46: 55.8405%
cdr47: 43.672%
cdr48: 50.3623%
cdr49: 65.4835%
cdr50: 52.6807%
cdr51: 45.7447%
cdr52: 53.7037%
cdr53: 49.6599%
cdr54: 38.5739%
cdr55: 63.3772%
cdr56: 37.6274%
cdr57: 38.0952%
cdr58: 39.6352%
cdr59: 39.6078%
cdr60: 58.4795%
cdr61: 49.4987%
cdr62: 47.0968%
cdr63: 45.0617%
```

```
 cdr64: 41.5133%
 cdr65: 40.2516%
 cdr66: 39.6208%
 cdr67: 40.4412%
 cdr68: 43.0199%
 cdr69: 40.5512%
 cdr70: 54.7325%
 cdr71: 45.3034%
 cdr72: 55.6634%
 cdr73: 43.7107%
 cdr74: 45.098%
 cdr75: 43.8406%
 cdr76: 49.4137%
 cdr77: 44.7006%
 cdr78: 44.6899%
 cdr79: 56.4151%
 cdr80: 36.1975%
 cdr81: 34.8238%
 cdr82: 38.5447%
 cdr83: 44.0451%
 cdr84: 45.6684%
 cdr85: 45.1696%
 cdr86: 40.9462%
 cdr87: 56.044%
 cdr88: 46.2366%
 cdr89: 41.1765%
 cdr90: 42.9698%
 cdr91: 47.8261%
 cdr92: 43.2234%
 cdr93: 49.7849%
 cdr94: 43.3755%
 cdr95: 51.2149%
Cdreg: 44.397%
Non-Cdreg: 37.1899%

ID: gb|AF022257.1| + gi|2415435
HIV-1 patient ACH0039, clone 3918C6 from The Netherlands, envelope
glycoprotein V3 region (env) gene, partial cds.
Sequence length: 388
Sequence map:
 Segment: pos=0, length=388, type=DATA
Total: 31.9588%
 cdr0: 31.9588%
Cdreg: 31.9588%
Non-Cdreg: 0%

ID: gb|AC116052.1| + gnl|WUGSC|RP23-291E18 + gi|19697559
Sequence length: 18561
Sequence map:
 Segment: pos=0, length=1082, type=DATA
 Segment: pos=1082, length=100, type=GAP
 Segment: pos=1182, length=1086, type=DATA
 Segment: pos=2268, length=100, type=GAP
 Segment: pos=2368, length=1096, type=DATA
 Segment: pos=3464, length=100, type=GAP
 Segment: pos=3564, length=1462, type=DATA
 Segment: pos=5026, length=100, type=GAP
 Segment: pos=5126, length=1217, type=DATA
 Segment: pos=6343, length=100, type=GAP
 Segment: pos=6443, length=1450, type=DATA
 Segment: pos=7893, length=100, type=GAP
 Segment: pos=7993, length=1086, type=DATA
 Segment: pos=9079, length=100, type=GAP
 Segment: pos=9179, length=1127, type=DATA
 Segment: pos=10306, length=100, type=GAP
 Segment: pos=10406, length=1145, type=DATA
 Segment: pos=11551, length=100, type=GAP
 Segment: pos=11651, length=1257, type=DATA
 Segment: pos=12908, length=100, type=GAP
 Segment: pos=13008, length=1024, type=DATA
 Segment: pos=14032, length=100, type=GAP
 Segment: pos=14132, length=1600, type=DATA
 Segment: pos=15732, length=100, type=GAP
 Segment: pos=15832, length=2729, type=DATA
Total: 43.9253%
No coding regions found

ID: sp|Q08345|DDR1_HUMAN + gi|729008
Epithelial discoidin domain receptor 1 precursor (Tyrosine kinase DDR)
(Discoidin receptor tyrosine kinase) (Tyrosine-protein kinase CAK)
(Cell adhesion kinase) (TRK E) (Protein-tyrosine kinase RTK 6)
(CD167a antigen) (HGK2).
Sequence length: 913
Sequence map:
 Segment: pos=0, length=913, type=DATA
Not a DNA
```

Common problems

1. How to construct Seq_id by accession?
2. What is the format of data CSeqVector returns?
3. What to pay attention to when processing cd-regions?

How to construct Seq_id by accession?

CSeq_id class has constructor, accepting a string, which may contain a Bioseq accession, or accession and version separated with dot. If no version is provided, the Object Manager will try to find and fetch the latest one.

What is the format of data CSeqVector returns?

GetSeqVector method of CBioseq_Handle has optional argument to select data coding. One of the possible values for this argument is CBioseq_Handle::eCoding_Iupac. It forces the resulting Seq-vector to convert data to printable characters - either Iupac-na or Iupac-aa, depending on the sequence type. Gaps in the sequence are coded with special character, which can be received using CSeqVector::GetGapChar, for nucleotides in Iupac coding it will be 'N' character. Note that when calculating the percentage of 'G' /'C' in a sequence you need to ignore gaps.

What to pay attention to when processing cd-regions?

When looking for cd-regions on a sequence, you get a set of features, which locations describe their position on the sequence. Please note, that these locations may, and do overlap, which makes calculating percentage of 'G'/'C' in the cd-regions much more difficult. To simplify this part of the task you can merge individual cd-region locations using CSeq_loc methods (do not forget to sort the Seq-locs for correct merging) and use the resulting Seq-loc to initialize a Seq-vector. To calculate percentage of 'G'/'C' for non-cdr parts of a sequence create a new Seq-loc with CSeq_loc::Subtract() method.

Table 1. Teaching Example: Sequence

Accession	Version	Gi	Definition
AJ438945	AJ438945.1	19584253	Homo sapiens SLC16A1 gene...

Table 2. Test Examples: Sequences

Accession	Version	Gi	Definition
J01066	J01066.1	156787	D.melanogaster alcohol dehydrogenase gene, complete cds
U01317	U01317.1	455025	Human beta globin region on chromosome 11.
AJ293577	AJ293577.1	14971422	Homo sapiens partial MOCS1 gene, exon 1 and joined CDS
AH01100	AH011004.1	19550966	Mus musculus light ear protein (le) gene, complete cds
NT_017168	NT_017168.8	18565551	Homo sapiens chromosome 7 working draft sequence segment
AF022257	AF022257.1	2415435	HIV-1 patient ACH0039, clone 3918C6 from The Netherlands...
AC116052	AC116052.1	19697559	Mus musculus chromosome UNK clone
Q08345	Q08345.1	729008	Epithelial discoidin domain receptor 1 precursor...

16: BLAST API

Thomas Madden
madden@ncbi.nlm.nih.gov

Jason Papadopoulos
papadopo@ncbi.nlm.nih.gov

Christiam Camacho
camacho@ncbi.nlm.nih.gov

George Coulouris
coulouri@ncbi.nlm.nih.gov

Kevin Bealer
bealer@ncbi.nlm.nih.gov

Created: August 22, 2006.
Last Update: April 13, 2010.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

BLAST (Basic Local Alignment Search Tool) is used to perform sequence similarity searches. Most often this means that BLAST is used to search a sequence (either DNA or protein) against a database of other sequences (either all nucleotide or all protein) in order to identify similar sequences. BLAST has many different flavors and can not only search DNA against DNA or protein against protein but also can translate a nucleotide query and search it against a protein database as well as the other way around. It can also compute a “profile” for the query sequence and use that for further searches as well as search the query against a database of profiles. BLAST is available as a web service at the NCBI, as a stand-alone binary, and is built into other tools. It is an extremely versatile program and probably the most heavily used similarity search program in the world. BLAST runs on a multitude of different platforms that include Windows, MacOS, LINUX, and many flavors of UNIX. It is also under continuing development with new algorithmic innovations. Multiple references to BLAST can be found at http://www.ncbi.nlm.nih.gov/BLAST/blast_references.shtml.

The version of BLAST in the NCBI C++ Toolkit was rewritten from scratch based upon the version in the C Toolkit that was originally introduced in 1997. A decision was made to break the code for the new version of BLAST into two different categories. There is the “core” code of BLAST that is written in vanilla C and does not use any part of the NCBI C or C++ Toolkits. There is also the “API” code that is written in C++ and takes full advantage of the tools provided by the NCBI C++ Toolkit. The reason to write the core part of the code in vanilla C was so that the same code could be used in the C Toolkit (to replace the 1997 version) as well as to make it possible for researchers interested in algorithmic development to work with the core of BLAST independently of any Toolkit. Even though the core part was written without the benefit of the C ++ or C Toolkits an effort was made to conform to the Programming Policies and Guidelines chapter of this book. Doxygen-style comments are used to allow API documentation to be automatically generated (see the BLAST Doxygen link at http://www.ncbi.nlm.nih.gov/IEB/

ToolBox/CPP_DOC/doxyhtml/group__AlgoBlast.html). Both the core and API parts of BLAST can be found under algo/blast in the C++ Toolkit.

An attempt was made to isolate the user of the BLAST API (as exposed in algo/blast/api) from the core of BLAST, so that algorithmic enhancements or refactoring of that code would be transparent to the API programmer as far as that is possible. Since BLAST is continually under development and many of the developments involve new features it is not always possible or desirable to isolate the API programmer from these changes. This chapter will focus on the API for the C++ Toolkit. A few different search classes will be discussed. These include the CLocalBlast class, typically used for searching a query (or queries) against a BLAST database; CRemoteBlast, used for sending searches to the NCBI servers; as well as CBl2Seq, useful for searching target sequences that have not been formatted as a BLAST database.

Chapter Outline

CLocalBlast

The class CLocalBlast can be used for searches that run locally on a machine (as opposed to sending the request over the network to use the CPU of another machine) and search a query (or queries) against a preformatted BLAST database, which holds the target sequence data in a format optimal for BLAST searches. The demonstration program blast_demo.cpp illustrates the use of CLocalBlast. There are a few different CLocalBlast constructors, but they always take three arguments reflecting the need for a query sequence, a set of BLAST options, and a set of target sequences (e.g., BLAST database). First we discuss how to construct these arguments and then we discuss how to access the results.

Query Sequence

The classes that perform BLAST searches expect to be given query sequences in one of a few formats. Each is a container for one or more query sequences expressed as CSeq_loc objects, along with ancillary information. In this document we will only discuss classes that take either a SSeqLoc or a TSeqLocVector, which is just a collection of SSeqLoc's.

CBlastInput is a class that converts an abstract source of sequence data into a format suitable for use by the BLAST search classes. This class may produce either a TSeqLocVector container or a CBlastQueryVector container to represent query sequences. As mentioned above we limit our discussion to the TSeqLocVector class here.

CBlastInput can produce a single container that includes all the query sequences, or can output a batch of sequences at a time (the combined length of the sequences within each batch can be specified) until all of the sequences within the data source have been consumed.

Sources of sequence data are represented by a CBlastInputSource, or a class derived from it. CBlastInput uses these classes to read one sequence at a time from the data source and convert to a container suitable for use by the BLAST search classes.

An example use of CBlastInputSource is CBlastFastaInputSource, which represents a stream containing fasta-formatted biological sequences. Usually this class represents a collection of sequences residing in a text file. One sequence at a time is read from the file and converted into a BLAST input container.

CBlastFastaInputSource uses CBlastInputConfig to provide more control over the file reading process. For example, the read process can be limited to a range of each sequence, or sequence letters that appear in lowercase can be scheduled for masking by BLAST. CBlastInputConfig can be used by other classes to provide the same kind of control, although not all class members will be appropriate for every data source.

Options

The BLAST options classes were designed to allow a programmer to easily set the options to values appropriate to common tasks, but then modify individual options as needed. Table 1 lists the supported tasks.

The CBlastOptionsFactory class offers a single static method to create CBlastOptionsHandle subclasses so that options applicable to all variants of BLAST can be inspected or modified. The actual type of the CBlastOptionsHandle returned by the Create() method is determined by its EProgram argument (see Table 1). The return value of this function is guaranteed to have reasonable defaults set for the selected task.

The CBlastOptionsHandle class encapsulates options that are common to all variants of BLAST, from which more specific tasks can inherit the common options. The subclasses of CBlastOptionsHandle should present an interface that is more specific, i.e.: only contain options relevant to the task at hand, although it might not be an exhaustive interface for all options available for the task. Please note that the initialization of this class' data members follows the template method design pattern, and this should be followed by subclasses also. Below is an example use of the CBlastOptionsHandle to create a set of options appropriate to "blastn" and then to set the expect value to non-default values:

```
using ncbi::blast;

CRef<CBlastOptionsHandle>
```

```
 opts_handle(CBlastOptionsFactory::Create(eBlastn));
opts_handle->SetEvalueThreshold(1e-10);
blast(query, opts_handle, db);
```

The CBlastOptionsHandle classes offers a Validate() method in its interface which is called by the BLAST search classes prior to performing the actual search, but users of the C++ BLAST options APIs might also want to invoke this method to ensure that any exceptions thrown by the BLAST search classes do not originate from an incorrect setting of BLAST options. Please note that the Validate() method throws a CBlastException in case of failure.

If the same type of search (e.g., nucleotide query vs. nucleotide database) will always be performed, then it may be preferable to create an instance of the derived classes of the CBlastOptionsHandle. These classes expose an interface that is relevant to the task at hand, but the popular options can be modified as necessary:

```
using ncbi::blast;

CRef<CBlastNucleotideOptionsHandle> nucl_handle(new
CBlastNucleotideOptionsHandle);
...
nucl_handle->SetTraditionalBlastnDefaults();
nucl_handle->SetStrandOption(objects::eNa_strand_plus);
...
CRef<CBlastOptionsHandle> opts = CRef<CBlastOptionsHandle> (&*nucl_handle);
CLocalBlast blast(query_factory, opts, db);
```

The CBlastOptionsHandle design arranges the BLAST options in a hierarchy. For example all searches that involve protein-protein comparisons (including proteins translated from a nucleotide sequence) are handled by CBlastProteinOptionsHandle or a subclass (e.g., CBlastxOptionsHandle). A limitation of this design is that the introduction of new algorithms or new options that only apply to some programs may violate the class hierarchy. To allow advanced users to overcome this limitation the GetOptions() and SetOptions() methods of the CBlastOptionsHandle hierarchy allow access to the CBlastOptions class, the lowest level class in the C++ BLAST options API which contains all options available to all variants of the BLAST algorithm. No guarantees about the validity of the options are made if this interface is used, therefore invoking Validate() is *strongly* recommended.

Target Sequences

One may specify a BLAST database to search with the CSearchDatabase class. Normally it is only necessary to provide a string for the database name and state whether it is a nucleotide or protein database. It is also possible to specify an entrez query or a vector of GI's that will be used to limit the search.

Results

The Run() method of CLocalBlast returns a CSearchResultSet that may be used to obtain results of the search. The CSearchResultSet class is a random access container of CSearchResults objects, one for each query submitted in the search. The CSearchResult class provides access to alignment (as a CSeq_align_set), the query Cseq_id, warning or error messages that were generated during the run, as well as the filtered query regions (assuming query filtering was set).

CRemoteBlast

The CRemoteBlast class sends a BLAST request to the SPLITD system at the NCBI. This can be advantageous in many situations. There is no need to download the (possibly) large BLAST databases to the user's machine; the search may be spread across many machines by the SPLITD system at the NCBI, making it very fast; and the results will be kept on the NCBI server for 36 hours in case the users wishes to retrieve them again the next day. On the other hand the user must select one of the BLAST databases maintained by the NCBI since it is not possible to upload a custom database for searching. Here we discuss a CRemoteBlast constructor that takes three arguments, reflecting the need for a query sequence(s), a set of BLAST options, and a BLAST database. Readers are advised to read the CLocalBlast section before they read this section.

Query Sequence

A TSeqLocVector should be used as input to CRemoteBlast. Please see the section on CLocalBlast for details.

Options

CBlastOptionsFactory::Create() can again be used to create options for CRemoteBlast. In this case though it is necessary to set the second (default) argument of Create() to CBlastOptions::eRemote.

Target Sequences

One may use the CSearchDatabase class to specify a BLAST database, similar to the method outlined in the CLocalBlast section. In this case it is important to remember though that the user must select from the BLAST databases available on the NCBI Web site and not one built locally.

Results

After construction of the CRemoteBlast object the user should call one of the SubmitSync() methods. After this returns the method GetResultSet() will return a CSearchResultSet which the user can interrogate using the same methods as in CLocalBlast. Additionally the user may obtain the request identifier (RID) issued by the SPLITD system with the method GetRID().

Finally CRemoteBlast provides a constructor that takes a string, which it expects to be an RID issued by the SPLITD system. This RID might have been obtained by an earlier run of CRemoteBlast or it could be one that was obtained from the NCBI SPLITD system via the web page. Note that the SPLITD system will keep results on it's server for 36 hours, so the RID cannot be older than that.

The Uniform Interface

The ISeqSearch class is an abstract interface class. Concrete subclasses can run either local (CLocalSeqSearch) or remote searches (CRemoteSeqSearch). The concrete classes will only perform an intersection of the tasks that CLocalBlast and CRemoteBlast can perform. As an example, there is no method to retrieve a Request identifier (RID) from subclasses of ISeqSearch as this is supported only for remote searches but not for local searches. The methods supported by the concrete subclasses and the return values are similar to those of CLocalBlast and CRemoteBlast.

CBl2Seq

CBl2Seq is a class useful for searching a query (or queries) against one or more target sequences that have not been formatted as a BLAST database. These sequences may, for example, come from a user who pasted them into a web page or be fetched from the Entrez or ID1 services at the NCBI. The CBl2Seq constructors all take three arguments, reflecting the need for a set of query sequences, a set of target sequences, and some information about the BLAST options or program type to use. In this section it is assumed the reader has already read the previous section on CLocalBlast.

The BLAST database holds the target sequence data in a format optimal for BLAST searches, so that if a target sequence is to be searched more than a few times it is best to convert it to a BLAST database and use CLocalBlast.

Query Sequence

The query sequence (or sequences) is represented either as a SSeqLoc (for a single query sequence) or as a TSeqLocVector (in the case of multiple query sequences). The CBlastInput class, described in the CLocalBlast section, can be used to produce a TSeqLocVector.

Options and Program Type

The CBl2Seq constructor takes either an EProgram enum (see Table 1) or CBlastOptionsHandle (see relevant section under CLocalBlast). In the former case the default set of options for the given EProgram are used. In the latter case it is possible for the user to set options to non-default values.

Target Sequences

The target sequence(s) is represented either as a SSeqLoc or TSeqLocVector.

Results

The Run() method of the CBl2Seq class returns a collection of CSeq_align_set's. The method GetMessages() may be used to obtain any error or warning messages generated during the search.

Sample Applications

The following are sample applications that demonstrate the usage of the CBl2Seq and CLocalBlast classes respectively:

- blast_sample.cpp
- blast_demo.cpp

Table 1: List of tasks supported by the CBlastOptionsHandle. “Translated nucleotide” means that the input was nucleotide, but the comparison is based upon the protein. PSSM is a “position-specific scoring matrix”. The “EProgram” can be used as an argument to CBlastOptionsFactory::Create

EProgram (enum)	Default Word-size	Query type	Target type	Notes
eBlastN	11	Nucleotide	Nucleotide	
eMegablast	28	Nucleotide	Nucleotide	Optimized for speed and closely related sequences
eDiscMegablast	11	Nucleotide	Nucleotide	Optimized for cross-species matches
eBlastp	3	Protein	Protein	
eBlastx	3	Translated nucleotide	Protein	
eTblastn	3	Protein	Translated nucleotide	
eTblastx	3	Translated nucleotide	Translated nucleotide	
eRPSBlast	3	Protein	PSSM	Can very quickly identify domains
eRPSTblastn	3	Translated nucleotide	PSSM	
ePSIBlast	3	PSSM	Protein	Extremely sensitive method to find distant homologies
ePHIBlastp	3	Protein	Protein	Uses pattern in query to start alignments

17: Access to NCBI data

Created: January 26, 2009.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter describes access to the NCBI data using the NCBI C++ Toolkit.

Chapter Outline

Object Manager: Generic API for retrieving and manipulating biological sequence data

The information about Object Manager library is here.

E-Utils: Access to Entrez Data

EUtils requests

The base class for all requests is CEUtils_Request. Derived request classes provide *Get/Set* methods to specify arguments for each request. The returned data can be read in several ways:

- ***Read()*** - reads the data returned by the server into a string.
- ***GetStream()*** - allows to read plain data returned by the server.
- ***GetObjectIStream()*** - returns serial stream for reading data (in most cases it's an XML stream).

Connection context

CEUtils_ConnContext allows transferring EUtils context from one request to another. It includes user-provided information (tool, email) and history data (WebEnv, query_key). If no context is provided for a request (the *ctx* argument is *NULL*), a temporary context will be created while executing the request.

EUtils objects

Most requests return specific data types described in EUtils DTDs. The C++ classes generated from the DTDs can be found in include/objtools/eutils/<*util-name*>.

Sample application

An example of using EUtils API can be found in sample/app/eutils/eutils_sample.cpp.

18: Biological Sequence Alignment

Last Update: June 25, 2013.

The Global Alignment Library [xalgoalign:include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The library contains C++ classes encapsulating global pairwise alignment algorithms frequently used in computational biology.

- CNWAligner is the base class for the global alignment algorithm classes. The class provides an implementation of the generic Needleman-Wunsch for computing global alignments of nucleotide and amino acid sequences. The implementation uses an affine scoring scheme. An optional end-space free variant is supported, which is useful in applications where one sequence is expected to align in the interior of the other sequence, or the suffix of one string to align with a prefix of the other.
  The classical Needleman-Wunsch algorithm is known to have memory and CPU requirements of the order of the sequence lengths' product. If consistent partial alignments are available, the problem is split into smaller subproblems taking fewer operations and less space to complete. CNWAligner provides a way to specify such partial alignments (ungapped).
- CBandAligner encapsulates the banded variant of the global alignment algorithm which is applicable when the number of differences in the target alignment is limited ('the band width'). The computational cost of the algorithm is of the order of the band width multiplied by the length of the query sequence.
- CMMAligner follows Hirschberg's divide-and-conquer approach under which the amount of space required to align two sequences globally becomes a linear function of the sequences' lengths. Although the latter is achieved at a cost of up to twice longer running time, a multithreaded version of the algorithm can run even faster than the classical Needleman-Wunsch algorithm in a multiple-CPU environment.
- CSplicedAligner is an abstract base for algorithms computing cDNA-to-genome, or spliced alignments. Spliced alignment algorithms specifically account for splice signals in their dynamic programming recurrences resulting in better alignments for these particular but very important types of sequences.

Chapter Outline

The following is an outline of the chapter topics:

Demo Cases [src/app/nw_aligner] [src/app/splign/]

Computing pairwise global sequence alignments

Generic **pairwise** global alignment functionality is provided by CNWAligner.

NOTE: CNWAligner is not a multiple sequence aligner. An example of using CNWAligner can be seen here.

This functionality is discussed in the following topics:

- Initialization
- Parameters of alignment
- Computing
- Alignment transcript

Initialization

Two constructors are provided to initialize the aligner:

```
CNWAligner(const char* seq1, size_t len1,
 const char* seq2, size_t len2,
 const SNCBIPackedScoreMatrix* scoremat = 0);
CNWAligner(void);
```

The first constructor allows specification of the sequences and the score matrix at the time of the object's construction. Note that the sequences must be in the proper strands, because the aligners do not build reverse complementaries. The last parameter must be a pointer to a properly initialized SNCBIPackedScoreMatrix object or zero. If it is a valid pointer, then the sequences are verified against the alphabet contained in the SNCBIPackedScoreMatrix object, and its score matrix is further used in dynamic programming recurrences. Otherwise, sequences are verified against the IUPACna alphabet, and match/mismatch scores are used to fill in the score matrix.

The default constructor is provided to support reuse of an aligner object when many sequence pairs share the same type and alignment parameters. In this case, the following two functions must be called before computing the first alignment to load the score matrix and the sequences:

```
void SetScoreMatrix(const SNCBIPackedScoreMatrix* scoremat = 0);
void SetSequences(const char* seq1, size_t len1,
 const char* seq2, size_t len2,
 bool verify = true);
```

where the meaning of scoremat is the same as above.

Parameters of alignment

CNWAligner realizes the affine gap penalty model, which means that every gap of length L (with the possible exception of end gaps) contributes Wg+L*Ws to the total alignment score, where Wg is a cost to open the gap and Ws is a cost to extend the gap by one basepair. These two parameters are always in effect when computing sequence alignments and can be set with:

```
void SetWg(TScore value); // set gap opening score
void SetWs(TScore value); // set gap extension score
```

To indicate penalties, both gap opening and gap extension scores are assigned with negative values.

Many applications (such as the shotgun sequence assembly) benefit from a possibility to avoid penalizing end gaps of alignment, because the relevant sequence's ends may not be expected to align. CNWAligner supports this through a built-in end-space free variant controlled with a single function:

```
void SetEndSpaceFree(bool Left1, bool Right1, bool Left2, bool Right2);
```

The first two arguments control the left and the right ends of the first sequence. The other two control the second sequence's ends. True value means that end spaces will not be penalized. Although an arbitrary combination of end-space free flags can be specified, judgment should be used to get plausible alignments.

The following two functions are only meaningful when aligning nucleotide sequences:

```
void SetWm(TScore value); // set match score
void SetWms(TScore value); // set mismatch score
```

The first function sets a bonus associated with every matching pair of nucleotides. The second function assigns a penalty for every mismatching aligned pair of nucleotides. It is important that values set with these two functions will only take effect after SetScoreMatrix() is called (with a zero pointer, which is the default).

One thing that could limit the scope of global alignment applications is that the classical algorithm takes quadratic space and time to evaluate the alignment. One wayto deal with it is to use the linear-space algorithm encapuslated in CMMAligner. However, when some pattern of alignment is known or desired, it is worthwhile to explicitly specify "mile posts" through which the alignment should pass. Long high-scoring pairs with 100% identity (no gaps or mismatches) are typically good candidates for them. From the algorithmic point of view, the pattern splits the dynamic programming table into smaller parts, thus alleviating space and CPU requirements. The following function is provided to let the aligner know about such guiding constraints:

```
void SetPattern(const vector<size_t>& pattern);
```

Pattern is a vector of hits specified by their zero-based coordinates, as in the following example:

```
// the last parameter omitted to indicate nucl sequences
CNWAligner aligner (seq1, len1, seq2, len2);
// we want coordinates [99,119] and [129,159] on seq1 be aligned
// with [1099,1119] and [10099,10129] on seq2.
const size_t hits [] = { 99, 119, 1099, 1119, 129, 159, 10099, 10129 };
vector<size_t> pattern ( hits, hits + sizeof(hits)/sizeof(hits[0]) );
aligner.SetPattern(pattern);
```

Computing

To start computations, call Run(), which returns the overall alignment score having aligned the sequences. Score is a scalar value associated with the alignment and depends on the parameters of the alignment. The global alignment algorithms align two sequences so that the score is the maximum over all possible alignments.

Alignment transcript

The immediate output of the global alignment algorithms is a transcript.The transcript serves as a basic representation of alignments and is simply a string of elementary commands transforming the first sequence into the second one on a per-character basis. These commands (transcript characters) are (M)atch, (R)eplace, (I)nsert, and (D)elete. For example, the alignment

```
TTC-ATCTCTAAATCTCTCTCATATATATCG
||| |||||| |||| || ||| ||||
TTCGATCTCT-----TCTC-CAGATAAATCG
```

has a transcript:

```
MMMIMMMMMMDDDDDMMMMDMMRMMMRMMMM
```

Several functions are available to retrieve and analyze the transcript:

```
// raw transcript
const vector<ETranscriptSymbol>* GetTranscript(void) const
{
 return &m_Transcript;
}
// converted transcript vector
void GetTranscriptString(vector<char>* out) const;
// transcript parsers
size_t GetLeftSeg(size_t* q0, size_t* q1,
 size_t* s0, size_t* s1,
 size_t min_size) const;
size_t GetRightSeg(size_t* q0, size_t* q1,
 size_t* s0, size_t* s1,
 size_t min_size) const;
size_t GetLongestSeg(size_t* q0, size_t* q1,
 size_t* s0, size_t* s1) const;
```

The last three functions search for a continuous segment of matching characters and return it in sequence coordinates through q0, q1, s0, s1.

The alignment transcript is a simple yet complete representation of alignments that can be used to evaluate virtually every characteristic or detail of any particular alignment. Some of them, such as the percent identity or the number of gaps or mismatches, could be easily restored from the transcript alone, whereas others, such as the scores for protein alignments, would require availability of the original sequences.

Computing multiple sequence alignments

COBALT (COnstraint Based ALignment Tool) is an experimental multiple alignment algorithm whose basic idea was to leverage resources at NCBI, then build up a set of pairwise constraints, then perform a fairly standard iterative multiple alignment process (with many tweaks driven by various benchmarks).

COBALT is available online at:

https://www.ncbi.nlm.nih.gov/tools/cobalt/

A precompiled binary, with the data files needed to run it, is available at:

ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/cobalt/

Work is being done on an improved COBALT tool.

The paper reference for this algorithm is:

J.S. Papadopoulos, R. Agarwala, "COBALT: Constraint-Based Alignment Tool for Multiple Protein Sequences". Bioinformatics, May 2007

Aligning sequences in linear space

CMMAligner is an interface to a linear space variant of the global alignment algorithm. This functionality is discussed in the following topics:

- The idea of the algorithm
- Implementation

The idea of the algorithm

That the classical global alignment algorithm requires quadratic space could be a serious restriction in sequence alignment. One way to deal with it is to use alignment patterns. Another approach was first introduced by Hirschberg and became known as a divide-and-conquer strategy. At a coarse level, it suggests computing of scores for partial alignments starting from two opposite corners of the dynamic programming matrix while keeping only those located in the middle rows or columns. After the analysis of the adjacent scores, it is possible to determine cells on those lines through which the global alignment's back-trace path will go. This approach reduces space to linear while only doubling the worst-case time bound. For details see, for example, Dan Gusfield's "Algorithms on Strings, Trees and Sequences".

Implementation

CMMAligner inherits its public interface from CNWAligner. The only additional method allows us to toggle multiple-threaded versions of the algorithm.

The divide-and-conquer strategy suggests natural parallelization, where blocks of the dynamic programming matrix are evaluated simultaneously. A theoretical acceleration limit imposed by the current implementation is 0.5. To use multiple-threaded versions, call

EnableMultipleThreads(). The number of simultaneously running threads will not exceed the number of CPUs installed on your system.

When comparing alignments produced with the linear-space version with those produced by CNWAligner, be ready to find many of them similar, although not exactly the same. This is normal, because several optimal alignments may exist for each pair of sequences.

Computing spliced sequences alignments

This functionality is discussed in the following topics:

- The problem
- Implementation

The problem

The spliced sequence alignment arises as an attempt to address the problem of eukaryotic gene structure recognition. Tools based on spliced alignments exploit the idea of comparing genomic sequences to their transcribed and spliced products, such as mRNA, cDNA, or EST sequences. The final objective for all splice alignment algorithms is to come up with a combination of segments on the genomic sequence that:

- makes up a sequence very similar to the spliced product, when the segments are concatenated; and
- satisfies certain statistically determined conditions, such as consensus splice sites and lengths of introns.

According to the classical eukaryotic transcription and splicing mechanism, pieces of genomic sequence do not get shuffled. Therefore, one way of revealing the original exons could be to align the spliced product with its parent gene globally. However, because of the specificity of the process in which the spliced product is constructed, the generic global alignment with the affine penalty model may not be enough. To address this accurately, dynamic programming recurrences should specifically account for introns and splice signals.

Algorithms described in this chapter exploit this idea and address a refined splice alignment problem presuming that:

- the genomic sequence contains only one location from which the spliced product could have originated;
- the spliced product and the genomic sequence are in the plus strand; and
- the poly(A) tail and any other chunks of the product not created through the splicing were cut off, although a moderate level of sequencing errors on genomic, spliced, or both sequences is allowed.

In other words, the library classes provide basic splice alignment algorithms to be used in more sophisticated applications. One real-life application, Splign, can be found under demo cases for the library.

Implementation

There is a small hierarchy of three classes involved in spliced alignment facilitating a quality/performance trade-off in the case of distorted sequences:

- CSplicedAligner - abstract base for spliced aligners.
- CSplicedAligner16 - accounts for the three conventional splices (GT/AG, GC/AG, AT/AC) and a generic splice; uses 2 bytes per back-trace matrix cell. Use this class with high-quality genomic sequences.

- CSplicedAligner32 - accounts for the three conventionals and splices that could be produced by damaging bases of any conventional; uses 4 bytes per back-trace matrix cell. Use this class with distorted genomic sequences.

The abstract base class for spliced aligners, CNWSplicedAligner, inherites an interface from its parent, CNWAligner, adding support for two new parameters: intron penalty and minimal intron size (the default is 50).

All classes assume that the spliced sequence is the first of the two input sequences passed. By default, the classes do not penalize gaps at the ends of the spliced sequence. The default intron penalties are chosen so that the 16-bit version is able able to pick out short exons, whereas the 32-bit version is generally more conservative.

As with the generic global alignment, the immediate output of the algorithms is the alignment transcript. For the sake of spliced alignments, the transcript's alphabet is augmented to accommodate introns as a special sequence-editing operation.

Formatting computed alignments

This functionality is discussed in the following topics:

- Formatter object

Formatter object

CNWFormatter is a single place where all different alignment representations are created. The only argument to its constructor is the aligner object that actually was or will be used to align the sequences.

The alignment must be computed before formatting. If the formatter is unable to find the computed alignment in the aligner that was referenced to the constructor, an exception will be thrown.

To format the alignment as a CSeq_align structure, call

```
void AsSeqAlign(CSeq_align* output) const;
```

To format it as text, call

```
void AsText(string* output, ETextFormatType type, size_t line_width = 100)
```

Supported text formats and their ETextFormatType constants follow:

- Type 1 (eFormatType1):
  ```
  TTC-ATCTCTAAATCTCTCTCATATATATCG
  TTCGATCTCT-----TCTC-CAGATAAATCG
             ^  ^
  ```
- Type 2 (eFormatType2):
  ```
  TTC-ATCTCTAAATCTCTCTCATATATATCG
  ||| ||||||     |||| || ||| ||||
  TTCGATCTCT-----TCTC-CAGATAAATCG
  ```
- Gapped FastA (eFormatFastA):
  ```
  >SEQ1
  TTC-ATCTCTAAATCTCTCTCATATATATCG
  ```

```
>SEQ2
TTCGATCTCT-----TCTC-CAGATAAATCG
```

- Table of exons (eFormatExonTable) - spliced alignments only. The exons are listed from left to right in tab-separated columns. The columns represent sequence IDs, alignment lengths, percent identity, coordinates on the query (spliced) and the subject sequences, and a short annotation including splice signals.
- Extended table of exons (eFormatExonTableEx) - spliced alignments only. In addition to the nine columns, the full alignment transcript is listed for every exon.
- ASN.1 (eFormatASN)

19: GUI and Graphics

Last Update: May 16, 2011.

The following approaches to developing GUI applications have been proved to work reasonably well:

Using wxWidgets (for GUI) and OpenGL (for graphics)

This approach is appropriate for projects requiring complex GUIs with rich user interactivity and layered event models.

wxWidgets has a heavier API than FOX, but is not more resource intensive (it uses the underlying system's native rendering toolkit). It offers a GUI builder, support for automated code generation, and a carefully designed event model that makes it a much more capable solution if your application needs extend beyond a dialog-based application with multiple controls. It additionally offers substantial support for OpenGL. Also, its installations are maintained in NCBI for a variety of OS's.

This approach is used in NCBI by the Cn3D application, and the Genome Workbench application is based on that too. Please see the wxWidgets and OpenGL websites for further information.

Using FOX as a third party package

This approach is appropriate for projects requiring uniform behavior across platforms (i.e. not a native look-and-feel).

FOX is very fast, with compact executables. The API is convenient and consistent, with a complete set of widgets. There is an extremely rich set of layout managers, which is very flexible and fast.

This approach is used in NCBI by the taskedit application. Please see the FOX website for further information.

Using the Genome Workbench wxWidgets-based GUI framework

This approach currently may not be appropriate for projects other than the Genome Workbench due to its complexity.

The Genome Workbench project has developed an advanced wxWidgets-based GUI framework - somewhat skewed to dealing with NCBI ASN.1 data model representations. The core framework offers a set of widget extensions and signalling libraries on top of wxWidgets. It also uses DialogBlocks as a GUI RAD development tool. The Genome Workbench project homepage has links for downloading various binaries and the source code. Note: This code makes extensive use of the Object Manager and has very specific build requirements - both of which are difficult to configure correctly and neither of which will be documented in the near future (i.e. use at your own risk).

20: Using the Boost Unit Test Framework

Last Update: July 2, 2013.

Overview

The overview for this chapter consists of the following topics:

Introduction

This chapter discusses the Boost Unit Test Framework and how to use it within NCBI. The NCBI C++ Toolkit has incorporated and extended the open source Boost.Test Library, and provides a simplified way for the developers to create Boost-based C++ unit tests.

The NCBI extensions add the ability to:

- execute the code in a standard (*CNcbiApplication* -like) environment;
- disable test cases or suites, using one of several methods;
- establish dependencies between test cases and suites;
- use NCBI command-line argument processing;
- add initialization and finalization functions; and
- use convenience macros for combining NO_THROW with other test tools.

While the framework may be of interest to outside organizations, this chapter is intended for NCBI C++ developers. See also the Doxygen documentation for tests.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Why Use the Boost Unit Test Framework?

"*...I would like to see a practical plan for every group in Internal Services to move toward standardized testing. Then, in addition to setting an example for the other coding groups, I hope that you will have guidance for them as well about how best to move ahead in this direction. Once you have that, and are adhering to it yourselves, I will start pushing the other coding groups in that direction.*"

- Jim Ostell, April 21, 2008

The value of unit testing is clearly recognized at the highest levels of management at NCBI. Here are some of the ways that using the Boost Unit Test Framework will directly benefit the developer:

- The framework provides a uniform (and well-supported) testing and reporting environment.
- Using the framework simplifies the process of creating and maintaining unit tests:
  - The framework helps keep tests well-structured, straightforward, and easily expandable.
  - You can concentrate on the testing of your functionality, while the framework takes care of all the testing infrastructure.
- The framework fits into the NCBI nightly build system:
  - All tests are run nightly on many platforms.
  - All results are archived and available through a web interface.

How to Use the Boost Unit Test Framework

This chapter assumes you are starting from a working Toolkit source tree. If not, please refer to the chapters on obtaining the source code, and configuring and building the Toolkit.

Creating a New Unit Test

On UNIX or MS Windows, use the new_project script to create a new unit test project:

```
new_project <name> app/unit_test
```

For example, to create a project named foo, type this in a command shell:

```
new_project foo app/unit_test
```

This creates a directory named foo and then creates two projects within the new directory. One project will be the one named on the command-line (e.g. foo) and will contain a sample unit test using all the basic features of the Boost library. The other project will be named unit_test_alt_sample and will contain samples of advanced techniques not required in most unit tests.

You can build and run these projects immediately to see how they work:

```
cd foo
make
make check
```

Once your unit test is created, you must customize it to meet your testing requirements. This involves editing these files:

File	Purpose
Makefile	Main makefile for this directory - builds both the foo and unit_test_alt_sample unit tests.
Makefile.builddir	Contains the path to a pre-built C++ Toolkit.
Makefile.foo_app	Makefile for the foo unit test.
Makefile.in	
Makefile.unit_test_alt_sample_app	Makefile for the unit_test_alt_sample unit test.
foo.cpp	Source code for the foo unit test.
unit_test_alt_sample.cpp	Source code for the unit_test_alt_sample unit test.
unit_test_alt_sample.ini	Configuration file for the unit_test_alt_sample unit test.

Customizing an Existing Unit Test

This section contains the following topics:

Modifying the Makefile

The new_project script generates a new unit test project that includes everything needed to use the Boost Unit Test Framework, but it won’t include anything specifically needed to build the library or application you are testing.

Therefore, edit the unit test makefile (e.g. Makefile.foo.app) and add the appropriate paths and libraries needed by your library or application. Note that although the new_project script creates five makefiles, you will generally need to edit only one. If you are using Windows, please see the FAQ on adding libraries to Visual C++ projects.

Because the unit tests are based on the Boost Unit Test Framework, the makefiles must specify:

```
REQUIRES = Boost.Test.Included
```

If you are using the new_project script (recommended), this setting is included automatically. Otherwise, make sure that Boost.Test.Included is listed in REQUIRES.

Note: Please also see the "Defining and running tests" section for unit test makefile information that isn't specific to Boost.

Modifying the Source File

A unit test is simply a test of a unit of code, such as a class. Because each unit has many requirements, each unit test has many test cases. Your unit test code should therefore consist of a test case for each testable requirement. Each test case should be as small and independent of other test cases as possible. For information on how to handle dependencies between test cases, see the section on managing dependencies.

Starting with an existing unit test source file, simply add, change, or remove test cases as appropriate for your unit test. Test cases are defined by the BOOST_AUTO_TEST_CASE macro, which looks similar to a function. The macro has a single argument (the test case name) and a block of code that implements the test. Test case names must be unique at each level of the test suite hierarchy (see managing dependencies). Test cases should contain code that will succeed if the requirement under test is correctly implemented, and fail otherwise. Determination of success is made using Boost testing tools such as BOOST_REQUIRE and BOOST_CHECK.

The following sections discuss modifying the source file in more detail:

- Using Testing Tools
- Adding Initialization and/or Finalization
- Handling Timeouts
- Handling Command-Line Arguments in Test Cases
- Creating Test Suites
- Managing Dependencies
- Unit Tests with Multiple Files

Using Testing Tools

Testing tools are macros that are used to detect errors and determine whether a given test case passes or fails.

While at a basic level test cases can pass or fail, it is useful to distinguish between those failures that make subsequent testing pointless or impossible and those that don’t. Therefore, there are two levels of testing: CHECK (which upon failure generates an error but allows subsequent testing to continue), and REQUIRE (which upon failure generates a fatal error and aborts the current test case). In addition, there is a warning level, WARN, that can report something of interest without generating an error, although by default you will have to set a command-line argument to see warning messages.

If the failure of one test case should result in skipping another then you should add a dependency between them.

Many Boost testing tools have variants for each error level. The most common Boost testing tools are:

Testing Tool	Purpose
BOOST_<level>(predicate)	Fails if the Boolean predicate (any logical expression) is false.
BOOST_<level>_EQUAL(left, right)	Fails if the two values are not equal.
BOOST_<level>_THROW(expression, exception)	Fails if execution of the expression doesn't throw an exception of the given type (or one derived from it).
BOOST_<level>_NO_THROW(expression)	Fails if execution of the expression throws any exception.

Note that BOOST_<level>_EQUAL(var1,var2) is equivalent to BOOST_<level>(var1==var2), but in the case of failure it prints the value of each variable, which can be helpful. Also, it is not a good idea to compare floating point values directly - instead, use BOOST_<level>_CLOSE(var1,var2,tolerance).

See the Boost testing tools reference page for documentation on these and other testing tools.

The NCBI extensions to the Boost library add a number of convenience testing tools that enclose the similarly-named Boost testing tools in a NO_THROW test:

Boost Testing Tool	NCBI "NO_THROW " Extension
BOOST_<level>(predicate)	NCBITEST_<level>(predicate)
BOOST_<level>_EQUAL(left, right)	NCBITEST_<level>_EQUAL(left, right)
BOOST_<level>_NE(left, right)	NCBITEST_<level>_NE(left, right)
BOOST_<level>_MESSAGE(pred, msg)	NCBITEST_<level>_MESSAGE(pred, msg)

Note: Testing tools are only supported within the context of test cases. That is, within functions defined by the BOOST_AUTO_TEST_CASE macro and within functions called by a test case. They are not supported in functions defined by the NCBITEST_* macros.

Adding Initialization and/or Finalization

If your unit test requires initialization prior to executing test cases, or if finalization / clean-up is necessary, use these functions:

```
NCBITEST_AUTO_INIT()
{
 // Your initialization code here...
}

NCBITEST_AUTO_FINI()
{
 // Your finalization code here...
}
```

Handling Timeouts

If exceeding a maximum execution time constitutes a failure for your test case, use this:

```
// change the second parameter to the duration of your timeout in seconds
BOOST_AUTO_TEST_CASE_TIMEOUT(TestTimeout, 3);
BOOST_AUTO_TEST_CASE(TestTimeout)
{
 // Your test code here...
}
```

Handling Command-Line Arguments in Test Cases

It is possible to retrieve command-line arguments from your test cases using the standard C++ Toolkit argument handling API. The first step is to initialize the unit test to expect the arguments. Add code like the following to your source file:

```
NCBITEST_INIT_CMDLINE(descrs)
{
 // Add calls like this for each command-line argument to be used.
 descrs->AddOptionalPositional("some_arg",
 "Sample command-line argument.",
 CArgDescriptions::eString);
}
```

For more examples of argument processing, see test_ncbiargs_sample.cpp.

Next, add code like the following to access the argument from within a test case:

```
BOOST_AUTO_TEST_CASE(TestCaseName)
{
 const CArgs& args = CNcbiApplication::Instance()->GetArgs();
 string arg_value = args["some_arg"].AsString();
 // do something with arg_value ...
}
```

Adding your own command-line arguments will not affect the application's ability to process other command-line arguments such as -help or -dryrun.

Creating Test Suites

Test suites are simply groups of test cases. The test cases included in a test suite are those that appear between the beginning and ending test suite declarations:

```
BOOST_AUTO_TEST_SUITE(TestSuiteName)

BOOST_AUTO_TEST_CASE(TestCase1)
{
 //...
}

BOOST_AUTO_TEST_CASE(TestCase2)
{
 //...
}

BOOST_AUTO_TEST_SUITE_END();
```

Note that the beginning test suite declaration defines the test suite name and does not include a semicolon.

Managing Dependencies

Test cases and suites can be dependent on other test cases or suites. This is useful when it doesn’t make sense to run a test after some other test fails:

```
NCBITEST_INIT_TREE()
{
 // define individual dependencies
 NCBITEST_DEPENDS_ON(test_case_dep, test_case_indep);
 NCBITEST_DEPENDS_ON(test_case_dep, test_suite_indep);
 NCBITEST_DEPENDS_ON(test_suite_dep, test_case_indep);
 NCBITEST_DEPENDS_ON(test_suite_dep, test_suite_indep);

 // define multiple dependencies
 NCBITEST_DEPENDS_ON_N(item_dep, 2, (item_indep1, item_indep2));
}
```

When an independent test item (case or suite) fails, all of the test items that depend on it will be skipped.

Unit Tests with Multiple Files

The new_project script is designed to create single-file unit tests by default, but you can add as many files as necessary to implement your unit test. Use of the BOOST_AUTO_TEST_MAIN macro is now deprecated.

Disabling Tests

The Boost Unit Test Framework was extended by NCBI to provide several ways to disable test cases and suites. Test cases and suites are disabled based on logical expressions in the application configuration file or, less commonly, by explicitly disabling or skipping them. The logical expressions are based on unit test variables which are defined either by the library or by the user. All such variables are essentially Boolean in that they are either defined (true) or not defined (false). Note: these methods of disabling tests don't apply if specific tests are run from the command-line.

- Disabling Tests with Configuration File Entries
- Library-Defined Variables
- User-Defined Variables
- Disabling or Skipping Tests Explicitly in Code

Disabling Tests with Configuration File Entries

The [UNITTESTS_DISABLE] section of the application configuration file can be customized to disable test cases or suites. Entries in this section should specify a test case or suite name and a logical expression for disabling it (expressions that evaluate to true disable the test). The logical expression can be formed from the logical constants true and false, numeric constants, library-defined or user-defined unit test variables, logical operators ('!', '&&', and '||'), and parentheses.

To disable specific tests, use commands like:

```
[UNITTESTS_DISABLE]
SomeTestCaseName = OS_Windows && PLATFORM_BigEndian
SomeTestSuiteName = (OS_Linux || OS_Solaris) && COMPILER_GCC
```

There is a special entry GLOBAL that can be used to disable all tests. For example, to disable all tests under Cygwin, use:

```
[UNITTESTS_DISABLE]
GLOBAL = OS_Cygwin
```

Note: If the configuration file contains either a test name or a variable name that has not been defined (e.g. due to a typo) then the test program will exit immediately with an error, without executing any tests.

Library-Defined Variables

When the NCBI-extended Boost Test library is built, it defines a set of unit test variables based on the build, compiler, operating system, and platform. See Table 1 for a list of related variables (test_boost.cpp has the latest list of variables).

At run-time, the library also checks the FEATURES environment variable and creates unit test variables based on the current set of features. See Table 2 for a list of feature, package, and project related variables (test_boost.cpp has the latest list of features).

The automated nightly test suite defines the FEATURES environment variable before launching the unit test applications. In this way, unit test applications can also use run-time detected features to exclude specific tests from the test suite.

Note: The names of the features are modified slightly when creating unit test variables from names in the FEATURES environment variable. Specifically, each feature is prefixed by FEATURE_ and all non-alphanumeric characters are changed to underscores. For example, to require the feature in-house-resources for a test (i.e. to disable the test if the feature is not present), use:

```
[UNITTESTS_DISABLE]
SomeTestCaseName = !FEATURE_in_house_resources
```

User-Defined Variables

You can define your own variables to provide finer control on disabling tests. First, define a variable in your source file:

```
NCBITEST_INIT_VARIABLES(parser)
{
 parser->AddSymbol("my_ini_var", <some bool expression goes here>);
}
```

Then add a line to the configuration file to disable a test based on the value of the new variable:

```
[UNITTESTS_DISABLE]
MyTestName = my_ini_var
```

User-defined variables can be used in conjunction with command-line arguments:

```
NCBITEST_INIT_VARIABLES(parser)
{
 const CArgs& args = CNcbiApplication::Instance()->GetArgs();
 parser->AddSymbol("my_ini_var", args["my_arg"].HasValue());
}
```

Then, passing the argument on the command-line controls the disabling of the test case:

```
./foo my_arg # test is disabled
./foo # test is not disabled (at least via command-line / config file)
```

Disabling or Skipping Tests Explicitly in Code

The NCBI extensions include a macro, NCBITEST_DISABLE, to unconditionally disable a test case or suite. This macro must be placed in the NCBITEST_INIT_TREE function:

```
NCBITEST_INIT_TREE()
{
 NCBITEST_DISABLE(test_case_name);
 NCBITEST_DISABLE(test_suite_name);
}
```

The extensions also include two functions for globally disabling or skipping all tests. These functions should be called only from within the NCBITEST_AUTO_INIT or NCBITEST_INIT_TREE functions:

```
NCBITEST_INIT_TREE()
{
 NcbiTestSetGlobalDisabled(); // A given unit test might include one
 NcbiTestSetGlobalSkipped(); // or the other of these, not both.
 // Most unit tests won't use either.
}
```

The difference between these functions is that globally disabled unit tests will report the status DIS to check scripts while skipped tests will report the status SKP.

Viewing Unit Tests Results from the Nightly Build

The Boost Unit Test Framework provides more than just command-line testing. Each unit test built with the framework becomes incorporated into nightly testing and is tested on multiple platforms and under numerous configurations. All such results are archived in the database and available through a web interface.

The main page (see Figure 1) provides many ways to narrow down the vast quantity of statistics available. The top part of the page allows you to select test date, test result, build configuration (branch, compiler, operating system, etc), debug/release, and more. The page also has a column for selecting tests, and a column for configurations. For best results, refine the selection as much as possible, and then click on the “See test statistics” button.

The “See test statistics” button retrieves the desired statistics in a second page (see Figure 2). The results are presented in tables: one for each selected date, with unit tests down the left side and configurations across the top. Further refinements of the displayed results can be made by removing rows, columns, or dates; and by selecting whether all columns, all cells, or only selected cells are displayed.

Each cell in the results tables represents a specific unit test performed on a specific date under a specific configuration. Clicking on a cell retrieves a third page (see Figure 3) that shows information about that test and its output.

Running Unit Tests from a Command-Line

To run one or more selected test cases from a command-line, use this:

```
./foo --run_test=TestCaseName1,TestCaseName2
```

Multiple test cases can be selected by using a comma-separated list of names.

To see all test cases in a unit test, use this:

```
./foo -dryrun
```

To see exactly which test cases passed and failed, use this:

```
./foo --report_level=detailed
```

To see warning messages, use this:

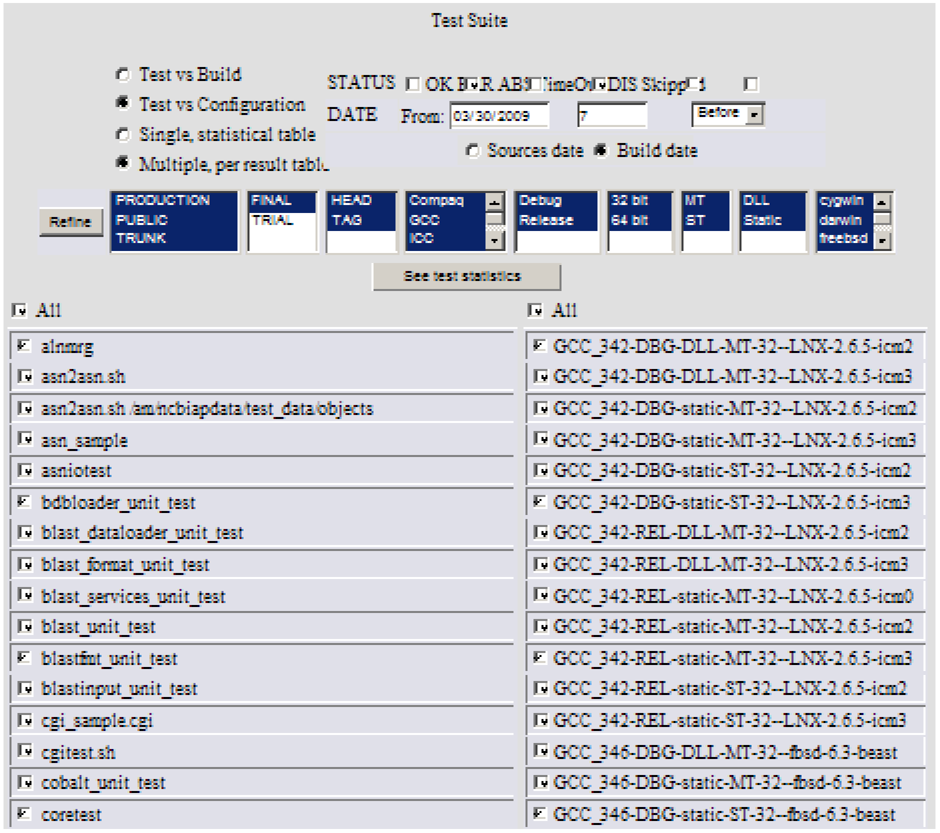
```
./foo --log_level=warning
```

Additional runtime parameters can be set. For a complete list, see the online documentation.

Limitations of the Boost Unit Test Framework

The currently known limitations are:

- It is not suitable for most multi-threaded tests.
- It is not suitable for "one-piece" applications (such as server or CGI). Such applications should be tested via their clients (which would preferably be unit test based).



Test Suite

Test vs Build [✓ Test vs Configuration] Single, statistical table [✓ Multiple, per result tabl]

STATUS OK ERR ABS TimeOut DIS Skipped

DATE From: 03/30/2009 7 Before

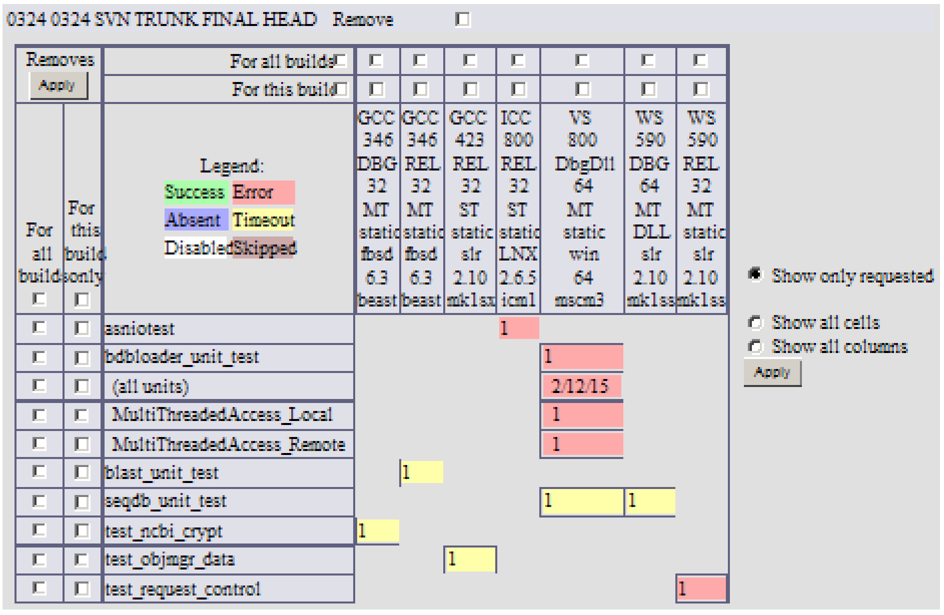
Sources date [✓ Build date]

Refine	PRODUCTION	FINAL	HEAD	Compaq	Debug	32 bit	MT	DLL	cygwin
	PUBLIC	TRIAL	TAG	GCC	Release	64 bit	ST	Static	darwin
	TRUNK			ICC					freebsd

See test statistics

✓ All	✓ All
✓ alnmrg	✓ GCC_342-DBG-DLL-MT-32--LNX-2.6.5-icm2
✓ asn2asn.sh	✓ GCC_342-DBG-DLL-MT-32--LNX-2.6.5-icm3
✓ asn2asn.sh /am/ncbiapdata/test_data/objects	✓ GCC_342-DBG-static-MT-32--LNX-2.6.5-icm2
✓ asn_sample	✓ GCC_342-DBG-static-MT-32--LNX-2.6.5-icm3
✓ asniotest	✓ GCC_342-DBG-static-ST-32--LNX-2.6.5-icm2
✓ bdbloader_unit_test	✓ GCC_342-DBG-static-ST-32--LNX-2.6.5-icm3
✓ blast_dataloader_unit_test	✓ GCC_342-REL-DLL-MT-32--LNX-2.6.5-icm2
✓ blast_format_unit_test	✓ GCC_342-REL-DLL-MT-32--LNX-2.6.5-icm3
✓ blast_services_unit_test	✓ GCC_342-REL-static-MT-32--LNX-2.6.5-icm0
✓ blast_unit_test	✓ GCC_342-REL-static-MT-32--LNX-2.6.5-icm2
✓ blastfmt_unit_test	✓ GCC_342-REL-static-MT-32--LNX-2.6.5-icm3
✓ blastinput_unit_test	✓ GCC_342-REL-static-ST-32--LNX-2.6.5-icm2
✓ cgi_sample.cgi	✓ GCC_342-REL-static-ST-32--LNX-2.6.5-icm3
✓ cgitest.sh	✓ GCC_346-DBG-DLL-MT-32--fbsd-6.3-beast
✓ cobalt_unit_test	✓ GCC_346-DBG-static-MT-32--fbsd-6.3-beast
✓ coretest	✓ GCC_346-DBG-static-ST-32--fbsd-6.3-beast

Figure 1. Test Interface

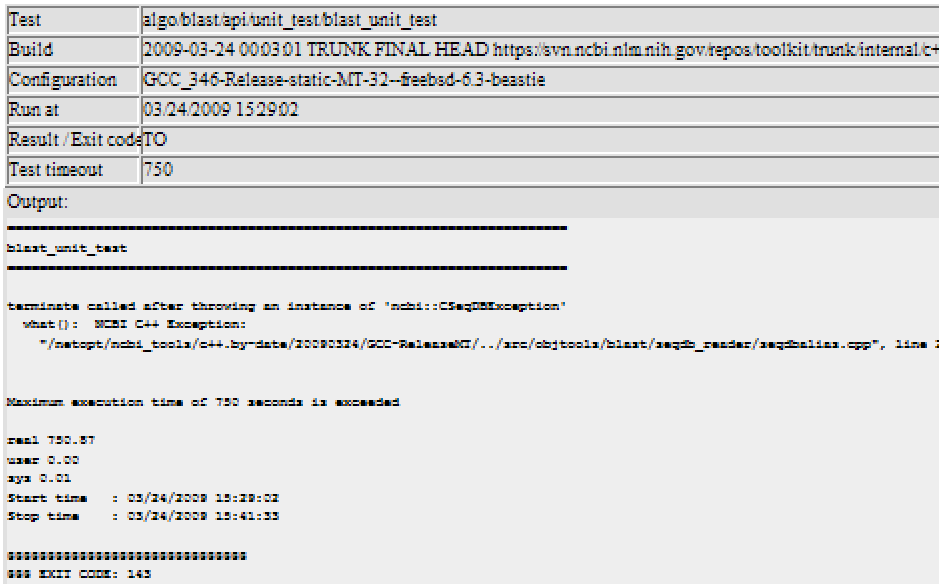


0324 0324 SVN TRUNK FINAL HEAD Remove

Removes Apply		For all builds							
		For this build							
For all builds	For this build only	Legend: 🟩 Success 🟥 Error 🟦 Absent 🟨 Timeout ⬜ Disabled ▨ Skipped	GCC 346 DBG 32 MT static fbsd 6.3 beast	GCC 346 REL 32 MT static fbsd 6.3 beast	GCC 423 REL 32 ST static slr 2.10 mk1sx	ICC 800 REL 32 ST static LNX 2.6.5 icm1	VS 800 DbgDll 64 MT static win 64 mscm3	WS 590 DBG 64 MT DLL slr 2.10 mk1ss	WS 590 REL 32 MT static slr 2.10 mk1ss
		asniotest				1			
		bdbloader_unit_test					1		
		(all units)					2/12/15		
		MultiThreadedAccess_Local					1		
		MultiThreadedAccess_Remote					1		
		blast_unit_test		1					
		seqdb_unit_test					1	1	
		test_ncbi_crypt	1						
		test_objmgr_data			1				
		test_request_control							1

[✓ Show only requested] Show all cells Show all columns Apply

Figure 2. Test Matrix



Test	algo/blast/api/unit_test/blast_unit_test
Build	2009-03-24 00:03:01 TRUNK FINAL HEAD https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/internal/c+
Configuration	GCC_346-Release-static-MT-32--freebsd-6.3-beastie
Run at	03/24/2009 15:29:02
Result / Exit code	TO
Test timeout	750

Output:

```
======================================================================
blast_unit_test
======================================================================

terminate called after throwing an instance of 'ncbi::CSeqDBException'
  what():  NCBI C++ Exception:
    "/netopt/ncbi_tools/c++.by-date/20090324/GCC-ReleaseMT/../src/objtools/blast/seqdb_reader/seqdbalias.cpp", line 


Maximum execution time of 750 seconds is exceeded

real 750.57
user 0.00
sys 0.01
Start time   : 03/24/2009 15:29:02
Stop time    : 03/24/2009 15:41:33

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$ EXIT CODE: 143
```

Figure 3. Test Result

Table 1. Build Generated Predefined Variables

Builds	**Compilers**	**Operating Systems**	**Platforms**
BUILD_Debug	COMPILER_Compaq	OS_AIX	PLATFORM_BigEndian
BUILD_Dll	COMPILER_GCC	OS_BSD	PLATFORM_Bits32
BUILD_Release	COMPILER_ICC	OS_Cygwin	PLATFORM_Bits64
BUILD_Static	COMPILER_KCC	OS_Irix	PLATFORM_LittleEndian
	COMPILER_MipsPro	OS_Linux	
	COMPILER_MSVC	OS_MacOS	
	COMPILER_VisualAge	OS_MacOSX	
	COMPILER_WorkShop	OS_Solaris	
		OS_Tru64	
		OS_Unix	
		OS_Windows	

Table 2. Check Script Generated Predefined Variables

Features	**Packages**	**Projects**
AIX	BerkeleyDB	algo
BSD	BerkeleyDB__ (use for BerkeleyDB++)	app
CompaqCompiler	Boost_Regex	bdb
Cygwin	Boost_Spirit	cgi
CygwinMT	Boost_Test	connext
DLL	Boost_Test_Included	ctools
DLL_BUILD	Boost_Threads	dbapi
Darwin	BZ2	gbench
GCC	C_ncbi	gui
ICC	C_Toolkit	local_bsm
in_house_resources	CPPUNIT	ncbi_crypt
IRIX	DBLib	objects
KCC	EXPAT	serial
Linux	Fast_CGI	
MIPSpro	FLTK	
MSVC	FreeTDS	
MSWin	FreeType	
MT	FUSE	
MacOS	GIF	
Ncbi_JNI	GLUT	
OSF	GNUTLS	
PubSeqOS	HDF5	
SRAT_internal	ICU	
Solaris	JPEG	
unix	LIBXML	
VisualAge	LIBXSLT	
WinMain	LocalBZ2	
WorkShop	LocalMSGMAIL2	
XCODE	LocalNCBILS	
	LocalPCRE	
	LocalSSS	
	LocalZ	
	LZO	

	MAGIC	
	MESA	
	MUPARSER	
	MySQL	
	NCBILS2	
	ODBC	
	OECHEM	
	OpenGL	
	OPENSSL	
	ORBacus	
	PCRE	
	PNG	
	PYTHON	
	PYTHON23	
	PYTHON24	
	PYTHON25	
	SABLOT	
	SGE	
	SP	
	SQLITE	
	SQLITE3	
	SQLITE3ASYNC	
	SSSDB	
	SSSUTILS	
	Sybase	
	SybaseCTLIB	
	SybaseDBLIB	
	TIFF	
	UNGIF	
	UUID	
	Xalan	
	Xerces	
	XPM	
	Z	
	wx2_8	

	wxWidgets	
	wxWindows	

Part 4: Wrappers for 3rd-Party Packages

Part 4 discusses NCBI wrappers for 3rd-party packages. The wrappers incorporate (and may extend) selected open source packages, and provide a simplified way for developers to use those packages in conjunction with the C++ Toolkit. Typical NCBI extensions include the ability to execute the 3rd-party code in a standard environment and adapting demo or test code to work within the Toolkit framework. The following is a list of chapters in this part:

21: XmlWrapp (XML parsing and handling, XSLT, XPath)

Created: August 2, 2009.
Last Update: July 22, 2013.

Overview

Introduction

The NCBI C++ Toolkit has forked and enhanced the open-source xmlwrapp project, which provides a simplified way for developers to work with XML. This chapter discusses the NCBI fork and how to use it. This chapter refers to NCBI's project as "XmlWrapp" and the open-source project as "xmlwrapp". Both projects produce a library named libxmlwrapp.

Chapter Outline

The following is an outline of the topics presented in this chapter:

General Information

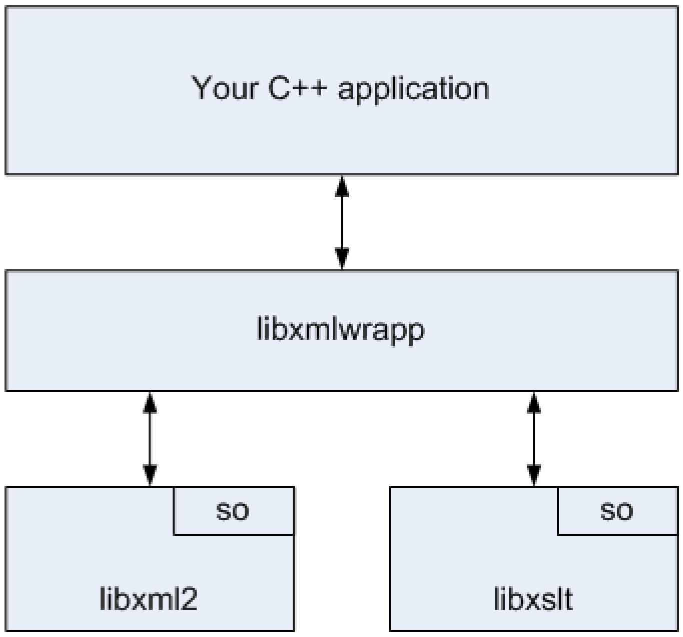
Both NCBI's XmlWrapp project and the open-source xmlwrapp project produce the libxmlwrapp library which is a generic XML handling C++ library built on top of widespread libxml2 / libxslt C libraries. The main features of libxmlwrapp are:

- Tree parser (DOM)
- Event parser (SAX)
- Creation / removal of nodes, attributes and documents
- Searching nodes and attributes
- XSLT transformation support
- DTD validation support
- XML catalog support

XmlWrapp was created by forking xmlwrapp and making these enhancements:

- Adding support for XPath.
- Implementing full-featured XML namespace support for both nodes and attributes.
- Adding XSD validation support.
- Extending the functionality of some existing classes.
- Adapting the demo code and test cases to work within the NCBI framework.
- Adding support for XSLT extension functions and extension elements.
- Adding the ability to transparently work with default attributes.
- Fixing some bugs that were in xmlwrapp.

The figure below illustrates the relationship between your C++ application and the XML libraries:



One goal of the libxmlwrapp library is to be a very thin wrapper around libxml2 / libxslt and to provide a simple yet powerful C++ interface without compromising speed. To achieve this goal, the library does not implement expensive run-time validity checks, and it is possible to write compilable C++ code that will cause a segmentation fault. For example, it is possible to create an unsafe XmlWrapp namespace object that points to an existing libxml2 namespace, then destroy the pointed-to namespace. This results in the unsafe libxmlwrapp namespace object containing a dangling pointer. Subsequent access of the pointer will cause an exception or abnormal termination.

The original open-source libxmlwrapp 0.6.0 was extended and modified to fit the NCBI C++ Toolkit build framework and API functionality requirements. Later, the functional changes introduced in 0.6.1 and 0.6.2 were patched into the NCBI code. Specific enhancements that NCBI incorporated into XmlWrapp include:

- XPath support:
  - XPath queries can be run based on XPath expressions. The queries return node sets which can be iterated.
- A new class, xml::schema, was added for XSD support.
- Implementing a full-featured XML namespace class, xml::ns, for use by both nodes and attributes, with these features:
  - Each node and attribute may be assigned to a namespace, or have their assignment removed. The assigned namespace can be retrieved.
  - Each node and attribute may contain a list of namespace definitions. Namespace definitions can be added to or removed from this list. The list can be retrieved.
  - XmlWrapp namespace objects can be either safe or unsafe. Safe namespace objects prevent program crashes by eliminating potentially invalid pointers. Using unsafe namespace objects requires less time and memory, but may result in invalid pointers and may cause a crash. See the safe and unsafe namespaces section for more details.

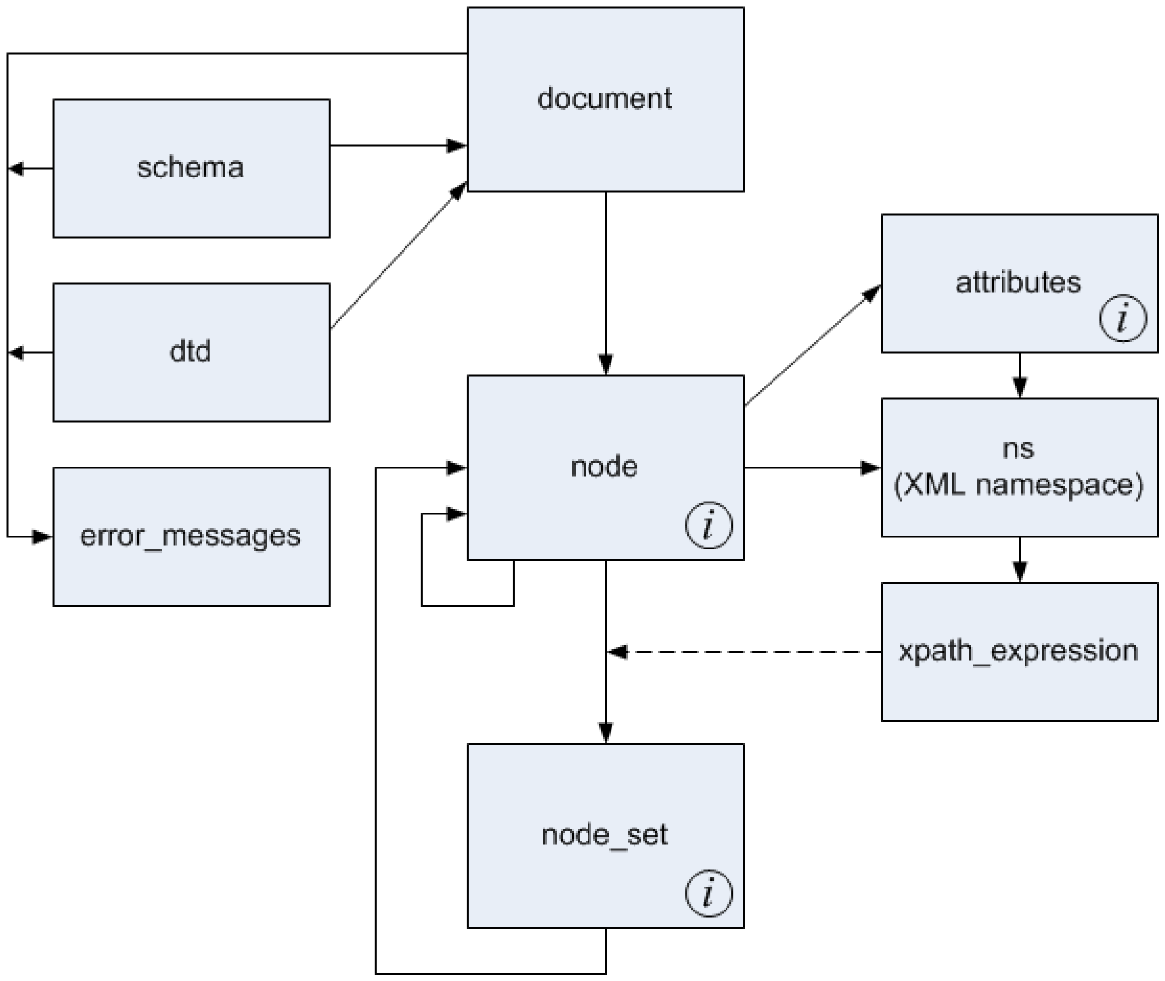
    - Nodes and attributes can now be searched by namespace as well as by name.
- Error handling was enhanced (or added) for tree and event parsing, and for DTD and XSD validation. Previously, only the last message was retained and processing stopped on the first error. Now all messages are retained and processing only stops if a fatal error is encountered.
- Adapting the demo code and test cases to work within the NCBI framework.
- Fixing some bugs that were in libxmlwrapp:
    - libxmlwrapp 0.6.0 did not copy the namespace when it copied an attribute. When XmlWrapp copies an attribute, it also copies the assigned namespace and all namespace definitions contained by the attribute.
    - The Sun WorkShop compiler failed to compile libxmlwrapp 0.6.0 because it was missing a definition for the STL distance algorithm. XmlWrapp conditionally defines this template for this compiler.
    - The XML parser in libxmlwrapp 0.6.0 failed to detect a certain form of malformed document. NCBI found and fixed this bug. The patch was submitted to the libxmlwrapp project and was accepted.
    - In libxmlwrapp 0.6.0 it was possible that using a reference to a node that was created by dereferencing an iterator could cause a core dump or unexpected data if the iterator was used after the reference was created.

The NCBI enhancements retain the generic nature of libxmlwrapp, and are not tailored to any particular application domain.

XmlWrapp demo applications and unit tests are available inside NCBI, but the most common and basic usage examples are given in the next section.

All the XmlWrapp functionality resides in the C++ namespaces xml:: and xslt::, and all the code is Doxygen-style commented.

XmlWrapp Classes



The figure above shows the most important classes in XmlWrapp. XML can be parsed from a file, memory buffer, or stream, creating a document object. One of the most important things you can get from the document object is the document root node.

Several classes in the figure are marked with the small "circled-i" symbol in the corner. This mark means that the class supports iterators and const iterators. The node class is a container of other nodes and you can iterate over immediate node children similar to how you do with STL containers.

A node may have an XML namespace and also may define namespaces. To support this, XmlWrapp provides the XML namespace class, xml::ns.

An XML node may have attributes as well, so XmlWrapp provides the xml::attributes class. This class is a container of attributes so both const and non-const iterators are provided.

The XPath support includes the xml::xpath_expression and xml::node_set classes. xpath_expression objects hold a single expression. node_set objects are created as the result of executing an XPath query for a given node. The node_set class is a container so it supports iterators.

To support XSD schema validation and DTD validation, XmlWrapp provides the xml::schema and xml::dtd classes. These classes work together with the xml::document class.

Warnings, errors and fatal errors may occur during the parsing and validating. To collect them, XmlWrapp provides the xml::error_messages class. The error_messages class includes the print() method, which returns a string containing a newline-separated list of messages. It also includes the has_warnings(), has_errors(), and has_fatal_errors() methods in case you are interested in the presence of a specific type of message. Note that errors and fatal errors are considered separately, so has_fatal_errors() may return true while has_errors() returns false.

How To

This section includes compact code fragments that show the essence of how to achieve typical goals using XmlWrapp. The examples do not illustrate all the features of XmlWrapp and are not intended to be complete and compilable. Your code will need to include the necessary headers, use try-catch blocks, check for errors, and validate the XML document.

Create a Document from an std::string Object

```
std::string xmldata( "<TagA>"
 "<TagB>stuff</TagB>"
 "</TagA>" );
xml::document doc( xmldata.c_str(), xmldata.size(), NULL );
```

Create a Document from a File

```
xml::document doc( "MyFile.xml", NULL );
```

Note: The second parameter above is a pointer to an error_messages object, which stores any messages collected while parsing the XML document (a NULL value can be passed if you're not interested in collecting error messages). For example:

```
xml::error_messages msgs;
xml::document doc( "MyFile.xml", &msgs );
std:cout << msgs.print() << std:endl;
```

Save a Document or Node to a File

The simplest way is inserting into a stream:

```
// save document
xml::document xmldoc( "abook" ); // "abook" is the root node
std::ofstream f( "doc_file.xml" );

f << xmldoc;
f.close();

// save node
xml::node n( "the_one" );
std::ofstream node_file( "node_file.xml" );

node_file << n << std::endl;
f.close();
```

The simplest way provides no control on how the output is formatted, but there is an alternative set of functions that accept formatting flags:

```
xml::document::save_to_string(...)
xml::document::save_to_stream(...)
xml::document::save_to_file(...)
xml::node::node_to_string(...)
```

For example, if you do not want to have the XML declaration at the beginning of the document then you might have code similar to:

```
xml::document doc( "example.xml", NULL );
std::string s;

doc.save_to_string( s, xml::save_op_no_decl );
```

For a complete list of available formatting flags, see enum xml::save_options.

Iterate Over Nodes

```
xml::document doc( "MyFile.xml", NULL );
xml::node & root = doc.get_root_node();

xml::node::const_iterator child( root.begin() );
xml::node::const_iterator child_end( root.end() );

std::cout << "root node is '" << root.get_name() << "'\n";
for ( ; child != child_end; ++child )
{
 if ( child->is_text() ) continue;
 std::cout << "child node '" << child->get_name() << "'" << std:endl;
}
```

Insert and Remove Nodes

```
xml::document doc( "MyFile2.xml", NULL );
xml::node & root = doc.get_root_node();
xml::node::iterator i = root.find( "insert_before", root.begin() );

root.insert( i, xml::node("inserted") );
i = root.find( "to_remove", root.begin() );
root.erase( i );
```

Iterate Over Attributes

```
xml::document doc( "MyFile.xml", NULL );
const xml::attributes & attrs = doc.get_root_node().get_attributes();

xml::attributes::const_iterator i = attrs.begin();
xml::attributes::const_iterator end = attrs.end();

for ( ; i!=end; ++i )
{
 std::cout << i->get_name() << "=" << i->get_value() << std:endl;
}
```

Insert and Remove Attributes

```
xml::document doc( "MyFile.xml", NULL );
xml::attributes & attrs = doc.get_root_node().get_attributes();

attrs.insert( "myAttr", "attrValue" );
xml::attributes::iterator i = attrs.find( "attrToRemove" );
attrs.erase( i );
```

Work with XML Namespaces

```
xml::document doc( "MyFile.xml", NULL );
xml::node & root = doc.get_root_node();
xml::ns rootSpace( root.get_namespace() );

std::cout << "Root namespace: " << rootSpace.get_prefix() << "->"
 << rootSpace.get_uri() << std:endl;

xml::attributes & attrs = root.get_attributes();
xml::attributes::iterator attr( attrs.find( "firstAttr" ) );
xml::ns attrSpace( attr->get_namespace() );

std::cout << "Attribute namespace: " << attrSpace.get_prefix() << "->"
 << attrSpace.get_uri() << std:endl;
root.add_namespace_definition( xml::ns( "myPrefix", "myURI" ),
 xml::node::type_throw_if_exists );
root.set_namespace( "myPrefix" );
attr->set_namespace( "myPrefix" );
```

Use an Event Parser

For those within NCBI, there is sample code showing how to use an event parser.

Make an XSLT Transformation

```
xml::document doc( "example.xml", NULL );
xslt::stylesheet style( "example.xsl" );
xml::document result = style.apply( doc );
std::string tempString;

std::cout << "Result:\n" << result << std:endl;
// or
result.save_to_string( tempString );

// you can also specify save options, e.g. to omit the XML declaration:
result.save_to_string( tempString, xml::save_op_no_decl );
```

Other methods and options are available for saving the transformation result - see save_to_stream(), save_to_file(), and save_options.

Note: The transformation output will be affected by a number of factors:

- If there is no output method specified in the XSL, or if the specified method is not "html" or "text", then the effective output method will be "xml".
- On Windows, the effective output method will be "xml", regardless of the output method specified in the XSL.
- The save options are only applicable when the effective output method is "xml".
- If the effective output method is "xml", an XML declaration will be prepended to the transformation result when serialized (unless suppressed by the xml::save_op_no_decl save option).
- There are three conditions for which an empty "<blank/>" node will be appended to the transformation output:
  - The output method specified in the XSL is not "xml" or "text".
  - The output method specified in the XSL is "xml" but the XML is not well-formed.
  - The output method specified in the XSL is "text" and the platform is Windows.

Run an XPath Query

```
xml::document doc( "example.xml", NULL );
xml::node & root = doc.get_root_node();
xml::xpath_expression expr( "/root/child" );
const xml::node_set nset( root.run_xpath_query( expr ) );
size_t nnum( 0 );
xml::node_set::const_iterator k( nset.begin() );

for ( ; k != nset.end(); ++k )
 std::cout << "Node #" << nnum++ << std::endl
 << *k << std::endl;
```

Please note that the node_set object holds a set of references to the nodes from the document which is used to run the XPath query. Therefore you can change the nodes in the original document if you use a non-constant node_set and non-constant iterators.

The xpath_expression object also supports:

- pre-compilation of the XPath query string
- namespace registration (a single namespace or a list of namespaces)

Run an XPath Query with a Default Namespace

The XPath specification does not support default namespaces, and it considers all nodes without prefixes to be in the null namespace, not the default namespace. This creates a problem when you want to search for nodes to which a default namespace applies, because the default namespace cannot be directly matched. For example, the following code will not find any matches:

```
std::string xmldata("<A xmlns=\"http://nlm.nih.gov\">"
 "<B><C>stuff</C></B>"
 "</A>" );
xml::document doc( xmldata.c_str(), xmldata.size(),
 NULL );
xml::node & root = doc.get_root_node();
xml::xpath_expression expr( "//B/C" );
```

```
const xml::node_set nset( root.run_xpath_query( expr ) );
size_t nnum( 0 );
xml::node_set::const_iterator k( nset.begin() );

for ( ; k != nset.end(); ++k )
 std::cout << "Node #" << nnum++ << std::endl
 << *k << std::endl;
```

The solution is to create a special namespace with the sole purpose of associating a made-up prefix with the URI of the default namespace. Use that namespace when creating the XPath expression, and prefix the nodes in your XPath expression with your made-up prefix. This prefix should be distinct from other prefixes in the document. The following code will find the desired node:

```
std::string xmldata("<A xmlns=\"http://nlm.nih.gov\">"
 "<B><C>stuff</C></B>"
 "</A>" );
xml::document doc( xmldata.c_str(), xmldata.size(),
 NULL );
xml::node & root = doc.get_root_node();

 // here we add a made-up namespace
xml::ns fake_ns( "fake_pfx", "http://nlm.nih.gov" );

 // now we register the made-up namespace and
 // use the made-up prefix
xml::xpath_expression expr( "//fake_pfx:B/fake_pfx:C", fake_ns );

const xml::node_set nset( root.run_xpath_query( expr ) );
size_t nnum( 0 );
xml::node_set::const_iterator k( nset.begin() );

for ( ; k != nset.end(); ++k )
 std::cout << "Node #" << nnum++ << std::endl
 << *k << std::endl;
```

Use an Extension Function

```
class myExtFunc : public xslt::extension_function
{
 public:
 void execute (const std::vector<xslt::xpath_object> & args,
 const xml::node & node,
 const xml::document & doc)
 {
 set_return_value( xslt::xpath_object( 42 ) );
 }
};

//...

 std::string doc_as_string = "<root><nested/></root>";
 xml::document doc( doc_as_string.c_str(),
 doc_as_string.size(), NULL );

 std::string style_as_string =
 "<xsl:stylesheet xmlns:xsl="
 "\"http://www.w3.org/1999/XSL/Transform\" "
 "xmlns:my=\"http://bla.bla.bla\">"
 "<xsl:output method=\"text\"/>"
 "<xsl:template match=\"/root/nested\">"
 "<xsl:value-of select=\"my:test(15)\"/>"
 "</xsl:template>"
 "</xsl:stylesheet>";
 xslt::stylesheet sheet( style_as_string.c_str(),
 style_as_string.size() );

 myExtFunc * myFunc = new myExtFunc;
 sheet.register_extension_function( myFunc, "test", "http://bla.bla.bla",
 xml::type_own );
 // sheet now owns myFunc, so there is no need to delete myFunc

 xml::document result = sheet.apply( doc );

 std::cout << result << std::endl; // "42"
```

Please also see the xslt::extension-function class reference.

Users inside NCBI can view the extension function unit tests for more usage examples.

Use an Extension Element

```
class myExtElem : public xslt::extension_element
{
 public:
 void process (xml::node & input_node,
 const xml::node & instruction_node,
 xml::node & insert_point,
 const xml::document & doc)
 {
 xml::node my( "inserted", "content" );
 insert_point.push_back( my );
 }
};

// ...

 std::string doc_as_string = "<root><nested/></root>";
 xml::document doc( doc_as_string.c_str(),
 doc_as_string.size(), NULL );

 std::string style_as_string =
 "<xsl:stylesheet xmlns:xsl="
 "\"http://www.w3.org/1999/XSL/Transform\" "
```

```
"xmlns:my=\"http://bla.bla.bla\" "
"extension-element-prefixes=\"my\">"
"<xsl:output method=\"xml\"/>"
"<xsl:template match=\"/root/nested\">"
"<my:test/>"
"</xsl:template>"
"</xsl:stylesheet>";
xslt::stylesheet sheet( style_as_string.c_str(),
style_as_string.size() );

myExtElem * myElem = new myExtElem;
sheet.register_extension_element( myElem, "test", "http://bla.bla.bla",
xml::type_own );
// sheet now owns myElem, so there is no need to delete myElem

xml::document result = sheet.apply( doc );
xml::node & result_root = result.get_root_node();

std::cout << result_root.get_name() << std::endl; // "inserted"
std::cout << result_root.get_content() << std::endl; // "content"
```

Please also see the xslt::extension-element class reference.

Users inside NCBI can view the extension element unit tests for more usage examples.

Use an XML Catalog

The XML_CATALOG_FILES environment variable may be used in one of three ways to control the XML catalog feature of libxml2 – i.e. the way libxml2 resolves unreachable external URI's:

1. If XML_CATALOG_FILES is not set in the process environment then the default catalog will be used.
2. If it is set to an empty value then the default catalog will be deactivated and there will be no resolution of unreachable external URI's.
3. If it is set to a space-separated list of catalog files, then libxml2 will use these files to resolve external URI's. Any invalid paths will be silently ignored.

The default catalog is /etc/xml/catalog for non-Windows systems. For Windows, the default catalog is <module_path>\..\etc\catalog, where <module_path> is the path to the installed libxml2.dll, if available, otherwise the path to the running program.

The XML_CATALOG_FILES environment variable is read once before the first parsing operation, and then any specified catalogs are used globally for URI resolution in all subsequent parsing operations. Therefore, if the XML_CATALOG_FILES value is to be set programmatically, it must be done prior to the first parsing operation.

There is another environment variable (XML_DEBUG_CATALOG) to control debug output. If it is defined, then debugging output will be enabled.

Warning: Collaborative Use of XmlWrapp and libxml2

XmlWrapp uses the _private field of the raw libxml2 xmlNode data structure for internal purposes. Therefore, if libxml2 and XmlWrapp are used collaboratively then this field must not be used in client code. If it is used, it may cause a core dump or other undefined behavior.

Implementation Details

Copying and Referencing Nodes

xml::node objects are frequently required when working with XML documents. There are two ways to work with a given node:

- by referencing it; or
- by copying it.

This example shows both ways:

```
xml::document doc( "example.xml", NULL );
xml::node_set nset( doc.get_root_node().
 run_xpath_query( "/root/child" ) );

// Iterate over the result node set
xml::node_set::iterator k = nset.begin();
for ( ; k != nset.end(); ++k ) {

 // just reference the existing node
 xml::node & node_ref = *k;

 // create my own copy (which I'll own and destroy)
 xml::node * my_copy = k->detached_copy();

 // Do something
 ...

 // Don't forget this
 delete my_copy;
}
```

What is the difference between the node_ref and my_copy variables?

The node_ref variable refers to a node in the original document loaded from example.xml. If you change something using the node_ref variable you'll make changes in the original document object.

The my_copy variable is a recursive copy of the corresponding node together with all used namespace definitions, non-default attributes, and nested nodes. The copy has no connection to the original document. The my_copy variable has no parent node and has no links to the internal and external subsets (DTDs) which the original document could have. If you change something using the my_copy variable you'll make changes in the copy but not in the original document. Obviously it takes more time to create such a recursive copy of a node.

Note: It is recommended to pass nodes by reference when appropriate to maximize performance and avoid modification of copies.

Using Namespaces with XPath Expressions

XmlWrapp provides the xml::xpath_expression class for building reusable XPath expressions. If namespaces are involved then one of the constructors which accept a namespace or a list of namespaces should be used. Otherwise the XPath query results may not have the nodes you expect to get.

XmlWrapp also provides a convenience method for the nodes: xml::node::run_xpath_query (const char * expr). This method builds an xpath_expression internally and registers all the effective namespaces for the certain node. While it is very convenient as you don't need to know in advance what the namespace definitions are, this method has some drawbacks:

- The internally built xpath_expression is not reusable, so it gets rebuilt every time a query is run - even if the same expression was used before.
- The list of effective namespace definitions for a certain node can be quite long and may exceed your actual needs. It takes time to build such a list and to register them all so it affects the performance.

Recommendations:

- If you need the best performance then use xml::xpath_expression explicitly and do not forget to provide a list of the required namespaces.
- If you aren't concerned about performance then use one of the xml::node::run_xpath_query(const char * expr) methods.

Containers of Attributes - Iteration and Size

Sometimes it is necessary to iterate over a node's attributes or to find an attribute. Let's take a simple example:

```
<?xml version="1.0" ?>
<root xmlns:some_ns="http://the.com"
 attr1 = "val1"
 foo = "fooVal"
 some_ns:bar = "barVal">
</root>
```

XmlWrapp provides an STL-like way of iterating over the attributes, e.g:

```
void f( const xml::node & theNode ) {
 const xml::attributes & attrs = theNode.get_attributes();

 for ( xml::attributes::const_iterator k = attrs.begin();
 k != attrs.end(); ++k )
 std::cout << "Attribute name: " << k->get_name()
 << " value: " << k->get_value() << std::endl;
}
```

You may notice that iterators are used here and the iterators can be incremented.

Note: Although iterating over attributes is STL-like, searching for an attribute is only partially STL-like. Iterators returned by the find() method cannot be incremented, but both operator -> and operator * can be used. The following code will work:

```
void f( const xml::node & theNode, const char * attrName ) {
 const xml::attributes & attrs = theNode.get_attributes();
 xml::attributes::const_iterator found = attrs.find( attrName );

 if ( found != attrs.end() )
 std::cout << "Found name: " << (*found).get_name()
 << "Found value: " << found->get_value() << std::endl;
}
```

but this code will generate an exception:

```
void f( const xml::node & theNode, const char * attrName ) {
 const xml::attributes & attrs = theNode.get_attributes();
 xml::attributes::const_iterator found = attrs.find( attrName );

 if ( found != attrs.end() )
 ++found; // Exception is guaranteed here
}
```

This implementation detail is related to the limitations of libxml2 with respect to default attributes. Let's take an example that has a DTD:

```
<?xml version="1.0"?>
<!DOCTYPE root PUBLIC "something" "my.dtd" [
<!ATTLIST root defaultAttr CDATA "defaultVal">
]>
<root xmlns:some_ns="http://the.com"
 attr1 = "val1"
 foo = "fooVal"
 some_ns:bar = "barVal">
</root>
```

This example introduces a default attribute called defaultAttr for the root node. The libxml2 library stores default and non-default attributes separately. The library provides very limited access the default attributes - there is no way to iterate over them and the only possible way to get a default attribute is to search for it explicitly. For example:

```
void f( const xml::node & theNode ) {
 const xml::attributes & attrs = theNode.get_attributes();
 xml::attributes::const_iterator found = attrs.find( "defaultAttr" );

 if ( found != attrs.end() ) {
 std::cout << "Default? " << found->is_default() << std::endl;
 std::cout << "Name: " << found->get_name()
 << " Value: " << found->get_value() << std::endl;
 }
}
```

XmlWrapp forbids incrementing iterators provided by xml::attributes::find(...) methods because:

- libxml2 has limited support for working with default attributes; and

- iterators provided by the xml::attributes::find() methods may point to either a default or a non-default attribute.

Note: This libxml2 limitation affects the xml::attributes::size() method behavior. It will always provide the number of non-default attributes and will never include the number of default attributes regardless of whether or not a node has default attributes.

Changing Default Attributes

libxml2 does not provide the ability to change a default attribute. XmlWrapp does provide this ability, but at the cost of implicitly converting the default attribute into a non-default attribute. Consider the following document:

```
<?xml version="1.0"?>
<!DOCTYPE root PUBLIC "something" "my.dtd" [
<!ATTLIST root language CDATA "EN">
]>
<root xmlns:some_ns="http://the.com"
 some_ns:bar = "barVal">
</root>
```

The code below demonstrates changing a default attribute and is totally OK as explained in the comments (error handling is omitted for clarity):

```
xml::document doc( "example.xml", NULL );
xml::node & root = doc.get_root_node();
xml::attributes & attrs = root.get_attributes();
xml::attributes::iterator j = attrs.find( "language" );

// Here j points to the default attribute
assert( j->is_default() == true );

// Now suppose we need to change the default language to French.
// It is forbidden to change the default attribute's values because
// the default attribute might be applied to many nodes while a change
// could be necessary for a single node only.
// So, to make a change operation valid, XmlWrapp first converts the default
// attribute to a non-default one and then changes its value.

j->set_value( "FR" );

// Now the iterator j is still valid and points to a non-default attribute
assert( j != attrs.end() );
assert( j->is_default() == false );

// If you decide to save the document at this point then you'll see
// the root node with one node attribute language="FR"
```

A similar conversion will happen if you decide to change a default attribute namespace.

XmlWrapp will also ensure that all iterators pointing to the same attribute remain consistent when multiple iterators point to the same default attribute and one of them is changed. For example:

```
xml::document doc( "example.xml", NULL );
xml::node & root = doc.get_root_node();
xml::attributes & attrs = root.get_attributes();
xml::attributes::iterator j = attrs.find( "language" );
xml::attributes::iterator k = attrs.find( "language" );

// Here we have two iterators j and k pointing to the same default attribute
assert( j->is_default() == true );
assert( k->is_default() == true );

// Now the attribute is implicitly converted to a non-default one
// using one of the iterators
j->set_value( "FR" );

// Both j and k iterators are now pointing to a non-default (ex-default)
// attribute
assert( j->is_default() == false );
assert( k->is_default() == false );

// And of course:
assert( j->get_value() == std::string( "FR" ) );
assert( k->get_value() == std::string( "FR" ) );
```

For a diagram illustrating how the XmlWrapp library handles iterators and changed default attributes, please see Figure 1, Phantom Attributes.

Event Parser and Named Entities

When using xml::event_parser, three functions are involved in parsing an XML document that contains named entities:

- xml::init::substitute_entities()
  This method controls whether the xml::event_parser::entity_reference() callback is called or not, and must be called before the event parser is created.
- xml::event_parser::text()
  This callback will be called for both text nodes and named entity nodes.
- xml::event_parser::entity_reference()
  This callback may be called for named entity nodes.

Imagine that an event parser which implements both text() and entity_reference() callbacks receives the following document as in input:

```
<?xml version="1.0"?>
<!DOCTYPE EXAMPLE SYSTEM "example.dtd" [ <!ENTITY my "VALUE">]>
<root><node>Super &my; oh!</node></root>
```

Then the table below lists the callbacks that are called, depending on the value passed to substitute_entities():

Having this call before the parser is created: xml::init::substitute_entities(true) results in the following callbacks:	**Having this call before the parser is created: xml::init::substitute_entities(false) results in the following callbacks:**
xml::event_parser::text("Super ")	xml::event_parser::text("Super ")
xml::event_parser::text("VALUE")	xml::event_parser::text("VALUE")
	xml::event_parser::entity_reference("my")
xml::event_parser::text(" oh!")	xml::event_parser::text(" oh!")

So the difference is that the entity_reference() callback is never called if substitute_entities (true) is called. Note: The entity_reference() callback is also not called if a standard entity is used (e.g. &apos;, &amp;, &quot;, &lt;, &gt;), regardless of any call to substitute_entities().

Character entities are handled the same way as named entities.

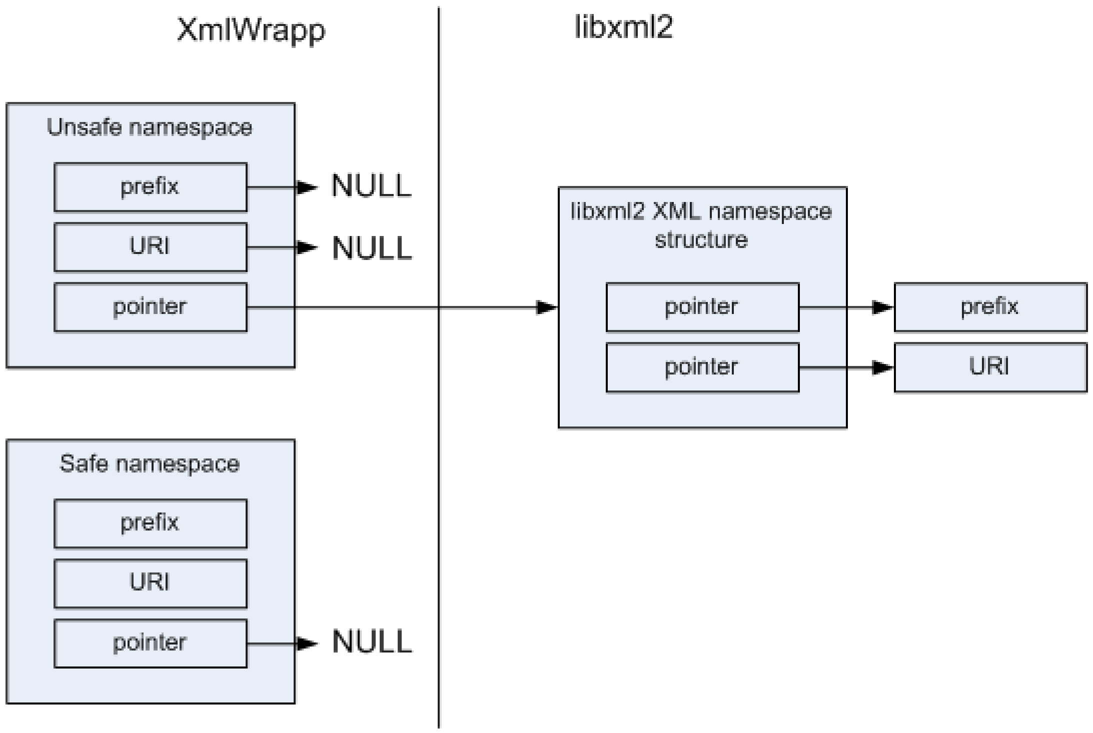
Generally speaking, the event parser in XmlWrapp behaves the same way as in libxml2 in terms of what callbacks are called - except that the callbacks in XmlWrapp are C++ methods whereas the callbacks in libxml2 are C functions.

Safe and Unsafe Namespaces

XmlWrapp provides a wrapper class called xml::ns to work with namespaces. The xml::ns class can be of two types: safe and unsafe.

To understand the difference between them it is necessary to know how libxml2 works with namespaces. Namespace structures in libxml2 store two pointers to character strings - a namespace prefix and a namespace URI. These structures are stored in a linked list and each XML document element that might have a namespace has a pointer that points to a namespace structure. Thus, namespaces can be uniquely identified by either a namespace pointer or by a prefix / URI pair.

XmlWrapp covers both ways. The xml::ns can store its own copies of the namespace prefix and URI, and in this case the namespace is called safe. Or, the xml::ns can store just a pointer to the corresponding namespace structure, and in this case the namespace is called unsafe.



A safe namespace can be constructed based on strings provided by the user or by making copies of the prefix and URI strings extracted from the libxml2 low level structure. Having a copy of the strings makes it absolutely safe to manipulate namespaces - it is even possible to get a namespace from one document, destroy the document, and then apply the stored namespace to another document.

When XmlWrapp receives an unsafe namespace for a namespace manipulation operation, it does not perform any checks and uses the raw pointer as-is. So there is a chance to break your document and even cause your application to core dump if an unsafe namespace is used improperly. For example the user may take an unsafe namespace from one document, destroy the document, and then apply the stored unsafe namespace to another document. At the time the original document is destroyed the low level namespace structure is destroyed as well but the pointer to the namespace is still stored so any access operation will cause problems.

Unsafe namespaces have some advantages though. They require less memory and they work faster. So the recommendation is to use safe namespaces unless you really need the best possible performance and slight reduction of the memory footprint.

FAQ

Q. Is libxmlwrapp **thread safe?**

A. As safe as libxml2 and libxslt are. It is still better to avoid simultaneous processing of the same document from many threads.

Q. Does libxmlwrapp **support XML catalogs?**

A. Yes, to the extent that libxml2 supports them. All the libxml2 fuctionality is available, but there is no special support code for XML catalogs in the libxmlwrapp library. See the How to Use an XML Catalog section for details.

Q. What header files do I need to include?

A. You need <misc/xmlwrapp/xmlwrapp.hpp> for functionality that resides in the xml:: C++ namespace, and <misc/xmlwrapp/xsltwrapp.hpp> for functionality that resides in the xslt:: C ++ namespace.

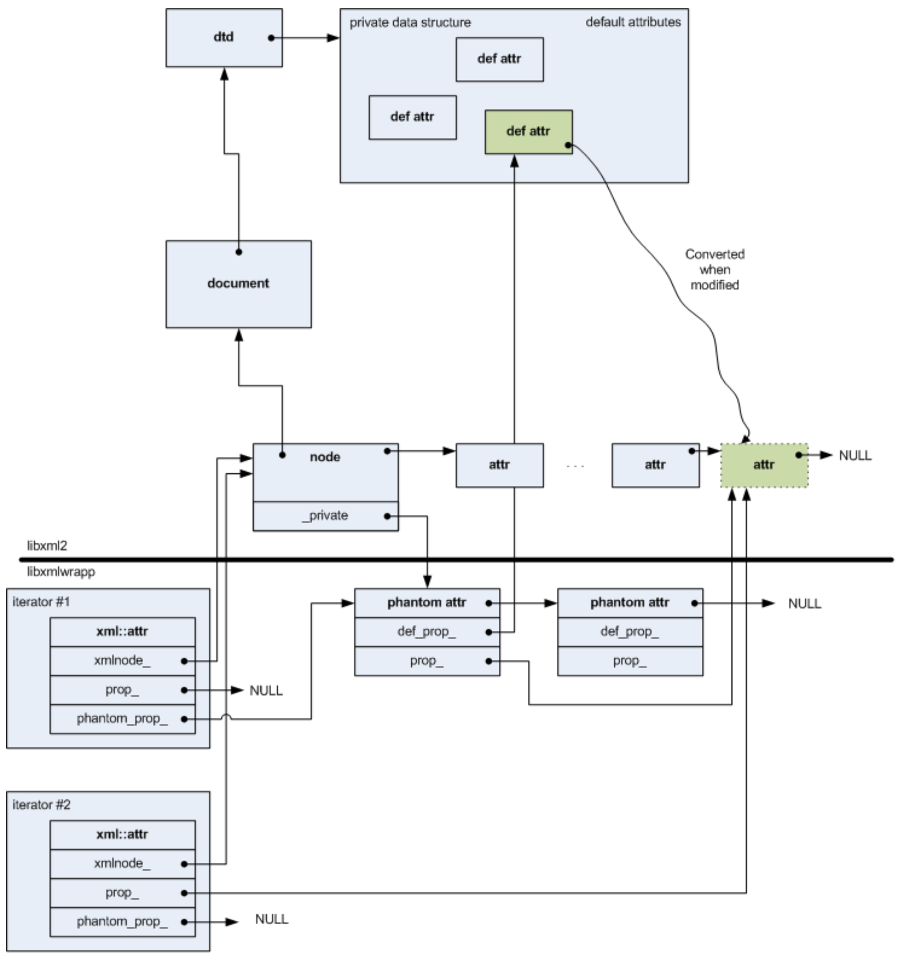
Q. What do I need to add to my Makefile?

A. You need to add the following:

```
LIB = xmlwrapp xncbi
LIBS = $(LIBXML_LIBS) $(LIBXSLT_LIBS) $(ORIG_LIBS)
CPPFLAGS = $(LIBXML_INCLUDE) $(LIBXSLT_INCLUDE) $(ORIG_CPPFLAGS)
REQUIRES = LIBXML LIBXSLT
```

Q. Does XmlWrapp support XPath 2.0?

A. XmlWrapp is based on libxml2, and libxml2 does not now and may never support XPath 2.0.



Sequence of events:

- The user searches for an attribute by name and finds a default attribute. The result is iterator #1. Internally this causes creation of the phantom attr structure linked to the corresponding node.
- The user modifies the value of the default attribute using iterator #1. This causes conversion of the default attribute to a regular one. The phantom attr structure pointers are set correspondingly.
- The user searches for an attribute by name (the same name as in the first search) and finds the [converted] attribute. The result is iterator #2.
- The iterators are compared and the result is true because they are not compared directly, they are compared via the final target data structure.

Figure 1. Phantom Attributes.

Part 5: Software

Part 5 discusses debugging mechansims, development tools, examples, demos and tests for the C++ Toolkit. The following is a list of chapters in this part:

22: Debugging, Exceptions, and Error Handling

Last Update: July 18, 2010.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discusse the debugging mechanisms available in the NCBI C++ toolkit. There are two approaches to getting more information about an application, which does not behave correctly:

- Investigate the application's log without recompiling the program,
- Add more diagnostics and recompile the program.

Of course, there is always the third method which is to run the program under an external debugger. While using an external debugger is a viable option, this method relies on an external program and not on a log or diagnostics produced by the program itself which in many cases is customized to reflect the program behavior, and can, therefore, more quickly reveal the source of errors.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Extracting Debug Data

The C++ Toolkit has several mechanisms which can be used by a programmer to extract information about the program usage, printing trace and diagnostic messages, and examining the object state dump. The following sections discuss these topics in more detail:

- Command Line Parameters.
- Getting More Trace Data.
- Tracing in the Connection Library
- NCBI C++ Toolkit Diagnostics
- Object state dump
- Exceptions

Command Line Parameters

There are several command line parameters (see Table 1), which are applicable to any program which utilizes NCBI C++ toolkit, namely CNcbiApplication class. They provide with the possibility

- to obtain a general description of the program as well as description of all available command line parameters (-h flag),
- to redirect the program's diagnostic messages into a specified file (-logfile key),
- to read the program's configuration data from a specified file (-conffile key).

Getting More Trace Data

All NCBI C++ toolkit libraries produce a good deal of diagnostic messages. Still, many of them remain "invisible" - as long as the tracing is disabled. Some tracing data is only available in debug builds - see _TRACE macro for example. Other - e.g., the one produced by ERR_POST or LOG_POST macros - could be disabled. There are three ways to manipulate these settings, that is enable or disable tracing, or set the severity level of messages to print:

- from the application itself,
- from the application's configuration file,
- with the help of environment variables.

The following additional topics relating to trace data are presented in the subsections that follow:

- Tracing
- Diagnostic Messages

Tracing

There are two ways to post trace messages: using either the _TRACE macro or the ERR_POST macro. Trace messages produced with the help of _TRACE macro are only available in debug mode, while those posted by ERR_POST are available in both release and debug builds. By default, tracing is disabled. See Table 2 for settings to enable tracing.

Please note, when enabling trace from a configuration file, some trace messages could be lost: before configuration file is found and read the application may assume that the trace was disabled. The only way to enable tracing from the very beginning is by setting the environment variable.

Diagnostic Messages

Diagnostic messages produced by ERR_POST macro are available both in debug and release builds. Such messages have a severity level, which defines whether the message will be actually printed or not, and whether the program will be aborted or not. To change the severity level threshold for posting diagnostic messages, see Table 3.

Only those messages, which severity is equal or exceeds the threshold will be posted. By default, messages posted with Fatal severity level also abort execution of the program. This can be changed by SetDiagDieLevel(EDiagSev dieSev) API function.

Tracing in the Connection Library

The connection library has its own tracing options. It is possible to print the connection parameters each time the link is established, and even log all data transmitted through the socket during the life of the connection (see Table 4).

NCBI C++ Toolkit Diagnostics

NCBI C++ toolkit provides with a sophisticated diagnostic mechanism. Diagnostic messages could be redirected to different output channels. It is possible to set up what additional information should be printed with a message, for example date/time stamp, file name, line number etc. Some macros are defined only in debug mode:_TRACE, _ASSERT, _TROUBLE. Others are also defined in release mode as well: _VERIFY, THROW*_TRACE.

Object state dump

Potentially useful technique in case of trouble is to use object state dump API. In order to use it, the object's class must be derived from CDebugDumpable class, and implementation of the class should supply meaningful dump data in its DebugDump function. Debug dump gives an object's state snapshot, which can help in identifying the cause of problem at run time.

Exceptions

NCBI C++ toolkit defines its own type of C++ exceptions. Unlike standard ones, this class

- makes it possible to define error codes (specific to each exception class), which could be analyzed from a program,
- provides with more information about where a particular exception has been thrown from (file name and line number),
- gives the possibility to create a stack of exceptions to accumulate a backlog of events (unfinished jobs) which caused the problem,
- has elaborated, customizable reporting mechanism,
- supports using standard diagnostic mechanism with all the configuration options it provides.

NCBI C++ Error Handling and Diagnostics

The following topics are discussed in this section:

- Debug-mode for Internal Use
- C++ Exceptions
- Standard NCBI C++ Message Posting

Debug-mode for Internal Use

#include <corelib/ncbidbg.hpp> [also included in <corelib/ncbistd.hpp>]

There are four preprocessor macros (_TROUBLE, _ASSERT, _VERIFY and _TRACE) to help the developer to catch some (logical) errors on the early stages of code development and to hardcode some assertions on the code and data behaviour for internal use. All these macros gets disabled in the non-debug versions lest to affect the application performance and functionality; to turn them on, one must #define the _DEBUG preprocessor variable. Developer must be careful and do not use any code with side effects in _ASSERT or _TRACE as this will cause a discrepancy in functionality between debug and non-debug code. For example, _ASSERT(a++) and _TRACE("a++ = " << a++) would increment "a" in the debug version but do nothing in the non-debug one).

- _TROUBLE -- Has absolutely no effect if _DEBUG is not defined; otherwise, unconditionally halt the application.
- _ASSERT(expr) -- Has absolutely no effect if _DEBUG is not defined; otherwise, evaluate expression expr and halt the application if expr resulted in zero(or "false").

- _VERIFY(expr) -- Evaluate expression expr; if _DEBUG is defined and expr resulted in zero(or "false") then halt the application.
- _TRACE(message) -- Has absolutely no effect if _DEBUG is not defined; otherwise, it outputs the message using Standard NCBI C++ message posting. NOTE: as a matter of fact, the tracing is turned off by default, even if _DEBUG is defined, and you still have to do a special configuration to really turn it on.

All these macros automatically report the file name and line number to the diagnostics. For example, this code located in file "somefile.cpp" at line 333:

```
int x = 100;
_TRACE( "x + 5 = " << (x + 5) );
```

will output:

```
"somefile.cpp", line 333: Trace: x + 5 = 105
```

C++ Exceptions

#include <corelib/ncbiexpt.hpp> [also included in <corelib/ncbistd.hpp>]

The following additional topics are discussed in this section:

- Standard C++ Exception Classes, and Two Useful NCBI Exception Classes (CErrnoTemplException, CParseTemplException)
- Using STD_CATCH_*(...) to catch and report exceptions
- Using THROW*_TRACE(...) to throw exceptions
- THROWS*(...) -- Exception Specification

Standard C++ Exception Classes, and Two Useful NCBI Exception Classes (CErrnoTemplException, CParseTemplException)

One must use CException as much as possible. When not possible, standard C++ exceptions should be used. There are also a couple of auxiliary exception classes derived from std::runtime_error that may be used if necessary.

- CErrnoTemplException -- to report failure in a standard C library function; it automatically appends to the user message a system-specific description reported by errno
- CParseTemplException -- to report an erroneous position (passed in the additional constructor parameter) along with the user message

Then, it is **strongly recommended** that when CException can't be used, and when the basic functionality provided by standard C++ exceptions is insufficient for some reason, one must derive new ad hoc exception classes from one of the standard exception classes. This provides a more uniform way of exception handling, because most exceptions can be caught and appropriately handled using the STD_CATCH_*(...) preprocessor macros as described below.

Using STD_CATCH_*(...) to catch and report exceptions

You can use the STD_CATCH_*(...) macros to catch exceptions potentially derived from the standard exception class std::exception when you just want to print out a given error name, subcode, and message along with the information returned from std::exception::what().

The STD_CATCH_X(subcode, message) and STD_CATCH_XX(name, subcode, message) macros only catch exceptions derived from std::exception, and post the given error name, subcode, and message along with the information returned from std::exception::what().

The STD_CATCH_ALL_X(subcode, message) and STD_CATCH_ALL_XX(name, subcode, message) macros first try to catch a std::exception-derived exception (using the STD_CATCH_X and STD_CATCH_XX macros, respectively), and if the thrown exception was not caught (i.e. if it is not derived from std::exception) then they catch all exceptions and post the given error name, subcode, and message.

The name argument must match one of the pre-defined values in the error_codes.hpp header for the relevant module (e.g. connect), and the subcode argument must be within the range specified in the same place. The message argument can be of any form acceptable by the diagnostic class CNcbiDiag.

Using these macros makes dealing with exceptions in NCBI C++ code easy:

```
class foreign_exception { ..... };
class exception_derived_user : public exception { ..... };
char arg1 = "qqq";
int arg2 = 888;
try {
 SomeFunc(arg1, arg2);
} catch (foreign_exception& fe) {
 // do something special with the particular "non-standard"
 // (not derived from "std::exception") exception "foreign_exception"
} catch (exception_derived_user& eu) {
 // do something special with the particular "standard"
 // (derived from "std::exception") exception "exception_derived_user"
}
// handle all other "standard" exceptions in a uniform way
STD_CATCH_X( 1, "in SomeFunc(" << arg1 << "," << arg2 << ")" );
```

Here, if SomeFunc() executes throw std::runtime_error("Invalid Arg2"); then the application will print out (to its diagnostic stream) something like:

```
Error: (101.1) [in SomeFunc(qqq,888)] Exception: Invalid Arg2
```

In this output, the (101.1) indicates the error code (defined in the module's error_codes.hpp header) and the subcode passed to STD_CATCH_X.

*Using THROW*_TRACE(...) to throw exceptions*

If you use one of THROW*_TRACE(...) macros to throw an exception, and the source was compiled in a debug mode (i.e. with the preprocessor _DEBUG defined), then you can turn on the following features that proved to be very useful for debugging:

- If the tracing is on, then the location of the throw in the source code and the thrown exception will be printed out to the current diagnostic stream, e.g.:
  THROW_TRACE(CParseException, ("Failed parsing(at pos. 123)", 123));

  "coretest.cpp", line 708: Trace: CParseException: {123}
  Failed parsing(at pos. 123)

```
---------------------------------

strtod("1e-999999", 0);
THROW1_TRACE(CErrnoException, "Failed strtod('1e-999999', 0)");

"coretest.cpp", line 718: Trace: CErrnoException:
Failed strtod('1e-999999', 0): Result too large
```

- Sometimes, it can be convenient to just abort the program execution at the place where you throw an exception, e.g. in order to examine the program stack and overall state that led to this throw. By default, this feature is not activated. You can turn it on for your whole application by either setting the environment variable $ABORT_ON_THROW to an arbitrary non-empty string, or by setting the application's registry entry ABORT_ON_THROW (in the [DEBUG] section) to an arbitrary non-empty value. You also can turn it on and off in your program code, calling function SetThrowTraceAbort().

NOTE: if the source was not compiled in the debug mode, then the THROW*_TRACE(...) would just throw the specified exception, without doing any of the "fancy stuff" we just described.

THROWS(...) -- Exception Specification*

One is discouraged from writing exception specifications - either with throw() or the THROWS* macros.

Standard NCBI C++ Message Posting

#include <corelib/ncbidiag.hpp> [also included in <corelib/ncbistd.hpp>]

Standard diagnostics is provided with the CNcbiDiag class. A given application can have as many objects of this class as needed. An important point to remember is that each instance of the CNcbiDiag class actually stores (and allows to append to) only one message at a time. When the message controlled by an instance of CNcbiDiag is complete, CNcbiDiag invokes the Post() method of a global handler object (of type CDiagHandler) and passes the message (along with its severity level) as the method's argument.

Usually, this global object would merely dump the message to a diagnostic stream, and there is an auxiliary function SetDiagStream() that can be used to specify the output stream for the diagnostics. One can call SetDiagStream(&NcbiCerr) to dump the diagnostics to the standard error output stream:

```
/// Set diagnostic stream.
///
/// Error diagnostics are written to output stream "os"
/// This uses the SetDiagHandler() functionality.
NCBI_XNCBI_EXPORT
extern void SetDiagStream
(CNcbiOstream* os,
 bool quick_flush = true,///< Do stream flush after every message
 FDiagCleanup cleanup = 0, ///< Call "cleanup(cleanup_data)" if diag.
 void* cleanup_data = 0 ///< Stream is changed (see SetDiagHandler)
 );
```

Using SetDiagHandler(), one can install a custom handler object of type CDiagHandler to process the messages posted via CNcbiDiag. The implementation of the CStreamDiagHandler in "ncbidiag.cpp" is a good example of how to do this.

```
/////////////////////////////////////////////////////////////////////////////
///
/// CDiagHandler --
///
/// Base diagnostic handler class.

class NCBI_XNCBI_EXPORT CDiagHandler
{
public:
 /// Destructor.
 virtual ~CDiagHandler(void) {}

 /// Post message to handler.
 virtual void Post(const SDiagMessage& mess) = 0;
};

/// Set the diagnostic handler using the specified diagnostic handler class.
NCBI_XNCBI_EXPORT
extern void SetDiagHandler(CDiagHandler* handler,
 bool can_delete = true);

/// Get the currently set diagnostic handler class.
NCBI_XNCBI_EXPORT
extern CDiagHandler* GetDiagHandler(bool take_ownership = false);
```

where:

```
/////////////////////////////////////////////////////////////////////////////
///
/// SDiagMessage --
///
/// Diagnostic message structure.
///
/// Defines structure of the "data" message that is used with message handler
/// function("func"), and destructor("cleanup").
/// The "func(..., data)" to be called when any instance of "CNcbiDiagBuffer"
/// has a new diagnostic message completed and ready to post.
/// "cleanup(data)" will be called whenever this hook gets replaced and
/// on the program termination.
/// NOTE 1: "func()", "cleanup()" and "g_SetDiagHandler()" calls are
/// MT-protected, so that they would never be called simultaneously
/// from different threads.
/// NOTE 2: By default, the errors will be written to standard error stream.

struct SDiagMessage {
 /// Initalize SDiagMessage fields.
 SDiagMessage(EDiagSev severity, const char* buf, size_t len,
 const char* file = 0, size_t line = 0,
 TDiagPostFlags flags = eDPF_Default, const char* prefix = 0,
 int err_code = 0, int err_subcode = 0,
 const char* err_text = 0);

 mutable EDiagSev m_Severity; ///< Severity level
 const char* m_Buffer; ///< Not guaranteed to be '\0'-terminated!
 size_t m_BufferLen; ///< Length of m_Buffer
 const char* m_File; ///< File name
 size_t m_Line; ///< Line number in file
 int m_ErrCode; ///< Error code
 int m_ErrSubCode; ///< Sub Error code
 TDiagPostFlags m_Flags; ///< Bitwise OR of "EDiagPostFlag"
 const char* m_Prefix; ///< Prefix string
 const char* m_ErrText; ///< Sometimes 'error' has no numeric code,
 ///< but can be represented as text

 // Compose a message string in the standard format(see also "flags"):
 // "<file>", line <line>: <severity>: [<prefix>] <message> [EOL]
 // and put it to string "str", or write to an output stream "os".

 /// Which write flags should be output in diagnostic message.
 enum EDiagWriteFlags {
 fNone = 0x0, ///< No flags
 fNoEndl = 0x01 ///< No end of line
 };

 typedef int TDiagWriteFlags; /// Binary OR of "EDiagWriteFlags"

 /// Write to string.
 void Write(string& str, TDiagWriteFlags flags = fNone) const;

 /// Write to stream.
 CNcbiOstream& Write(CNcbiOstream& os, TDiagWriteFlags flags = fNone) const;
};
```

Installing a new handler typically destroys the previous handler, which can be a problem if you need to keep the old handler around for some reason. There are two ways to address this issue:

- Declare an object of class CDiagRestorer at the top of the block of code in which you will be using your new handler. This will protect the old handler from destruction, and automatically restore it -- along with any other diagnostic settings -- when the block exits in any fashion. As such, you can safely use the result of calling GetDiagHandler () at the beginning of the block even if you have changed the handler within the block.
- Call GetDiagHandler(true) and then destroy the old handler yourself when done with it. This works in some circumstances in which CDiagRestorer is unsuitable, but places much more responsibility on your code.

For compatibility with older code, the diagnostic system also supports specifying simple callbacks:

```
/// Diagnostic handler function type.
typedef void (*FDiagHandler)(const SDiagMessage& mess);
```

```
/// Diagnostic cleanup function type.
typedef void (*FDiagCleanup)(void* data);

/// Set the diagnostic handler using the specified diagnostic handler class.
NCBI_XNCBI_EXPORT
extern void SetDiagHandler(CDiagHandler* handler,
 bool can_delete = true);
```

However, it is better to use the object-based interface for new code.

The following additional topics are discussed in this section:

- Formatting and Manipulators
- ERR_POST macro
- Turn on the Tracing

Formatting and Manipulators

To compose a diagnostic message with CNcbiDiag you can use the formatting operator "<<". It works practically the same way as operator "<<" for standard C++ output streams. CNcbiDiag class also has some CNcbiDiag-specific manipulators to control the message severity level:

- Info -- set severity level to eDiag_Info
- Warning -- set severity level to eDiag_Warning
- Error -- set severity level to eDiag_Error [default]
- Fatal -- set severity level to eDiag_Fatal
- Trace -- set severity level to eDiag_Trace

NOTE: whenever the severity level is changed, CNcbiDiag also automatically executes the following two manipulators:

- Endm -- means that the message is complete and to be flushed(via the global callback as described above)
- Reset -- directs to discard the content of presently composed message

The Endm manipulator also gets executed on the CNcbiDiag object destruction.

For example, this code:

```
int iii = 1234;
CNcbiDiag diag1;

diag1 << "Message1_Start " << iii;
 // message 1 is started but not ready yet
{ CNcbiDiag diag2; diag2 << Info << "Message2"; }
 // message 2 flushed in destructor
diag1 << "Message1_End" << Endm;
 // message 1 finished and flushed by "Endm"
diag1 << "Message1_1"; // will be flushed by the following "Warning"
diag1 << Warning << "Discard this warning" << ++iii << Reset;
 // message discarded
diag1 << "This is a warning " << iii;
diag1 << Endm;
```

will write to the diagnostic stream(if the latter was set with SetDiagStream()):

```
Error: Message1_Start 1234
Info: Message2
Error: Message1_End
Error: Message1_1
Warning: This is a warning 1235
```

ERR_POST macro

There is an ERR_POST(message) macro that can be used to shorten the error posting code. This macro is discussed in the chapter on Core Library.

Turn on the Tracing

The tracing (messages with severity level eDiag_Trace) is considered to be a special, debug-oriented feature, and therefore it is not affected by SetDiagPostLevel() and SetDiagDieLevel(). To turn the tracing on or off in your code you can use function SetDiagTrace().

By default, the tracing is off -- unless you assign environment variable $DIAG_TRACE to an arbitrary non-empty string (or, alternatively, you can set DIAG_TRACE entry in the [DEBUG] section of your registry to any non-empty value).

DebugDump: Take an Object State Snapshot

The following topics are discussed in this section:

- Terminology
- Requirements
- Architecture
- Implementation
- Examples

Debugging is an inevitable part of software development. When it comes to a "mystical" problem, one can spend days and days hunting for a glitch. So, being prepared is not just a "nice thing to have", it is a requirement.

When a system being developed crashes consistently, debugging is easy in the sense that the problem is reproducable. Were that all bugs like this! It is much more "fun", when the system crashes intermittently, under circumstances about which we have only a vague idea, if any, of the symptoms or the cause. What the developer needs in this case is information - the more the better. One short message ("Assertion failed") is good and a coredump is better, but we typically need a more user-friendly reporting of the program status at the point of failure.

One possible idea is to make the object tell about itself. That is, in case of trouble (but not necessarily trouble), an object could call a function that would report as much as possible about itself and other object it contains or to which it refers. During such operation the object should not do anything important, something that could potentially cause other problems. The diagnostic must of course be safe - it should only take a snapshot of an object's state and never alter that data.

Sure, DebugDump may cause problems by itself, even if everything is "correct". Let us say there are two objects, which "know" each other: Object A refers to Object B, while Object B refers to Object A (very common scenario in fact). Now dumping contents of Object A will cause dumping of Object B, which in turn will cause dumping of Object A, and so on until the stack overflows.

Terminology

So, dumping the object contents should look as a single function call, i.e. something like this:

```
Object name;
...
name.DebugDump(?);
```

The packet of information produced by such operation we call bundle. The class Object is most likely derived from other classes. The function should be called sequentially for each subclass, so it could print its data members. The piece of information produced by the subclass we call frame. The object may refer to other objects. Dumping of such object produces a sub-bundle, which consists of its own frames. To help fight cyclicity, we introduce depth of the dump. When an object being dumped wants to dump other objects it refers to, it should reduce the depth by one. If the depth is already zero, other objects should not be dumped.

Requirements

- The dump data should be separated from its representation. That is, the object should only supply data, something else should format it. Examples of formatting may include generating human-readable text or file in a special format (HTML, XML), or even transmitting the data over the network.
- Debug and release libraries should be compatible.
- It should be globally configurable as to whether the dump produces any output or not,

Architecture

Class CDebugDumpable is a special abstract base class. Its purpose is to define a virtual function DebugDump, which any derived class should implement. Another purpose is to store any global dump options. Any real dump should be initiated through a non-virtual function of this class - so, global option could be applied. Class CObject is derived from this class. So, any classes based on CObject may benefit from this functionality right away. Other classes may use this class as a base later on (e.g. using multiple inheritance).

Class CDebugDumpContext provides a generic dump interface for dumpable objects. The class has nothing to do with data representation. Its purpose is the ability to describe the location of where the data comes from, accept it from the object and transfer to the data formatter.

Class CDebugDumpFormatter defines the dump formatting interface. It is an abstract class.

Class CDebugDumpFormatterText is derived from CDebugDumpFormatter. Based on incoming data, it generates a human-readable text and passes it into any output stream (ostream).

In general, the system works like this:

- Client creates DebugDump formatter object (it could be an object of class CDebugDumpFormatterText or any other class derived from CDebugDumpFormatter) and passes it to a proper, non-virtual function of the object to be dumped. Bundle name is to be defined here - it can be anything, but a reasonable guess would be to specify the location of the call and the name of the object being dumped.

- CDebugDumpable analyses global settings, creates CDebugDumpContext object and calls virtual DebugDump() function of the object.
- DebugDump function of each subclass defines a frame name (which must be the type of the subclass), calls DebugDump function of a base class and finally logs its own data members. From within the DebugDump(), the object being dumped "sees" only CDebugDumpContext. It does not know any specifics about target format in which dump data will be eventually represented.

Implementation

The following topics are discussed in this section:

- CDebugDumpable
- CDebugDumpContext
- CDebugDumpFormatter

CDebugDumpable

The class is an abstract one. Global options are stored as static variable(s).

```
public:
 // Enable/disable debug dump
 static void EnableDebugDump(bool on);

 // Dump using text formatter
 void DebugDumpText(ostream& out,
 const string& bundle,
 unsigned int depth) const;

 // Dump using external dump formatter
 void DebugDumpFormat(CDebugDumpFormatter& ddf,
 const string& bundle,
 unsigned int depth) const;

 // Function that does the dump - to be overloaded
 virtual void DebugDump(CDebugDumpContext ddc,
 unsigned int depth) const = 0;
```

Any derived class must impelement a relevant DebugDump function.

CDebugDumpContext

The class defines a public dump interface for a client object. It receives the data from the object and decides when and what functions of dump formatter to call.

The dump interface looks like this:

```
public:
 CDebugDumpContext(CDebugDumpFormatter& formatter,
 const string& bundle);
 // This is not exactly a copy constructor -
 // this mechanism is used internally to find out
 // where are we on the Dump tree
 CDebugDumpContext(CDebugDumpContext& ddc);
 CDebugDumpContext(CDebugDumpContext& ddc, const string& bundle);

public:
 // First thing in DebugDump() function - call this function
 // providing class type as the frame name
 void SetFrame(const string& frame);
 // Log data in the form [name, data, comment]
 // All data is passed to a formatter as string, still sometimes
 // it is probably worth to emphasize that the data is REALLY a
 // string
 void Log(const string& name,
 const string& value,
 bool is_string = true,
 const string& comment = kEmptyStr
 );
 void Log(const string& name,
 bool value,
 const string& comment = kEmptyStr
 );
 void Log(const string& name,
 long value,
 const string& comment = kEmptyStr
 );
 void Log(const string& name,
 unsigned long value,
 const string& comment = kEmptyStr
 );
 void Log(const string& name,
 double value,
 const string& comment = kEmptyStr
 );
 void Log(const string& name,
 const void* value,
 const string& comment = kEmptyStr
 );
 void Log(const string& name,
 const CDebugDumpable* value,
 unsigned int depth
 );
```

A number of overloaded Log functions is provided for convenience only.

CDebugDumpFormatter

This abstract class defines dump formatting interface:

```
public:
 virtual bool StartBundle(unsigned int level, const string& bundle) = 0;
 virtual void EndBundle( unsigned int level, const string& bundle) = 0;

 virtual bool StartFrame( unsigned int level, const string& frame) = 0;
 virtual void EndFrame( unsigned int level, const string& frame) = 0;
 virtual void PutValue( unsigned int level, const string& name,
 const string& value, bool is_string,
 const string& comment) = 0;
```

Examples

Supposed that there is an object m_ccObj of class CSomeObject derived from CObject. In order to dump it into the standard cerr stream, one should do one of the following:

```
m_ccObj.DebugDumpText(cerr, "m_ccObj", 0);
```

or

```
{
 CDebugDumpFormatterText ddf(cerr);
 m_ccObj.DebugDumpFormat(ddf, "m_ccObj", 0);
}
```

The DebugDump function should look like this:

```
void CSomeObject::DebugDump(CDebugDumpContext ddc, unsigned int depth) const
{
 ddc.SetFrame("CSomeObject");
 CObject::DebugDump(ddc,depth);
 ddc.Log("m_1", m_1);
 ddc.Log("m_2", m_2);
 ... etc for each data member
}
```

Exception Handling (*) in the NCBI C++ Toolkit

The following topics are discussed in this section:

- NCBI C++ Exceptions
- The CErrnoTemplException Class
- The CParseTemplException Class
- Macros for Standard C++ Exception Handling
- Exception Tracing

NCBI C++ Exceptions

C++ exceptions is a standard mechanism of communicating abnormal or unexpected events to a higher execution context. By throwing an exception a piece of code says it was unable to complete the task and it is up to others to decide what to do next.

What the standard mechanism lacks is backlog, history of unfinished tasks and its consequences. Say for instance, a program tries to load some data from a database. An exception occurs, which says a connection to some port could not be created -- so what? How meaningfull is it? What did the program try to do? Where did the request for the connection come from?

Another problem is analyzing and handling exceptions in a program. When an exception is caught, what is known for sure is only that something bad has happened -- but what exactly? The standard exception has only type (exception class) and a text message. The latter probably makes sense for a human, but not for a program. The former does not seem to be clear enough.

The following topics are discussed in this section:

Requirements

In order for exceptions to be more useful, they should meet the following requirements:

- Exceptions should contain information about where exactly has it been thrown -- for a human.
- Exceptions should have a numeric id -- for a program.
- It should be possible to create a stack of exceptions -- to accumulate a backlog of events (unfinished jobs) which caused the problem. Still, for a client, it should look like a single exception. That is, a client should be able to ignore completely the compound structure of the exception being thrown and still get some meaningful information.
- The system should provide for the ability to analyze the exception backlog and possibly print information about each exception separately.
- It should be possible to report the exception data into an arbitrary output channel and possibly format it differently for each channel.

Architecture

Each subsystem (library) has its own type of exceptions. It may have several types, if necessary, but all of them should be derived from a single base class (which in turn is derived from a system-wide base class). So, the type of an exception uniquely identifies the library which produced it.

Each exception has a numeric id, which is unique throughout the subsystem. Such an id gives an unambiguous description of the problem occurred. Each id is associated with a text message. Strictly speaking, there is only one message associated with a given id, so there is no need to include the message in the exception itself -- it could be taken from an external source. Still, we suggest using the message -- it serves as an additional comment. Also, it does not restrict us from using an external source of messages in the future.

Each exception has information about the location where it has been thrown -- file name and line number.

An exception can have a reference to the "lower level" one, which makes it possible to analyze the backlog. Naturally, such a backlog cannot be created automatically - it is a developer's responsibility. The system only provides the mechanism, it does not solve problems by itself. The developer is supposed to catch exceptions in proper places and re-throw them with the backlog information added.

The exception constructor's mandatory parameters include location information, exception id and a message. This constructor is to be used at the lower level, when the exception is thrown initially. At higher levels we need a constructor, which would accept the exception from the lower level as one of its parameters.

The NCBI exception mechanism has a sophisticated reporting mechanism -- the standard exception::what() function is definitely not enough. There are three groups of reporting mechanisms:

- exception formats its data by itself and either returns the result as a string or puts it into an output stream;
- client provides an external exception data formatter;
- NCBI standard diagnostic mechanism is used.

Implementation

The following topics are discussed in this section:

- CException
- Derived exceptions
- Reporting an exception
- CExceptionReporter
- Choosing and analyzing error codes

CException

There is a single system-wide exception base class -- CException. Each subsystem **must** implement its own type of exceptions, which must be be derived from this class. The class defines basic requirements of an exception construction, backlog and reporting mechanisms.

The CException constructor includes location information, exception id and a message. Each exception class defines its own error codes. So, the error code "by itself" is meaningless -- one should also know the the exception class, which produced it.

```
/// Constructor.
///
/// When throwing an exception initially, "prev_exception" must be 0.
CException(const char* file, int line,
 const CException* prev_exception,
 EErrCode err_code,const string& message) throw();
```

To make it easier to throw/re-throw an exception, the following macros are defined:

```
NCBI_THROW(exception_class, err_code, message)
NCBI_RETHROW(prev_exception, exception_class, err_code,message)
NCBI_RETHROW_SAME(prev_exception, message)
```

The last one (NCBI_RETHROW_SAME) re-throws the same exception with backlog information added.

The CException class has numerous reporting methods (the contents of reports is defined by diagnostics post flags):

```
 /// Standard report (includes full backlog).
 virtual const char* what(void) const throw();
```

```
/// Report the exception.
///
/// Report the exception using "reporter" exception reporter.
/// If "reporter" is not specified (value 0), then use the default
/// reporter as set with CExceptionReporter::SetDefault.
void Report(const char* file, int line,
const string& title, CExceptionReporter* reporter = 0,
TDiagPostFlags flags = eDPF_Trace) const;

/// Report this exception only.
///
/// Report as a string this exception only. No backlog is attached.
string ReportThis(TDiagPostFlags flags = eDPF_Trace) const;

/// Report all exceptions.
///
/// Report as a string all exceptions. Include full backlog.
string ReportAll (TDiagPostFlags flags = eDPF_Trace) const;

/// Report "standard" attributes.
///
/// Report "standard" attributes (file, line, type, err.code, user message)
/// into the "out" stream (this exception only, no backlog).
void ReportStd(ostream& out, TDiagPostFlags flags = eDPF_Trace) const;

/// Report "non-standard" attributes.
///
/// Report "non-standard" attributes (those of derived class) into the
/// "out" stream.
virtual void ReportExtra(ostream& out) const;

/// Enable background reporting.
///
/// If background reporting is enabled, then calling what() or ReportAll()
/// would also report exception to the default exception reporter.
/// @return
/// The previous state of the flag.
static bool EnableBackgroundReporting(bool enable);
```

Also, the following macro is defined that calls the CExceptionReporter::ReportDefault() method to produce a report for the exception:

```
NCBI_REPORT_EXCEPTION(title,e)
```

Finally, the following data access functions help to analyze exceptions from a program:

```
/// Get class name as a string.
virtual const char* GetType(void) const;

/// Get error code interpreted as text.
virtual const char* GetErrCodeString(void) const;
```

```
 /// Get file name used for reporting.
 const string& GetFile(void) const;

 /// Get line number where error occurred.
 int GetLine(void) const;

 /// Get error code.
 EErrCode GetErrCode(void) const;

 /// Get message string.
 const string& GetMsg (void) const;

 /// Get "previous" exception from the backlog.
 const CException* GetPredecessor(void) const;
```

Derived exceptions

The only requirement for a derived exception is to define error codes as well as its textual representation. Implementation of several other functions (e.g. constructors) are, in general, pretty straightforward -- so we put it into a macro definition, NCBI_EXCEPTION_DEFAULT. Please note, this macro can only be used when the derived class has no additional data members. Here is an example of an exception declaration:

```
class CSubsystemException : public CException
{
public:
 /// Error types that subsystem can generate.
 enum EErrCode {
 eType1, ///< Meaning of eType1
 eType2 ///< Meaning of eType2
 };

 /// Translate from the error code value to its string representation.
 virtual const char* GetErrCodeString(void) const
 {
 switch (GetErrCode()) {
 case eType1: return "eType1";
 case eType2: return "eType2";
 default: return CException::GetErrCodeString();
 }
 }

 // Standard exception boilerplate code.
 NCBI_EXCEPTION_DEFAULT(CSubsystemException, CException);
};
```

In case the derived exception has data members not found in the base class, it should also implement its own ReportExtra method -- to report this non-standard data.

Reporting an exception

There are several way to report an NCBI C++ exception:

- An exception is capable of formatting its own data, returning a string (or a pointer to a string buffer). Each exception report occupies one line. Still, since an exception may contain a backlog of previously thrown exceptions, the resulting report could contain several lines of text - one for each exception thrown. The report normally contains information about the location from which the exception has been thrown, the text representation of the exception class and error code, and a description of the error. The content of the report is defined by diagnostics post flags. The following methods generate reports of this type:

  ```
  /// Standard report (includes full backlog).
  virtual const char* what(void) const throw();

  /// Report the exception.
  ///
  /// Report the exception using "reporter" exception reporter.
  /// If "reporter" is not specified (value 0), then use the default
  /// reporter as set with CExceptionReporter::SetDefault.
  void Report(const char* file, int line,
  const string& title, CExceptionReporter* reporter = 0,
  TDiagPostFlags flags = eDPF_Trace) const;

  /// Report this exception only.
  ///
  /// Report as a string this exception only. No backlog is attached.
  string ReportThis(TDiagPostFlags flags = eDPF_Trace) const;

  /// Report all exceptions.
  ///
  /// Report as a string all exceptions. Include full backlog.
  string ReportAll (TDiagPostFlags flags = eDPF_Trace) const;

  /// Report "standard" attributes.
  ///
  /// Report "standard" attributes (file, line, type, err.code, user message)
  /// into the "out" stream (this exception only, no backlog).
  void ReportStd(ostream& out, TDiagPostFlags flags = eDPF_Trace) const;
  ```

  Functions what() and ReportAll() may also generate a background report - the one generated by a default exception reporter. This feature can be disabled by calling the static method

  ```
  CException::EnableBackgroundReporting(false);
  ```

- A client can provide its own exception reporter. An object of this class may either use exception data access functions to create its own reports, or redirect reports into its own output channel(s). While it is possible to specify the reporter in the CException::Report() function, it is better if the same reporting functions are used for exceptions, to install the reporter as a default one instead, using CExceptionReporter::SetDefault(const CExceptionReporter* handler); static function, and use the standard NCBI_REPORT_EXCEPTION macro in the program.

- Still another way to report an exception is to use the standard diagnostic mechanism provided by NCBI C++ toolkit. In this case the code to generate the report would look like this:

  ```
  try {
  ...
  ```

```
} catch (CException& e) {
ERR_POST_X(1, Critical << "Your message here." << e);
}
```

CExceptionReporter

One of possible ways to report an exception is to use an external "reporter" modeled by the CExceptionReporter abstract class. The reporter is an object that formats exception data and sends it to its own output channel. A client can install its own, custom exception reporter. This is not required, though. In case the default was not set, the standard NCBI diagnostic mechanism is used.

The CExceptionReporter is an abstract class, which defines the reporter interface:

```
 /// Set default reporter.
 static void SetDefault(const CExceptionReporter* handler);

 /// Get default reporter.
 static const CExceptionReporter* GetDefault(void);

 /// Enable/disable using default reporter.
 ///
 /// @return
 /// Previous state of this flag.
 static bool EnableDefault(bool enable);

 /// Report exception using default reporter.
 static void ReportDefault(const char* file, int line,
 const string& title, const CException& ex,
 TDiagPostFlags flags = eDPF_Trace);

 /// Report exception with _this_ reporter
 virtual void Report(const char* file, int line,
 const string& title, const CException& ex,
 TDiagPostFlags flags = eDPF_Trace) const = 0;
```

Choosing and analyzing error codes

Choosing and interpreting error codes can potentially create some problems because each exception class has its own error codes, and interpretation. Error codes are implemented as an enum type, EErrCode, and the enumerated values are stored internally in a program as numbers. So, the same number can be interpreted incorrectly for a different exception class than the one in which the enum type was defined. Say for instance, there is an exception class, which is derived from CSubsystemException -- let us call it CBiggersystemException -- which also defines two error codes: eBigger1 and eBigger2:

```
class CBiggersystemException : public CSubsystemException
{
public:
 /// Error types that subsystem can generate.
 enum EErrCode {
 eBigger1, ///< Meaning of error code, eBigger1
 eBigger2 ///< Meaning of error code, eBigger2
 };

 /// Translate from the error code value to its string representation.
 virtual const char* GetErrCodeString(void) const
 {
 switch (GetErrCode()) {
 case eBigger1: return "eBigger1";
 case eBigger2: return "eBigger2";
 default: return CException::GetErrCodeString();
 }
 }

 // Standard exception boilerplate code.
 NCBI_EXCEPTION_DEFAULT(CBiggersystemException, CSubsystemException);
};
```

Now, suppose an exception CBiggersystemException has been thrown somewhere. On a higher level it has been caught as CSubsystemException. It is easy to see that the error code returned by the CSubsystemException object would be completely meaningless: the error code of CBiggersystemException cannot be interpreted in terms of CSubsystemException.

One reasonable solution seems to be isolating error codes of different exception classes -- by assigning different numeric values to them. And this has to be done by the developer. Such isolation should only be done within each branch of derivatives only. Another solution is to make sure that the exception in question does belong to the desired class, not to any intermediate classes in the derivation hierarchy. The template function UppermostCast() can be used to perform this check:

```
/// Return valid pointer to uppermost derived class only if "from" is
_really_
/// the object of the desired type.
///
/// Do not cast to intermediate types (return NULL if such cast is
attempted).
template <class TTo, class TFrom>
const TTo* UppermostCast(const TFrom& from)
{
 return typeid(from) == typeid(TTo) ? dynamic_cast<const TTo*>(&from) : 0;
}
```

UppermostCast() utilizes the runtime information using the typeid() function, and dynamic cast conversion to return either a pointer to "uppermost" exception object or NULL.

The following shows how UppermostCast() can be used to catch the correct error types:

```
try {
 ...
 NCBI_THROW(CBiggersystemException,eBigger1,"your message here");
 ...
}
catch (CSubsystemException& e) {
 // call to UppermostCast<CSubsystemException>(e) would return 0 here!
 // which means that "e" was actually the object of a different class
 const CBiggersystemException *p = UppermostCast<CBiggersystemException>(e);
 if (p) {
 switch (p->GetErrCode()) {
 case CBiggersystemException::eBigger1:
 ...
 break;
 case CBiggersystemException::eBigger2:
 ...
 break;
 default:
 ...
 break;
 }
 }
 NCBI_RETHROW_SAME(e,"your message here");
}
```

It is possible to use the runtime information to do it even better. Since GetErrCode function is non-virtual, it might check the type of the object, for which it has been called, against the type of the class to which it belongs. If these two do not match, the function returns invalid error code. Such code only means that the caller did not know the correct type of the exception, and the function is unable to interpret it.

Examples

The following topics are discussed in this section:

- Throwing an exception
- Reporting an exception

Throwing an exception

It is important to remember that the system only provides a mechanism to create a backlog of unfinished tasks, it does not create this backlog automatically. It is up to developer to catch exceptions and re-throw them with the backlog information added. Here is an example of throwing CSubsystemException exception:

```
... // your code
NCBI_THROW(CSubsystemException,eType1,"your message here");
...
```

The code that catches, and possibly re-throws the exception might look like this:

```
try {
 ... // your code
} catch (CSubsystemException& e) {
 if (e.GetErrCode() == CSubsystemException::eType2) {
 ...
 } else {
 NCBI_RETHROW(e, CSubsystemException, eType1, " your message here")
 }
} catch (CException& e) {
 NCBI_RETHROW(e, CException, eUnknown, "your message here")
}
```

Reporting an exception

There are a number of ways to report CException, for example:

```
try {
 ... // your code
} catch (CSubsystemException& e) {
 NCBI_REPORT_EXCEPTION("your message here", e);
 ERR_POST_X(CMyException::eMyErrorXyz, Critical << "message" << e);
 cerr << e.ReportAll();
 cerr << e.what();
 e.Report(__FILE__, __LINE__, "your message here");
}
```

We suggest using NCBI_REPORT_EXCEPTION(title,e) macro (which is equivalent to calling e.Report(__FILE__,__LINE__,title)) - it redirects the output into standard diagnostic channels and is highly configurable.

The CErrnoTemplException Class

The CErrnoTemplException class is a template class used for generating error exception classes:

```
/////////////////////////////////////////////////////////////////////////////
///
/// CErrnoTemplException --
///
/// Define template class for easy generation of Errno-like exception
classes.

template<class TBase> class CErrnoTemplException :
 public CErrnoTemplExceptionEx<TBase, CStrErrAdapt::strerror>
{
public:
 /// Parent class type.
 typedef CErrnoTemplExceptionEx<TBase, CStrErrAdapt::strerror> CParent;

 /// Constructor.
 CErrnoTemplException<TBase>(const char* file,int line,
 const CException* prev_exception,
 typename CParent::EErrCode err_code,const string& message) throw()
 : CParent(file, line, prev_exception,
 (typename CParent::EErrCode) CException::eInvalid, message)
 NCBI_EXCEPTION_DEFAULT_IMPLEMENTATION_TEMPL(CErrnoTemplException<TBase>,
CParent)
};
```

The template class is derived form another template class, the ErrnoTemplExceptionEx which implements a parent class with the template parameter TBase. The parent ErrnoTemplExceptionEx class implements the basic exception methods such as ReportExtra

(), GetErrCode(), GetErrno(), GetType(). The ErrnoTemplExceptionEx class has an int data member called m_Errno. The constructor automatically adds information about the most recent error state as obtained via the global system variable errno to this data member.

The constructor for the derived CErrnoTemplException class is defined in terms of the NCBI_EXCEPTION_DEFAULT_IMPLEMENTATION_TEMPL macro which defines the program code for implementing the constructor.

The TBase template parameter is an exception base class such as CException or CCoreException, or another class similar to these. The CStrErrAdapt::strerror template parameter is a function defined in an adaptor class for getting the error description string. The CErrnoTemplException has only one error core - eErrno defined in the parent class, ErrnoTemplExceptionEx. To analyze the actual reason of the exception one should use GetErrno() method:

```
int GetErrno(void) const;
```

The CErrnoTemplException is used to create exception classes. Here is an example of how the CExecException class is created from CErrnoTemplException. In this example, the TBase template parameter is the exception base class CCoreException:

```
/////////////////////////////////////////////////////////////////////////////
///
/// CExecException --
///
/// Define exceptions generated by CExec.
///
/// CExecException inherits its basic functionality from
/// CErrnoTemplException<CCoreException> and defines additional error codes
/// for errors generated by CExec.

class NCBI_XNCBI_EXPORT CExecException :
 public CErrnoTemplException<CCoreException>
{
public:
 /// Error types that CExec can generate.
 enum EErrCode {
 eSystem, ///< System error
 eSpawn ///< Spawn error
 };

 /// Translate from the error code value to its string representation.
 virtual const char* GetErrCodeString(void) const
 {
 switch (GetErrCode()) {
 case eSystem: return "eSystem";
 case eSpawn: return "eSpawn";
 default: return CException::GetErrCodeString();
 }
 }

 // Standard exception boilerplate code.
 NCBI_EXCEPTION_DEFAULT(CExecException,
 CErrnoTemplException<CCoreException>);
};
```

The CParseException Class

The CParseTemplException is a template class whose parent class is the template parameter TBase. The CParseTemplException class includes an additional int data member, called m_Pos. This class was specifically defined to support complex parsing tasks, and its constructor requires that positional information be supplied along with the description message. This makes it impossible to use the standard NCBI_THROW macro to throw it, so we defined two additional macros:

```
/// Throw exception with extra parameter.
///
/// Required to throw exceptions with one additional parameter
/// (e.g. positional information for CParseException).
#define NCBI_THROW2(exception_class, err_code, message, extra) \
 throw exception_class(__FILE__, __LINE__, \
 0,exception_class::err_code, (message), (extra))

/// Re-throw exception with extra parameter.
///
/// Required to re-throw exceptions with one additional parameter
/// (e.g. positional information for CParseException).
#define NCBI_RETHROW2(prev_exception,exception_class,err_code,message,extra)
\
 throw exception_class(__FILE__, __LINE__, \
 &(prev_exception), exception_class::err_code, (message), (extra))
```

Macros for Standard C++ Exception Handling

The C++ throw() statement provides a mechanism for specifying the types of exceptions that may be thrown by a function. Functions that do **not** include a throw() statement in their declaration can throw any type of exception, but where the throw() statement **is** used, undeclared exception types that are thrown will cause std::unexpected() to be raised. Various compilers handle these events differently, and the first two macros listed in Table 5, (THROWS (()), THROWS_NONE, are provided to support platform-independent exception specifications.

The catch macros provide uniform, routine exception handling with minimal effort from the programmer. We provide convenient STD_CATCH_*() macros to print formatted messages to the application's diagnostic stream. For example, if F() throws an exception of the form:

```
throw std::runtime_error(throw-msg)
```

then

```
try {F();}
STD_CATCH_X(1, catch-msg); // here 1 is the error subcode
```

will generate a message of the form:

```
Error: (101.1) [catch-msg] Exception: throw-msg
```

In this output, the (101.1) indicates the error code (defined in the module's error_codes.hpp header) and the subcode passed to STD_CATCH_X.

In this example, the generated message starts with the Error tag, as that is the severity level for the default diagnostic stream. User-defined classes that are derived from std::exception will be treated uniformly in the same manner. The throw clause in this case creates a new instance of std::runtime_error whose data member desc is initialized to throw-msg. When the exception is then caught, the exception's member function what() can be used to retrieve that message.

The STD_CATCH_ALL_X macro catches all exceptions. If however, the exception caught is **not** derived from std::exception, then the catch clause cannot assume that what() has been defined for this object, and a default message is generated of the form:

```
Error: (101.1) [catch-msg] Exception: Unknown exception
```

Again, the (101.1) indicates the error code (defined in the module's error_codes.hpp header) and the subcode passed to STD_CATCH_ALL_X.

Exception Tracing

Knowing exactly where an exception first occurs can be very useful for debugging purposes. CException class has this functionality built in, so it is highly recommended to use exceptions derived from it. In addition to this a set of THROW*_TRACE() macros defined in the NCBI C++ Toolkit combine exception handling with trace mechanisms to provide such information.

The most commonly used of these macros, THROW1_TRACE(class_name, init_arg), instantiates an exception object of type class_name using init_arg to initialize it. The definition of this macro is:

```
/// Throw trace.
///
/// Combines diagnostic message trace and exception throwing. First the
/// diagnostic message is printed, and then exception is thrown.
///
/// Arguments can be any exception class with the specified initialization
/// argument. The class argument need not be derived from std::exception as
/// a new class object is constructed using the specified class name and
/// initialization argument.
///
/// Example:
/// - THROW1_TRACE(runtime_error, "Something is weird...");
# define THROW1_TRACE(exception_class, exception_arg) \
 throw NCBI_NS_NCBI::DbgPrint(DIAG_COMPILE_INFO, \
 exception_class(exception_arg), #exception_class)
```

From the throw() statement here, we see that the object actually being thrown by this macro is the value returned by DbgPrint(). DbgPrint() in turn calls DoDbgPrint(). The latter is an overloaded function that simply creates a diagnostic stream and writes the file name, line number, and the exception's what() message to that stream. The exception object (which is of type class_name) is then the value returned by DbgPrint().

More generally, three sets of THROW*_TRACE macros are defined:

- THROW0_TRACE(exception_object)

- THROW0p_TRACE(exception_object)
- THROW0np_TRACE(exception_object)
- THROW1_TRACE(exception_class, exception_arg)
- THROW1p_TRACE(exception_class, exception_arg)
- THROW1np_TRACE(exception_class, exception_arg)
- THROW_TRACE(exception_class, exception_args)
- THROWp_TRACE(exception_class, exception_args)
- THROWnp_TRACE(exception_class, exception_args)

The first three macros (THROW0*_TRACE) take a single argument, which may be a newly constructed exception, as in:

```
THROW0_TRACE(runtime_error("message"))
```

or simply a printable object to be thrown, as in:

```
THROW0_TRACE("print this message")
```

The THROW0_TRACE macro accepts either an exception object or a string as the argument to be thrown. The THROW0p_TRACE macro generalizes this functionality by accepting any printable object, such as complex(1,3), as its single argument. Any object with a defined output operator is, of course, printable. The third macro generalizes this one step further, and accepts aggregate arguments such as vector<T>, where T is a printable object. Note that in cases where the object to be thrown is not a std::exception, you will need to use STD_CATCH_ALL_{X|XX} or a customized catch statement to catch the thrown object.

The remaining six macros accept two arguments: an "exception" class name and an initialization argument, where both arguments are also passed to the trace message. The class argument need not actually be derived from std::exception, as the pre-processor simply uses the class name to construct a new object of that type using the initialization argument. All of the THROW1*_TRACE macros assume that there is a single initialization argument. As in the first three macros, THROW1_TRACE(), THROW1p_TRACE() and THROW1np_TRACE() specialize in different types of printable objects, ranging from exceptions and numeric and character types, to aggregate and container types.

The last three macros parallel the previous two sets of macros in their specializations, and may be applied where the exception object's constructor takes multiple arguments. (See also the discussion on Exception handling).

It is also possible to specify that execution should abort immediately when an exception occurs. By default, this feature is not activated, but the SetThrowTraceAbort() function can be used to activate it. Alternatively, you can turn it on for the entire application by setting either the $ABORT_ON_THROW environment variable, or the application's registry ABORT_ON_THROW entry (in the [DEBUG] section) to an arbitrary non-empty value.

Table 1. Command line parameters available for use to any program that uses CNcbiApplication

Flag	Description	Example
-h	Print description of the application's command line parameters.	theapp -h
-logfile	Redirect program's log into the specified file	theapp -logfile theapp_log
-conffile	Read the program's configuration data from the specified file	theapp -conffile theapp_cfg

Table 2. Enabling Tracing

C++ toolkit API	**Configuration file**	**Environment**
call: SetDiagTrace(eDT_Enable);	define DIAG_TRACE entry in the DEBUG section: [DEBUG] DIAG_TRACE=1	define DIAG_TRACE environment variable: set DIAG_TRACE=1

Table 3. Changing severity level for diagnostic messages

C++ toolkit API	Configuration file	Environment
call: SetDiagPostLevel(EDiagSev postSev); Valid arguments are eDiag_Info, eDiag_Warning, eDiag_Error, eDiag_Critical, eDiag_Fatal.	define DIAG_POST_LEVEL entry in the DEBUG section: [DEBUG] DIAG_POST_LEVEL=Info Valid values are Info, Warning, Error, Critical, Fatal.	define DIAG_POST_LEVEL environment variable: set DIAG_POST_LEVEL=Info Valid values are Info, Warning, Error, Critical, Fatal.

Table 4. Setting up trace options for connection library

	Configuration file	**Environment**
Connection parameters:	define DEBUG_PRINTOUT entry in the CONN section: [CONN] DEBUG_PRINTOUT=TRUE Valid values are TRUE, or YES, or SOME.	define CONN_DEBUG_PRINTOUT environment variable: set CONN_DEBUG_PRINTOUT=TRUE Valid values are TRUE, or YES, or SOME.
All data:	define DEBUG_PRINTOUT entry in the CONN section: [CONN] DEBUG_PRINTOUT=ALL Valid values are ALL, or DATA.	define CONN_DEBUG_PRINTOUT environment variable: set CONN_DEBUG_PRINTOUT=ALL Valid values are ALL, or DATA.

Table 5. Platform Independent Exception Macros

Macro	**C++ Equivalent**	**Synopsis**
THROWS((types))	throw(types)	Defines the type of exceptions thrown by the given function. types may be a single object type or a comma delimited list.
THROWS_NONE	throw()	Specifies that the given function throws no exceptions.
STD_CATCH_X(subcode, message)	catch(std::exception)	Calls STD_CATCH_XX() using the currently selected error code name.
STD_CATCH_XX(name, subcode, message)	catch(std::exception)	Provides uniform handling of all exceptions derived from std::exception using the given error code name, subcode, and message. Does not catch exceptions *not* derived from std::exception.
STD_CATCH_ALL_X(subcode, message)	catch(...)	Calls STD_CATCH_ALL_XX() using the currently selected error code name.
STD_CATCH_ALL_XX(name, subcode, message)	catch(...)	Applies STD_CATCH_XX() to std::exception derived objects. Catches non-standard exceptions and generates an "Unknown exception" message using the given error code name, subcode, and message.

23: Distributed Computing

Created: May 14, 2007.
Last Update: April 23, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter describes the NCBI GRID framework. This framework allows creating, running and maintaining a scalable, load-balanced and fault-tolerant pool of network servers (Worker Nodes).

Note: Users within NCBI may find additional information on the internal Wiki page.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Getting Help

Users at NCBI have the following sources for help:

- JIRA for submitting a request or bug report. Select project C++ Toolkit and component GRID.
- Mailing lists:
  - The grid mailing list (grid@ncbi.nlm.nih.gov) for general GRID-related discussion and announcements.
  - The grid-core mailing list (grid-core@ncbi.nlm.nih.gov) for getting help using or trouble-shooting a GRID service.
- The GRID developers:
  - Dmitry Kazimirov for questions about Client-side APIs, Worker Nodes, NetCache and NetSchedule deployment, auxiliary tools and utilities, administration - setup, installation, and upgrades.
  - Andrei Gourianov for NetCache server questions.
  - Sergey Satskiy for NetSchedule server questions.
  - David McElhany for questions about related documentation in the C++ Toolkit book.
  - Denis Vakatov for supervision questions.

GRID Overview

The following sections provide an overview of the GRID system:

- Purpose
- Components
- Architecture and Data Flow
- The GRID Farm

Purpose

The NCBI GRID is a framework to create, run and maintain a scalable, load-balanced and fault-tolerant pool of network servers (Worker Nodes).

It includes independent components that implement distributed data storage and job queueing. It also provides APIs and frameworks to implement worker nodes and job submitters.

Worker nodes can be written from scratch, but there are also convenience APIs and frameworks to easily create worker nodes out of existing C++ CGI code, or even from CGI or command-line scripts and executables.

There is also a GRID farm where developers can jump-start their distributed computation projects.

Two PowerPoint presentations have additional information about the NCBI GRID:

- ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/DOC/PPT/GRID-Dec14-2006/GRID_Dec14_2006.pps
- ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/DOC/PPT/NCBI-Grid.ppt

Components

The NCBI GRID framework is built of the following components:

1. Network job queue (NetSchedule)
2. Network data storage (NetCache)
3. Server-side APIs and tools to develop Worker Nodes:
   a. Out of an existing command-line executable
   b. Out of an existing CGI executable
   c. Out of an existing CGI code (if it's written using the NCBI C++ CGI framework)
   d. Create a GRID Worker Node from scratch
4. Client-side API
5. Remote CGI -- enables moving the actual CGI execution to the grid.
6. GRID Utilities for remote administration, monitoring, retrieval and submission (netschedule_control, netcache_control, ns_remote_job_control, ns_submit_remote_job, etc.)

All these components are fully portable, in the sense that they can be built and then run and communicate with each other across all platforms that are supported by the NCBI C++ Toolkit (UNIX, MS-Windows, MacOSX).

The NetCache and NetSchedule components can be used independently of each other and the rest of the grid framework - they have their respective client APIs. Worker Nodes get their tasks from NetSchedule, and may also use NetCache to get the data related to the tasks and to store the results of computation. Remote-CGI allows one to easily convert an existing CGI into a back-end worker node -- by a minor, 1 line of source code, modification. It can solve the infamous "30-sec CGI timeout" problem.

All these components can be load-balanced and are highly scalable. For example, one can just setup 10 NetCache servers or 20 Worker Nodes on new machines, and the storage/computation throughput would increase linearly. Also, NetCache and NetSchedule are lightning-fast.

To provide more flexibility, load balancing, and fault-tolerance, it is highly advisable to pool NetSchedule and NetCache servers using NCBI Load Balancer and Service Mapper (LBSM).

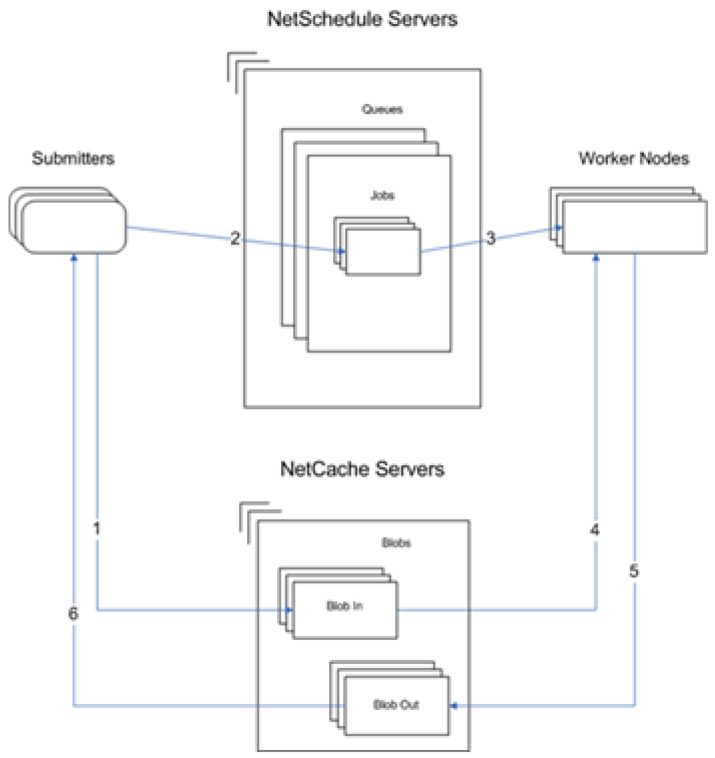
Architecture and Data Flow

NetSchedule and NetCache servers create a media which Submitters and Worker Nodes use to pass and control jobs and related data:

1. Submitter prepares input data and stores it in the pool of NetCache servers, recording keys to the data in the job's description.
2. Submitter submits the job to the appropriate queue in the pool of NetSchedule servers.
3. Worker Node polls "its" queue on the NetSchedule servers for jobs, and takes the submitted job for processing.

4. Worker Node retrieves the job's input data from the NetCache server(s) and processes the job.
5. Worker Node stores the job's results in NetCache and changes the job's status to "done" in NetSchedule.
6. Submitter sees that the job is done and reads its result from NetCache.

The following diagram illustrates this flow of control and data:



The GRID Farm

To help developers jump-start their distributed computation projects, there is a small farm of machines for general use, running:

- Several flavors of job queues
- Several flavors of network data storage
- A framework to run and maintain users' Worker Nodes

NOTE: Most of the GRID components can be deployed or used outside of the GRID framework (applications can communicate with the components directly via the components' own client APIs). However, in many cases it is beneficial to use the whole GRID framework from the start.

NCBI users can find more information on the GRID farm Wiki page.

Worker Nodes

The following sections describe how to create, configure and run worker nodes:

- Create a GRID Worker Node from scratch
- Converting an existing CGI application into a GRID Node

- Wrapping an existing CGI application into a GRID Node
- Wrapping an existing command-line application into a GRID Node
- Worker Node Cleanup Procedure

Create a GRID Worker Node from scratch

The following sections describe how to Create a GRID Worker Node from scratch:

- Purpose
- Diagram

Purpose

Framework to create a multithreaded server that can run on a number of machines and serve the requests using NetSchedule and NetCache services to exchange the job info and data.

Diagram

ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/DOC/PPT/IMAGES/GRID_Dec14_2006/Slide3.PNG

Converting an existing CGI application into a GRID Node

The following sections describe how to convert an existing CGI application into a GRID node:

- Purpose
- Converting a CGI into a Remote-CGI server
- Diagram
- Features and benefits

Purpose

With a rather simple and formal conversion, a CGI's real workload can be moved from the Web servers to any other machines. It also helps to work around the infamous "30-sec Web timeout problem".

Converting a CGI into a Remote-CGI server

1. Modify the code of your original CGI to make it a standalone Remote-CGI server (Worker Node). The code conversion is very easy and formal:
   a. Change application's base class from CCgiApplication to CRemoteCgiApp
   b. Link the application with the library xgridcgi rather than with xcgi
2. Replace your original CGIs by a one-line shell scripts that calls "remote CGI gateway" (cgi2rcgi.cgi) application.
3. Match "remote CGI gateways" against Remote-CGI servers:
   a. Ask us to register your remote CGI in the GRID framework
   b. Define some extra parameters in the configuration files of "remote CGI gateway" and Remote-CGI servers to connect them via the GRID framework
4. Install and run your Remote-CGI servers on as many machines as you need. They don't require Web server, and can be installed even on PCs and Macs.

Diagram

ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/DOC/PPT/IMAGES/GRID_Dec14_2006/Slide1.PNG

Features and benefits

- Solves 30-sec Web server timeout problem.
- Provides software infrastructure for back-end computation farm for CGIs. Cross-platform, Unix-Windows compatible, minimal administration.
- Existing CGIs can be easily converted into back-end worker nodes.
- While the request is being executed by the Remote-CGI server, the user can be interactively provided with a standard or customized progress report.
- Can be used for parallel network programming.
- High availability infrastructure. All central components can have 2-3 times reservation to accommodate request peak hours and possible hardware failures.
- Remote-CGI servers are extremely mobile.
- Remote-CGI servers can be administered (gentle shutdown, request statistics, etc.) using special tool.
- Easy to debug, as the Remote-CGI server can be run under debugger or any memory checker on any machine (UNIX or MS-Windows)

Wrapping an existing CGI application into a GRID Node

The following sections describe how to wrap an existing CGI application into a GRID Node:

- Running existing CGI executable through Grid Framework
- Diagram

Running existing CGI executable through Grid Framework

In this case a real CGI does not need to be modified at all and remote_cgi utility serves as an intermediate between NetSchedule service and a real CGI. The real CGI and remote_cgi utility go to the server side. The remote_cgi gets a job from NetSchedule service, deserializes the CGI request and stdin stream and runs the real CGI with this data. When the CGI finishes the remote_cgi utility serializes its stdout stream and sends it back to the client.

On the client side (front-end) cgi2rcgi sees that the job's status is changed to “done” gets the data sent by the server side (back-end), deserializes it and writes it on its stdout.

cgi2rcgi utility has two html template files to define its look. The first file is cgi2rcgi.html (can be redefined in cgi2rcgi.ini file) which is the main html template file and it contains all common html tags for the particular application. It also has to have two required tags.

<@REDIRECT@> should be inside <head> tag and is used to inject a page reloading code.

<@VIEW@> should be inside <body> tag and is to render information about a particular job's status.

The second file is cgi2rcgi.inc.html (can be redefined in cgi2.rcgi.ini) which defines tags for particular job's states. The tag for the particular job's state replaces <@VIEW@> tag in the main html template file.

Diagram

ftp://ftp.ncbi.nlm.nih.gov/toolbox/ncbi_tools++/DOC/PPT/IMAGES/GRID_Dec14_2006/Slide1.PNG

Wrapping an existing command-line application into a GRID Node

The following sections describe how to wrap an existing CGI application into a GRID Node:

- Running arbitrary applications through Grid Framework
- Diagram

Running arbitrary applications through Grid Framework

The client side collects a command line, a stdin stream and some other parameters, serialize them and through Grid Framework to the server side. On the server side remote_app utility picks up submitted job, deserializes the command line, the stdin and other parameters, and starts a new process with the application and the input data. Then remote_app waits for the process to finish collecting its stdout, stdin and errcode. After that it serializes collected data and sends it back to the client side. The application for run is set in remote_app.ini configuration file.

Source code: src/app/grid/remote_app/remote_app_wn.cpp

Config file: remote_app.ini

Classes that should be used to prepare an input data a remote application and get its results are CRemoteAppRequest and CRemoteAppResult. See also CGridClient, CGridClientApp.

Client example: src/sample/app/netschedule/remote_app_client_sample.cpp

Config file: src/sample/app/netschedule/remote_app_client_sample.ini

ns_submit_remote_job utility allows submitting a job for a remote application from its command line or a jobs file. See ns_submit_remote_job –help.

Jobs file format:

Each line in the file represents one job (lines starting with '#' are ignored). Each job consists of several parameters. Each parameter has in the form: name="value". The parameter's value must be wrapped in double quotes. All of these parameters are optional. Supported parameters:

- args – command line arguments.
- aff – affinity token.
- tfiles – a list of semicolon-separated file names which will be transferred to the server side.
- jout – a file name where the application's output to stdout will be stored.
- jerr – a file name where the application's output to stderr will be stored.
- runtime – a time in seconds of the remote application's running time. If the application is running longer then this time it is assumed to be failed and its execution is terminated.
- exclusive – instructs the remote_app to not get any other jobs from the NetSchedule service while this job is being executed.

Diagram

ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/DOC/PPT/IMAGES/GRID_Dec14_2006/Slide2.PNG

Worker Node Cleanup Procedure

The following sections describe the procedure for cleaning up Worker Nodes:

- Purpose
- Job Cleanup
- Worker Node Cleanup

Purpose

It is necessary to provide a framework to support worker node and job cleanup. For example, a job may create temporary files that need to be deleted, or a worker node may need to clean up resources shared by multiple jobs.

To receive cleanup events, the worker node must implement interface IWorkerNodeCleanupEventListener. The interface has a single abstract method:

void HandleEvent(EWorkerNodeCleanupEvent cleanup_event)

At the time of the call, cleanup_event will be set to either eRegularCleanup (for normal cleanup) or eOnHardExit (for an emergency shutdown).

There are two types of listeners: those called after each job is done and those called when the worker node is shutting down.

Job Cleanup

Listeners of the first type (per-job cleanup) are installed in the Do() method via a call to CWorkerNodeJobContext::GetCleanupEventSource()->AddListener():

```
class CMyWorkerNodeJob : public IWorkerNodeJob
/* ... */
virtual int Do(CWorkerNodeJobContext& context)
{
 context.GetCleanupEventSource()->AddListener( new
CMyWorkerNodeJobCleanupListener(resources_to_free));
}
```

Worker Node Cleanup

Listeners of the second type (worker node cleanup) are installed in the constructor of the IWorkerNodeJob-derived class via a call to IWorkerNodeInitContext::GetCleanupEventSource()->AddListener():

```
class CMyWorkerNodeJob : public IWorkerNodeJob
/* ... */
CMyWorkerNodeJob(const IWorkerNodeInitContext& context)
{
 context.GetCleanupEventSource()->AddListener( new
CMyWorkerNodeCleanupListener(resources_to_free));
}
```

Note that depending on the current value of the [server]/reuse_job_object configuration parameter, this constructor of CMyWorkerNodeJob can be called multiple times - either once per job or once per worker thread, so additional guarding may be required.

The approach of doing worker node cleanup described above is a newer approach, but there is an older approach which may also be used:

The IGridWorkerNodeApp_Listener interface has two methods, OnGridWorkerStart() and OnGridWorkerStop() which are called during worker node initialization and shutdown respectively. A handler implementing this interface can be installed using the SetListener() method of CGridWorkerApp. The code that calls the OnGridWorkerStop() method will run in the context of the dedicated cleanup thread and also respect the force_close parameter.

The older method does not require the guarding that the new method requires.

Job Submitters

An API is available to submit tasks to Worker Nodes, and to monitor and control the submitted tasks.

Implementing a Network Server

The CServer, IServer_ConnectionFactory, and IServer_ConnectionHandler classes provide a framework for creating multithreaded network servers with one-thread-per-request scheduling. The server creates a pool of connection handlers for maintaining the socket connections, and a pool of threads for handling the socket events. With each socket event, CServer allocates a thread from the thread pool to handle the event, thereby making it possible to serve a large number of concurrent requests efficiently.

The following topics discuss the various aspects of implementing a network server:

Typical Client-Server Interactions

The CServer framework is based on sockets and imposes few constraints on client-server interactions. Servers can support many concurrent connections, and the client and server can follow any protocol, provided that they handle errors. If the protocol includes a server response, then the client and server should alternate between requests and responses on a given connection.

Typical client-server interactions differ in the following categories:

- Protocols
- Request Format
- Response Handling

Protocols

The simplest protocol is probably a consistent pattern of a client request followed by a server response. The Track Manager server uses this protocol.

The NetScheduler server follows a modified request / response protocol. It expects three "messages" - two information lines followed by a command line - then it returns a response.

The Genome Pipeline server protocol adds a client acknowledgment to the interaction. A missing or corrupt acknowledgment triggers a rollback of the transaction.

Your server can follow whatever pattern of requests and responses is appropriate for the service. Note that a given server is not limited to a fixed communication pattern. As long as the client and server follow the same rules, the protocol can be adapted to whatever is appropriate at the moment.

Request Format

At a low level, the server simply receives bytes through a socket, so it must parse the input data into separate requests.

Perhaps the easiest request format to parse simply delimits requests with newlines - this is the request format used by the NetScheduler server.

A more robust way to define the request and response formats is with an ASN.1 specification. NCBI servers that use an ASN.1-defined request format include:

- Ideogram
- OS Gateway
- Track Manager
- Genome Pipeline

Response Handling

Servers may be implemented to respond immediately (i.e. in the same thread execution where the request is read), or to delay their responses until the socket indicates that the client is ready to receive. Responding immediately can make the code simpler, but may not be optimal for resource scheduling.

NCBI Servers that use respond immediately include:

- NetScheduler
- Ideogram

NCBI servers that delay their response include:

- OS Gateway
- Track Manager

- Genome Pipeline

The CServer Framework Classes

The main classes in the CServer framework are:

- CServer
- IServer_ConnectionFactory
- IServer_ConnectionHandler

CServer

The CServer class manages connections, socket event handling for reading and writing, timer and timeout events, and error conditions. CServer creates a connection pool and a thread pool. When a client request arrives, a socket is established and assigned to one of the connection handler objects. For each socket event (e.g. connection opened, data arrival, client ready for data, etc.), a thread is allocated from the pool to serve that particular event and is returned to the pool when the handler finishes. You can use CServer directly, but typically it is subclassed.

If you want to provide a gentle shutdown ability, then create a CServer subclass and override ShutdownRequested(). It should return true when the application-specific logic determines that the server is no longer needed - for example, if a shutdown command has been received; if a timeout has expired with no client communication; if a watchfile has been updated; etc. A typical subclass could also include a RequestShutdown() method that sets a flag that is in turn checked by ShutdownRequested(). This approach makes it easy to trigger a shutdown from a client.

If you want to process data in the main thread on timeout, then create a CServer subclass, override ProcessTimeout(), and use GetParameters() / SetParameters() to replace the accept_timeout parameter with the proper value for your application.

If you don't want to provide a gentle shutdown ability and you don't want to process data in the main thread on timeout, then you can use CServer directly.

Your server application will probably define, configure, start listening, and run a CServer object in its Run() method - something like:

```
CMyServer server;
server.SetParameters(params);
server.AddListener(new CMyConnFactory(&server), params.port);
server.Run();
```

IServer_ConnectionFactory

The connection factory simply creates connection handler objects. It is registered with the server and is called when building the connection pool.

It is possible to create a server application without defining your own connection factory (the CServer framework has a default factory). However you must create a connection factory if you want to pass server-wide parameters to your connection handler objects - for example to implement a gentle shutdown.

The connection factory class can be as simple as:

```
class CMyConnFactory : public IServer_ConnectionFactory
{
public:
 CMyConnFactory(CMyServer * server)
 : m_Server(server) {}
 ~CMyConnFactory(void) {}
 virtual IServer_ConnectionHandler * Create(void)
 {
 return new CMyConnHandler(m_Server);
 }
private:
 CMyServer * m_Server;
};
```

IServer_ConnectionHandler

Classes derived from IServer_ConnectionHandler do the actual work of handling requests. The primary methods are:

- GetEventsToPollFor() - This method identifies the socket events that should be handled by this connection, and can establish a timer.
- OnOpen() - Indicates that a client has opened a connection. The socket can be configured here.
- OnClose() - Indicates that a connection was closed. Note that connections can be closed by either the server or the client (the closer is indicated by a parameter).
- OnRead() - Indicates that a client has sent data. This is where you should parse the data coming from the socket.
- OnWrite() - Indicates that a client is ready to receive data. This is where you should write the response to the socket.
- OnTimeout() - Indicates that a client has been idle for too long. The connection will be closed synchronously after this method is called.
- OnTimer() - Called when the timer established by GetEventsToPollFor() has expired.
- OnOverflow() - Called when there's a problem with the connection - for example, the connection pool cannot accommodate another connection. Note: The connection is destroyed after this call.

The OnOpen(), OnRead(), and OnWrite() methods are pure virtual and must be implemented by your server.

Note: If your client-server protocol is line-oriented, you can use IServer_LineMessageHandler instead of IServer_ConnectionHandler. In this case you would implement the OnMessage() method instead of OnRead().

State, Events, and Flow of Control

Generally, your connection handler class should follow a state model and implement the GetEventsToPollFor() method, which will use the state to select the events that will be handled. This is typically how the connection state is propagated and how socket events result in the flow of control being passed to the events handlers.

Note: You don't need to implement a state model or the GetEventsToPollFor() method if you immediately write any reponses in the same handler that performs the reading. For line-oriented protocols, your connection handler can inherit from IServer_LineMessageHandler instead of from IServer_ConnectionHandler. IServer_LineMessageHandler implements OnRead() and parses the input into lines, calling OnMessage() for each line. In this case you would both read from and write to the client in the OnMessage() method (and implement a dummy OnWrite() method).

For servers that implement a state model and follow a simple request / response protocol, the state variable should be initialized to "reading"; set to "writing" after the request is read in the OnRead() method; and set to "reading" after the response is sent in the OnWrite() method. This results in an orderly alternation between reading and writing. The GetEventsToPollFor() method uses the current connection state (the current state corresponds to the next expected event) to select the appropriate event to respond to. For example:

```
EIO_Event CMyConnHandler::GetEventsToPollFor(const CTime** alarm_time)
{
 return (m_State == eWriting) ? eIO_Write : eIO_Read;
}
```

Your state model should reflect the communication protocol and can be more complex than a simple read / write alternation. It could include acknowledgements, queuing, timed responses, etc. Of course it should include error handling.

GetEventsToPollFor() is guaranteed to not be called at the same time as the event handling functions (OnOpen(), OnRead(), etc.), so you shouldn't guard the variables they use with mutexes.

GetEventsToPollFor() is called from the main thread while the other socket event handling methods are called from various threads allocated from the thread pool.

Socket Closure and Lifetime

Nominally, sockets are owned by (and therefore closed by) the CServer framework. However, there may be cases where your derived class will need to manually close or take ownership of the socket.

- Well-behaved clients will close a connection when they have no more outstanding requests and have completed reading the responses to all requests made on the connection. CServer-based applications are intended to operate in this paradigm. In this case you don't need to close or take ownership of the socket.

  Note: If connections are not closed by the client after reading the response, then you may run out of file descriptors and/or available port numbers. If you have a high connection volume and try to mitigate slow connection closings by closing connections in your code, you run the risk of terminating the connection before the client has read all the data. Well-behaved clients are therefore necessary for optimum server performance.
- CServer will automatically close a connection after an inactivity timeout or if an exception occurs in an event handler. You don't need to manage sockets in these cases.
- If you encounter problems such as a broken protocol or an invalid command then you should close the connection from your code.
- If you need to close a connection from your code, you should do so by calling CServer::CloseConnection() - not by explicitly closing the socket object. The CServer framework generally owns the socket and therefore needs to manage it.

- Note: There is one case when the CServer framework shouldn't own the socket. If you create a CConn_SocketStream on an existing socket, then you must take ownership as shown here:

  SOCK sk = GetSocket().GetSOCK();
  GetSocket().SetOwnership(eNoOwnership);
  GetSocket().Reset(0, eTakeOwnership, eCopyTimeoutsToSOCK);
  AutoPtr<CConn_SocketStream> stream = new CConn_SocketStream(sk);

Diagnostics

To facilitate logfile analysis, the more detailed "new" log posting format is recommended. To enable the new format, call SetOldPostFormat() before calling AppMain():

```
int main(int argc, const char* argv[])
{
 GetDiagContext().SetOldPostFormat(false);
 return CMyServerApp().AppMain(argc, argv);
}
```

Grouping diagnostics into request-specific blocks is very helpful for post-processing. To facilitate this, CDiagContext provides the PrintRequestStart(), PrintRequestStop(), Extra(), and various Print(), methods.

The CDiagContext::SetRequestContext() method enables you to use a CRequestContext object to pass certain request-specific information - such as request ID, client IP, bytes sent, request status, etc. - to the diagnostics context. The request context information can be invaluable when analyzing logs.

CRequestContext objects are merely convenient packages for passing information - they can be preserved across multiple events or re-created as needed. However, as CObject-derived objects, they should be wrapped by CRef to avoid inadvertent deletion by code accepting a CRef parameter.

The following code fragments show examples of API calls for creating request-specific blocks in the logfile. Your code will be slightly different and may make these calls in different event handlers (for example, you might call PrintRequestStart() in OnRead() and PrintRequestStop () in OnWrite()).

```
// Set up the request context:
CRef<CRequestContext> rqst_ctx(new CRequestContext());
rqst_ctx->SetRequestID();
rqst_ctx->SetClientIP(socket.GetPeerAddress(eSAF_IP));

// Access the diagnostics context:
CDiagContext & diag_ctx(GetDiagContext());
diag_ctx.SetRequestContext(rqst_ctx.GetPointer());

// Start the request block in the log:
diag_ctx.PrintRequestStart()
 .Print("peer", "1.2.3.4")
 .Print("port", 5555);
```

```
// Other relevant info...
CDiagContext_Extra extra(diag_ctx.Extra());
extra.Print("name1", "value1")
 .Print("name2", "value2");

// Terminate the request block in the log.
rqst_ctx->SetBytesRd(socket.GetCount(eIO_Read));
rqst_ctx->SetBytesWr(socket.GetCount(eIO_Write));
rqst_ctx->SetRequestStatus(eStatus_OK);
diag_ctx.PrintRequestStop();
```

Code like the above will result in AppLog entries that look similar to:

```
01234/027/224659948 4B56F75BBF3A2D80 2012-08-07 00:07:33 server1 0.0.0.0 UNK_SESSION myserverapp
   request-start  peer=1.2.3.4 & port=5555
          extra  name1=value1 & name2=value2
    request-stop  200 0.292 234 543
```

Each thread has its own diagnostics context. Therefore, simultaneous calls to GetDiagContext().SetRequestContext() in multiple event handlers will not interfere with each other.

The connection handler should ensure that each request-start has a corresponding request-stop - for example by writing the request-stop in a destructor if it wasn't already written.

Handling Exceptions

There are server application-wide configuration parameters to control whether or not otherwise-unhandled exceptions will be caught by the server. See the Server Configuration section for details.

Note: If your event handler catches an exception, it does **not** need to close the connection because CServer automatically closes connections in this case.

See the Socket Closure and Lifetime section for related information.

Server Configuration

The following configuration parameters can be used to fine-tune CServer-derived server behavior:

Parameter	**Brief Description**	**Default**
CSERVER_CATCH_UNHANDLED_EXCEPTIONS	Controls whether CServer should catch exceptions.	true
NCBI_CONFIG__THREADPOOL__CATCH_UNHANDLED_EXCEPTIONS	Controls whether CThreadInPool_ForServer should catch exceptions.	true

See the connection library configuration reference for more information on configuration parameters.

Other Resources

Here are some places to look for reference and to see how to CServer is used in practice:

- CServer Class Reference
- CServer test application

- NetScheduler
- Ideogram (NCBI only)
- OS Gateway (NCBI only)
- Track Manager (NCBI only)
- Genome Pipeline (NCBI only)

GRID Utilities

The following sections describe the GRID Utilities:

- netschedule_control
- ns_remote_job_control
- Alternate list input and output

netschedule_control

DESCRIPTION:

NCBI NetSchedule control utility. This program can be used to operate NetSchedule servers and server groups from the command line.

OPTIONS:

-h	Print brief usage and description; ignore other arguments.
-help	Print long usage and description; ignore other arguments.
-xmlhelp	Print long usage and description in XML format; ignore other arguments.
-version-full	Print extended version data; ignore other arguments.
-service <SERVICE_NAME>	Specifies a NetSchedule service name to use. It can be either an LBSMD service name or a server name / port number pair separated by a colon, such as: host:1234
-queue <QUEUE_NAME>	The queue name to operate with.
-jid <JOB_ID>	This option specifies a job ID for those operations that need it.
-shutdown	This command tells the specified server to shut down. The server address is defined by the -service option. An LBSMD service name cannot be used with -shutdown.
-shutdown_now	The same as -shutdown but does not wait for job termination.
-log <ON_OFF>	Switch server side logging on and off.
-monitor	Starts monitoring of the specified queue. Events associated with that queue will be dumped to the standard output of netschedule_control until it's terminated with Ctrl-C.
-ver	Prints server version(s) of the server or the group of servers specified by the -service option.
-reconf	Send a request to reload server configuration.
-qlist	List available queues.
-qcreate	Create queue (qclass should be present, and comment is an optional parameter).
-qclass <QUEUE_CLASS>	Class for queue creation.
-comment <COMMENT>	Optional parameter for the -qcreate command
-qdelete	Delete the specified queue.

-drop	Unconditionally drop ALL jobs in the specified queue.
-stat <STAT_TYPE>	Print queue statistics. Available values for STAT_TYPE: all, brief.
-affstat <AFFINITY_NAME>	Print queue statistics summary based on affinity.
-dump	Print queue dump or job dump if -jid parameter is specified.
-reschedule <JOB_ID>	Reschedule the job specified by the JOB_ID parameter.
-cancel <JOB_ID>	Cancel the specified job.
-qprint <JOB_STATUS>	Print queue content for the specified job status.
-count <QUERY_STRING>	Count all jobs within the specified queue with tags set by query string.
-count_active	Count active jobs in all queues.
-show_jobs_id <QUERY_STRING>	Show all job IDs by query string.
-query <QUERY_STRING>	Perform a query on the jobs withing the specified queue.
-fields <FIELD_LIST>	Fields (separated by ','), which should be returned by one of the above query commands.
-select <QUERY_STRING>	Perform a select query on the jobs withing the specified queue.
-showparams	Show service parameters.

-read <BATCH_ID_OUTPUT,JOB_IDS_OUTPUT,LIMIT,TIMEOUT>	Retrieve IDs of the completed jobs and change their state to Reading.
	For the first two parameters, the Alternate list output format can be used.
	Parameter descriptions:
	BATCH_ID_OUTPUT
	Defines where to send the ID of the retrieved jobs. Can be either a file name or '-'.
	JOB_IDS
	Defines where to send the list of jobs that were switched to the state Reading. Can be either a file name or '-'.
	LIMIT
	Maximum number of jobs retrieved.
	TIMEOUT
	Timeout before jobs will be switched back to the state Done so that they can be returned again in response to another -read.
	Examples:
	`netschedule_control -service NS_Test -queue test \`
	`-read batch_id.txt,job_ids.lst,100,300`
	`netschedule_control -service NS_Test -queue test \`
	`-read -,job_ids.lst,100,300`
	`netschedule_control -service NS_Test -queue test \`
	`-read batch_id.txt,-,100,300`

-read_confirm <JOB_LIST>	Mark jobs in JOB_LIST as successfully retrieved. The Alternate list input format can be used to specify JOB_LIST. If this operation succeeds, the specified jobs will change their state to Confirmed.
	
	Examples:
	
	`netschedule_control -service NS_Test -queue test \`
	`-read_confirm @job_ids.lst`
	`netschedule_control -service NS_Test -queue test \`
	`-read_confirm - < job_ids.lst`
	`netschedule_control -service NS_Test -queue test \`
	`-read_confirm`
	`JSID_01_4_130.14.24.10_9100,JSID_01_5_130.14.24.10_9100`

-read_rollback <JOB_LIST>	Undo the -read operation for the specified jobs thus making them available for the subsequent -read operations. See the description of -read_confirm for information on the JOB_LIST argument and usage examples.
-read_fail <JOB_LIST>	Undo the -read operation for the specified jobs thus making them available for the subsequent -read operations. This command is similar to -read_rollback with the exception that it also increases the counter of the job result reading failures for the specified jobs. See the description of -read_confirm for information on the JOB_LIST argument and usage examples.
-logfile <LOG_FILE>	File to which the program log should be redirected.
-conffile <INI_FILE>	Override configuration file name (by default, netschedule_control.ini).
-version	Print version number; ignore other arguments.
-dryrun	Do nothing, only test all preconditions.

ns_remote_job_control

DESCRIPTION:

This utility acts as a submitter for the remote_app daemon. It initiates job execution on remote_app, and then checks the status and the results of the job.

OPTIONS:

-h	Print brief usage and description; ignore other arguments.
-help	Print long usage and description; ignore other arguments.
-xmlhelp	Print long usage and description in XML format; ignore other arguments.
-q <QUEUE>	NetSchedule queue name.
-ns <SERVICE>	NetSchedule service address (service_name or host:port).
-nc <SERVICE>	NetCache service address (service_name or host:port).

-jlist <STATUS>	Show jobs by status. STATUS can be one of the following: • all • canceled • done • failed • pending • returned • running
-qlist	Print the list of queues available on the specified NetSchedule server or a group of servers identified by the service name.
-wnlist	Show registered worker nodes.
-jid <JOB_ID>	Show information on the specified job.
-bid <BLOB_ID>	Show NetCache blob contents.
-attr <ATTRIBUTE>	Show one of the following job attributes: • cmdline • progress • raw_input • raw_output • retcode • status • stdin • stdout • stderr Alternatively, the ATTRIBUTE parameter can be specified as one of the following attribute sets: • standard • full • minimal
-stdout <JOB_IDS>	Dump concatenated standard output streams of the specified jobs. The JOB_IDS argument can be specified in the Alternate list input format. Examples: `ns_remote_job_control -ns NS_Test -q test \` `-stdout JSID_01_4_130.14.24.10_9100,JSID_01_5_130.14.24.10_9100` `ns_remote_job_control -ns NS_Test -q test -stdout @job_ids.lst` `ns_remote_job_control -ns NS_Test -q test -stdout - < job_ids.lst`
-stderr <JOB_IDS>	Dump concatenated standard error streams of the specified jobs. The JOB_IDS argument can be specified in the Alternate list input format. See the description of the -stdout command for examples.
-cancel <JOB_ID>	Cancel the specified job.
-cmd <COMMAND>	Apply one of the following commands to the queue specified by the -q option: • drop_jobs • kill_nodes • shutdown_nodes

-render <OUTPUT_FORMAT>	Set the output format of the informational commands like -qlist. The format can be either of the following: text, xml.
-of <OUTPUT_FILE>	Output file for operations that actually produce output.
-logfile <LOG_FILE>	File to which the program log should be redirected.
-conffile <INI_FILE>	Override configuration file name (by default, ns_remote_job_control.ini).
-version	Print version number; ignore other arguments.
-version-full	Print extended version data; ignore other arguments.
-dryrun	Do nothing, only test all preconditions.

Alternate list input and output

This section describes two alternative methods of printing the results of operations that generate lists (e.g. lists of job IDs) and three methods of inputting such lists as command line arguments.

Alternate list output

The -read command of netschedule_control produces a list of job IDs as its output. This list can be sent either to a file (if a file name is specified) or to stdout (if a dash ('-') is specified in place of the file name).

Example:

```
# Read job results: send batch ID to STDOUT,
# and the list of jobs to job_ids.lst
netschedule_control -service NS_Test -queue test \
-read -,job_ids.lst,10,300
```

Alternate list input

There are three ways one can specify a list of arguments in a command line option that accepts the Alternate list input format (like the -stdout and stderr options of ns_remote_job_conrol):

1. Via a comma-separated (or a space-separated) list.
2. By using a text file (one argument per line). The name of the file must be prefixed with '@' to distinguish from the explicit enumeration of the previous case.
3. Via stdin (denoted by '-'). This variant does not differ from using a text file except that list items are red from the standard input - one item per line.

Examples:

```
# Concatenate and print stdout
ns_remote_job_control -ns NS_Test -q rmcgi_sample \
-stdout JSID_01_4_130.14.24.10_9100,JSID_01_5_130.14.24.10_9100

# Confirm job result reading for batch #6
netschedule_control -service NS_Test -queue test \
-read_confirm 6,@job_ids.lst

# The same using STDIN
netschedule_control -service NS_Test -queue test \
-read_confirm 6,- < job_ids.lst
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

24: Applications

Created: April 1, 2003.

Last Update: May 8, 2013.

Overview

- Introduction
- Chapter Outline

Introduction

Most of the applications discussed in this chapter are built on a regular basis, at least once a day from the latest sources, and if you are in NCBI, then you can find the latest version in the directory: $NCBI/c++/Release/bin/ (or $NCBI/c++/Debug/bin/).

Chapter Outline

The following is an outline of the topics presented in this chapter:

DATATOOL: Code Generation and Data Serialization Utility

DATATOOL source code is located at c++/src/serial/datatool; this application can perform the following:

- Generate C++ data storage classes based on ASN.1, DTD or XML Schema specification to be used with NCBI data serialization streams.
- Convert ASN.1 specification into a DTD or XML Schema specification and vice versa.
- Convert data between ASN.1, XML and JSON formats.

Note: Because ASN.1, XML and JSON are, in general, incompatible, the last two functions are supported only partially.

The following additional topics are discussed in subsections:

- Invocation
- Data specification conversion

- Definition file
- Module file
- Generated code
- Class diagrams

Invocation

The following topics are discussed in this section:

- Main arguments
- Code generation arguments

Main Arguments

See Table 1.

Code Generation Arguments

See Table 2.

Data Specification Conversion

When parsing data specification, DATATOOL identifies specification format by source file extension - ASN, DTD or XSD.

Scope Prefixes

Initially, DATATOOL and serial library supported serialization in ASN.1 and XML format, and conversion of ASN.1 specification into DTD. Comparing with ASN, DTD is a very sketchy specification in a sense that there is only one primitive type - string, and all elements are defined globally. The latter feature of DTD led to a decision to use 'scope prefixes' in XML output to avoid potential name conflicts. For example, consider the following ASN.1 specification:

```
Date ::= CHOICE {
 str VisibleString,
 std Date-std
}
Time ::= CHOICE {
 str VisibleString,
 std Time-std
}
```

Here, accidentally, element str is defined identically both in Date and Time productions; while the meaning of element std depends on the context. To avoid ambiguity, this specification translates into the following DTD:

```
<!ELEMENT Date (Date_str | Date_std)>
<!ELEMENT Date_str (#PCDATA)>
<!ELEMENT Date_std (Date-std)>
<!ELEMENT Time (Time_str | Time_std)>
<!ELEMENT Time_str (#PCDATA)>
<!ELEMENT Time_std (Time-std)>
```

Accordingly, these scope prefixes made their way into XML output.

Later, DTD parsing was added into DATATOOL. Here, scope prefixes were not needed. Also, since these prefixes considerably increase the size of the XML output, they could be omitted when it is known in advance that there can be no ambiguity. So, DATATOOL has got command line flags, which would enable that.

With the addition of XML Schema parser and generator, when converting ASN.1 specification, elements can be declared in Schema locally if needed, and scope prefixes make almost no sense. Still, they are preserved for compatibility.

Modular DTD and Schemata

Here, ‘module’ means ASN.1 module. Single ASN.1 specification file may contain several modules. When converting it into DTD or XML schema, it might be convenient to put each module definitions into a separate file. To do so, one should specify a special file name in -fx or -fxs command line parameter. The names of output DTD or Schema files will then be chosen automatically - they will be named after ASN modules defined in the source. ‘Modular’ output does not make much sense when the source specification is DTD or Schema.

You can find a number of DTDs and Schema converted by DATATOOL from NCBI public ASN.1 specifications here.

Converting XML Schema into ASN.1

There are two major problems in converting XML schema into ASN.1 specification: how to define XML attributes and how to convert complex content models. The solution was greatly affected by the underlying implementation of data storage classes (classes which DATATOOL generates based on a specification). So, for example the following Schema

```
<xs:element name="Author">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="LastName" type="xs:string"/>
 <xs:choice minOccurs="0">
 <xs:element name="ForeName" type="xs:string"/>
 <xs:sequence>
 <xs:element name="FirstName" type="xs:string"/>
 <xs:element name="MiddleName" type="xs:string" minOccurs="0"/>
 </xs:sequence>
 </xs:choice>
 <xs:element name="Initials" type="xs:string" minOccurs="0"/>
 <xs:element name="Suffix" type="xs:string" minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="gender" use="optional">
 <xs:simpleType>
 <xs:restriction base="xs:string">
 <xs:enumeration value="male"/>
 <xs:enumeration value="female"/>
 </xs:restriction>
 </xs:simpleType>
 </xs:attribute>
 </xs:complexType>
</xs:element>
```

translates into this ASN.1:

```
Author ::= SEQUENCE {
 attlist SET {
 gender ENUMERATED {
 male (1),
 female (2)
 } OPTIONAL
 },
 lastName VisibleString,
 fF CHOICE {
 foreName VisibleString,
 fM SEQUENCE {
 firstName VisibleString,
 middleName VisibleString OPTIONAL
 }
 } OPTIONAL,
 initials VisibleString OPTIONAL,
 suffix VisibleString OPTIONAL
}
```

Each unnamed local element gets a name. When generating C++ data storage classes from Schema, DATATOOL marks such data types as anonymous.

It is possible to convert source Schema into ASN.1, and then use DATATOOL to generate C ++ classes from the latter. In this case DATATOOL and serial library provide compatibility of ASN.1 output. If you generate data storage classes from Schema, and use them to write data in ASN.1 format (binary or text), if you then convert that Schema into ASN.1, generate classes from it, and again write same data in ASN.1 format using this new set of classes, then these two files will be identical.

Definition File

It is possible to tune up the C++ code generation by using a definition file, which could be specified in the -od argument. The definition file uses the generic NCBI configuration format also used in the configuration (*.ini) files found in NCBI's applications.

DATATOOL looks for code generation parameters in several sections of the file in the following order:

- [ModuleName.TypeName]
- [TypeName]
- [ModuleName]
- [-]

Parameter definitions follow a "name = value" format. The "name" part of the definition serves two functions: (1) selecting the specific element to which the definition applies, and (2) selecting the code generation parameter (such as **_class**) that will be fine-tuned for that element.

To modify a top-level element, use a definition line where the name part is simply the desired code generation parameter (such as **_class**). To modify a nested element, use a definition where the code generation parameter is prefixed by a dot-separated "path" of the successive container element names from the data format specification. For path elements of type SET OF or SEQUENCE OF, use an "E" as the element name (which would otherwise be anonymous). Note: Element names will depend on whether you are using ASN.1, DTD, or Schema.

For example, consider the following ASN.1 specification:

```
MyType ::= SEQUENCE {
 label VisibleString ,
 points SEQUENCE OF
 SEQUENCE {
 x INTEGER ,
 y INTEGER
 }
}
```

Code generation for the various elements can be fine-tuned as illustrated by the following sample definition file:

```
[MyModule.MyType]
; modify the top-level element (MyType)
_class = MyTypeX

; modify a contained element
label._class = Title

; modify a "SEQUENCE OF" container type
points._type = vector

; modify members of an anonymous SEQUENCE contained in a "SEQUENCE OF"
points.E.x._type = double
points.E.y._type = double
```

The following additional topics are discussed in this section:

- Common definitions
- Definitions that affect specific types
- The Special [-] Section
- Examples

Common Definitions

Some definitions refer to the generated class as a whole.

_file Defines the base filename for the generated or referenced C++ class.

For example, the following definitions:

```
[ModuleName.TypeName]_file=AnotherName
```

Or

```
[TypeName]
_file=AnotherName
```

would put the class CTypeName in files with the base name AnotherName, whereas these two:

```
[ModuleName]
_file=AnotherName
```

Or

```
[-]
_file=AnotherName
```

put **all** the generated classes into a single file with the base name AnotherName.

_extra_headers Specify additional header files to include.

For example, the following definition:

```
[-]
_extra_headers=name1 name2 \"name3\"
```

would put the following lines into all generated headers:

```
#include <name1>
#include <name2>
#include "name3"
```

Note the name3 clause. Putting name3 in quotes instructs DATATOOL to use the quoted syntax in generated files. Also, the quotes must be escaped with backslashes.

_dir Subdirectory in which the generated C++ files will be stored (in case _file not specified) or a subdirectory in which the referenced class from an external module could be found. The subdirectory is added to include directives.

_class The name of the generated class (if _class=- is specified, then no code is generated for this type).

For example, the following definitions:

```
[ModuleName.TypeName]
_class=AnotherName
```

Or

```
[TypeName]
_class=AnotherName
```

would cause the class generated for the type TypeName to be named CAnotherName, whereas these two:

```
[ModuleName]
_class=AnotherName
```

Or

```
[-]
_class=AnotherName
```

would result in **all** the generated classes having the same name CAnotherName (which is probably not what you want).

_namespace The namespace in which the generated class (or classes) will be placed.

_parent_class The name of the base class from which the generated C++ class is derived.

_parent_type Derive the generated C++ class from the class, which corresponds to the specified type (in case _parent_class is not specified).

It is also possible to specify a storage-class modifier, which is required on Microsoft Windows to export/import generated classes from/to a DLL. This setting affects all generated classes in a module. An appropriate section of the definition file should look like this:

```
[-]
_export = EXPORT_SPECIFIER
```

Because this modifier could also be specified in the command line, the DATATOOL code generator uses the following rules to choose the proper one:

- If no -oex flag is given in the command line, no modifier is added at all.
- If -oex "" (that is, an empty modifier) is specified in the command line, then the modifier from the definition file will be used.
- The command-line parameter in the form -oex FOOBAR will cause the generated classes to have a FOOBAR storage-class modifier, unless another one is specified in the definition file. The modifier from the definition file always takes precedence.

Definitions That Affect Specific Types

The following additional topics are discussed in this section:

- INTEGER, REAL, BOOLEAN, NULL
- ENUMERATED
- OCTET STRING
- SEQUENCE OF, SET OF
- SEQUENCE, SET
- CHOICE

INTEGER, REAL, BOOLEAN, NULL

_type C++ type: int, short, unsigned, long, etc.

ENUMERATED

_type C++ type: int, short, unsigned, long, etc.

_prefix Prefix for names of enum values. The default is "e".

OCTET STRING

_char Vector element type: char, unsigned char, or signed char.

SEQUENCE OF, SET OF

_type STL container type: list, vector, set, or multiset.

SEQUENCE, SET

memberName._delay Mark the specified member for delayed reading.

CHOICE

_virtual_choice If not empty, do not generate a special class for choice. Rather make the choice class as the parent one of all its variants.

variantName._delay Mark the specified variant for delayed reading.

The Special [-] Section

There is a special section [-] allowed in the definition file which can contain definitions related to code generation. This is a good place to define a namespace or identify additional headers. It is a "top level" section, so entries placed here will override entries with the same name in other sections or on the command-line. For example, the following entries set the proper parameters for placing header files alongside source files:

```
[-]
; Do not use a namespace at all:
-on = -

; Use the current directory for generated .cpp files:
-opc = .

; Use the current directory for generated .hpp files:
-oph = .

; Do not add a prefix to generated file names:
-or = -

; Generate #include directives with quotes rather than angle brackets:
-orq = 1
```

Any of the code generation arguments in Table 2 (except -od, -odi, and -odw which are related to specifying the definition file) can be placed in the [-] section.

In some cases, the special value "-" causes special processing as noted in Table 2.

Examples

If we have the following ASN.1 specification (this not a "real" specification - it is only for illustration):

```
Date ::= CHOICE {
 str VisibleString,
 std Date-std
}
Date-std ::= SEQUENCE {
 year INTEGER,
 month INTEGER OPTIONAL
}
Dates ::= SEQUENCE OF Date
Int-fuzz ::= CHOICE {
 p-m INTEGER,
 range SEQUENCE {
 max INTEGER,
 min INTEGER
 },
 pct INTEGER,
 lim ENUMERATED {
 unk (0),
 gt (1),
 lt (2),
 tr (3),
 tl (4),
 circle (5),
 other (255)
 },
 alt SET OF INTEGER
}
```

Then the following definitions will effect the generation of objects:

Definition	**Effected Objects**
[Date]str._type = string	the str member of the Date structure
[Dates]E._pointer = true	elements of the Dates container
[Int-fuzz]range.min._type = long	the min member of the range member of the Int-fuzz structure
[Int-fuzz]alt.E._type = long	elements of the alt member of the Int-fuzz structure

Module File

Module files are not used directly by DATATOOL, but they are input for new_module.sh and project_tree_builder and therefore determine what DATATOOL's command line will be during the build process.

Module files simply consist of lines of the form "KEY = VALUE". Only the key MODULE_IMPORT is currently used (and is the only key ever recognized by project_tree_builder). Other keys used to be recognized by module.sh and still harmlessly remain in some files. The possible keys are:

MODULE_IMPORT These definitions contain a space-delimited list of other modules to import. The paths should be relative to .../src and should not include extensions.

For example, a valid entry could be:

```
MODULE_IMPORT = objects/general/general objects/seq/seq
```

MODULE_ASN, MODULE_DTD, MODULE_XSD These definitions explicitly set the specification filename (normally foo.asn, foo.dtd, or foo.xsd for foo.module). Almost no module files contain this definition. It is no longer used by the project_tree_builder and is therefore not necessary

MODULE_PATH Specifies the directory containing the current module, again relative to .../src. Almost all module files contain this definition, however it is no longer used by either new_module.sh or the project_tree_builder and is therefore not necessary.

Generated Code

The following additional topics are discussed in this section:

- Normalized name
- ENUMERATED types

Normalized Name

By default, DATATOOL generates "normalized" C++ class names from ASN.1 type names using two rules:

1. Convert any hyphens ("-") into underscores ("_"), because hyphens are not legal characters in C++ class names.
2. Prepend a 'C' character.

For example, the default normalized C++ class name for the ASN.1 type name "Seq-data" is "CSeq_data".

The default C++ class name can be overridden by explicitly specifying in the definition file a name for a given ASN.1 type name. For example:

```
[MyModule.Seq-data]
_class=CMySeqData
```

ENUMERATED Types

By default, for every ENUMERATED ASN.1 type, DATATOOL will produce a C++ enum type with the name ENormalizedName.

Class Diagrams

The following topics are discussed in this section:

- Specification analysis
- Data types
- Data values
- Code generation

Specification Analysis

The following topics are discussed in this section:

- ASN.1 specification analysis
- DTD specification analysis

ASN.1 Specification Analysis

See Figure 1.

DTD Specification Analysis

See Figure 2.

Data Types

See CDataType.

Data Values

See Figure 3.

Code Generation

See Figure 4.

Load Balancing

Note: For security reasons not all links in the public version of this document are accessible by the outside NCBI users.

The section covers the following topics:

- The purpose of load balancing
- All the separate components' purpose, internal details, configuration
- Communications between the components
- Monitoring facilities

Overview

The purpose of load balancing is distributing the load among the service providers available on the NCBI network basing on certain rules. The load is generated by both locally-connected and Internet-connected users. The figures below show the most typical usage scenarios.

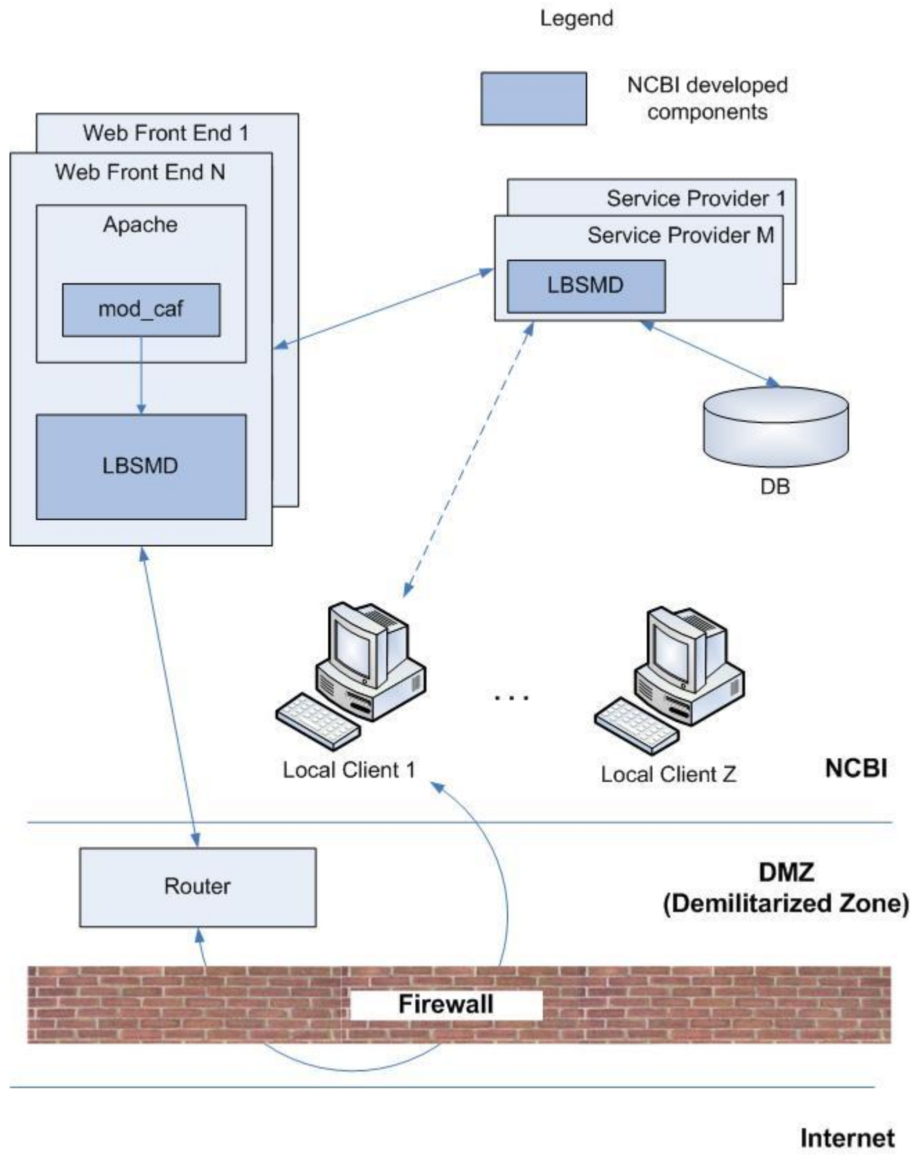


Figure 5. Local Clients

Please note that the figure is simplified slightly to remove unnecessary details for the time being.

In case of local access to the NCBI resources there are two NCBI developed components which are involved into the interactions. These are LBSMD daemon (Load Balancing Service Mapping Daemon) and mod_caf (Cookie/Argument Affinity module) - an Apache web server module.

The LBSMD daemon is running on each host in the NCBI network. The daemon reads its configuration file with all the services available on the host described. Then the LBSMD daemon broadcasts the available services and the current host load to the adjacent LBSMD daemons on a regular basis. The data received from the other LBSMD daemons are stored in a special table. So at some stage the LBSMD daemon on each host has a full description of the services available on the network as well as the current hosts' load.

The mod_caf Apache's module analyses special cookies, query line arguments and reads data from the table populated by the LBSMD daemon. Basing on the best match it makes a decision of where to pass a request further.

Suppose for a moment that a local NCBI client runs a web browser, points to an NCBI web page and initiates a DB request via the web interface. At this stage the mod_caf analyses the request line and makes a decision where to pass the request. The request is passed to the ServiceProviderN host which performs the corresponding database query. Then the query results are delivered to the client. The data exchange path is shown on the figure above using solid lines.

Another typical scenario for the local NCBI clients is when client code is run on a user workstation. That client code might require a long term connection to a certain service, to a database for example. The browser is not able to provide this kind of connection so a direct connection is used in this case. The data exchange path is shown on the figure above using dashed lines.

The communication scenarios become more complicated in case when clients are located outside of the NCBI network. The figure below describes the interactions between modules when the user requested a service which does not suppose a long term connection.

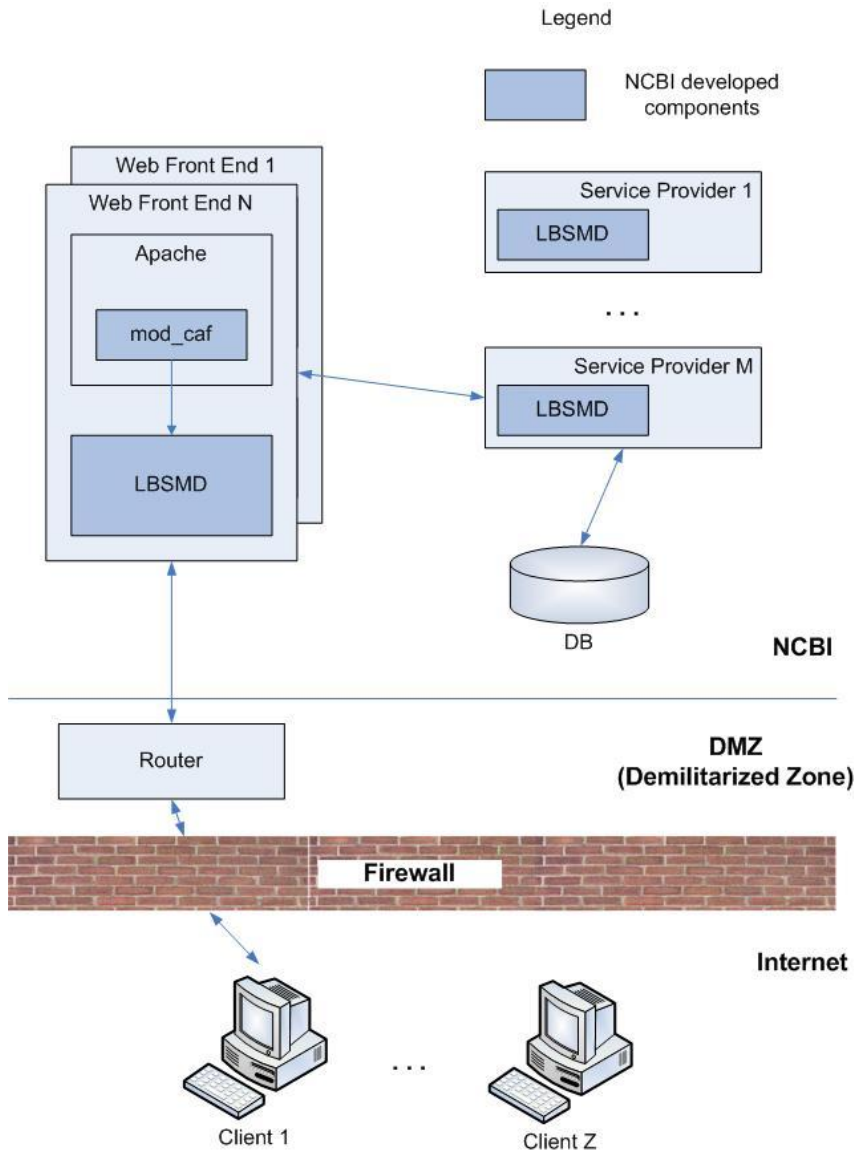


Figure 6. Internet Clients. Short Term Connection

The clients have no abilities to connect to front end Apache web servers directly. The connection is done via a router which is located in DMZ (Demilitarized Zone). The router selects one of the available front end servers and passes the request to that web server. Then the web server processes the request very similar to how it processes requests from a local client.

The next figure explains the interactions for the case when an Internet client requests a service which supposes a long term connection.

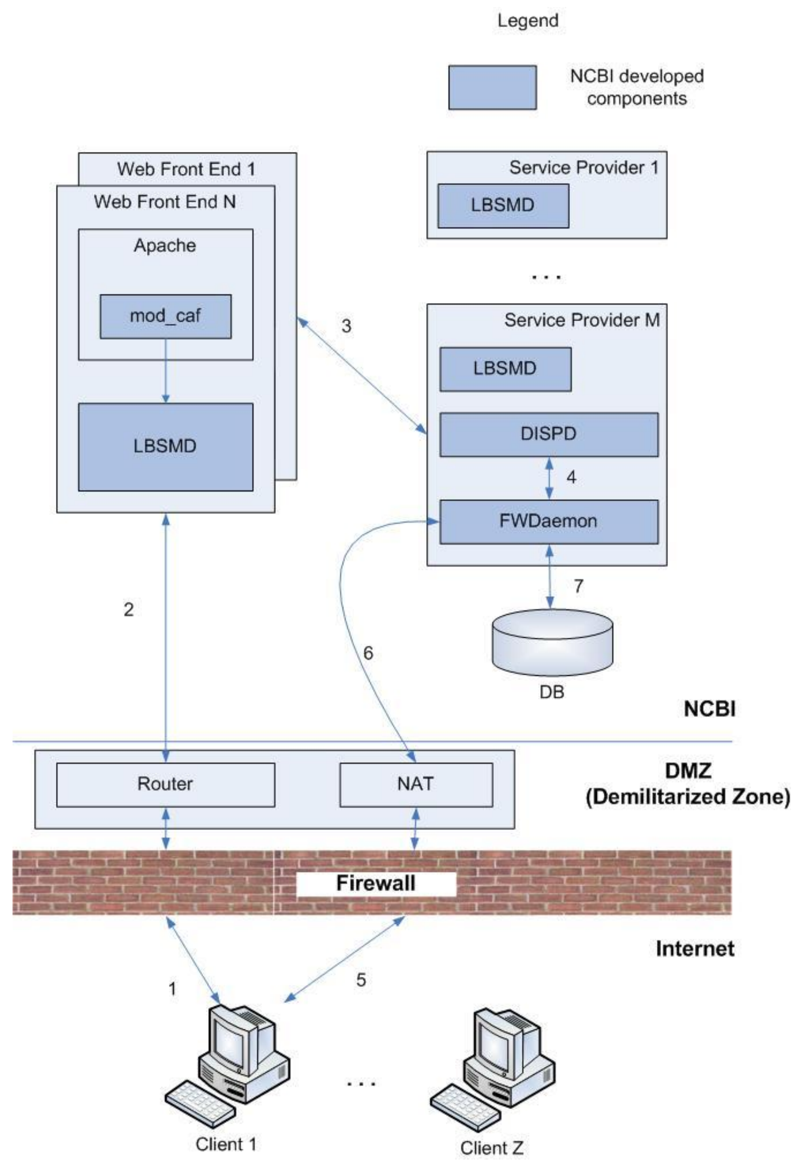


Figure 7. Internet Clients. Long Term Connection

In opposite to the local clients the internet clients are unable to connect to the required service directly because of the DMZ zone. This is where DISPD, FWDaemon and a proxy come to help resolving the problem.

The data flow in the scenario is as follows. A request from the client reaches a front end Apache server as it was discussed above. Then the front end server passes the request to the DISPD dispatcher. The DISPD dispatcher communicates to FWDaemon (Firewall Daemon) to provide the required service facilities. The FWDaemon answers with a special ticket for the requested service. The ticket is sent to the client via the front end web server and the router. Then the client connects to the NAT service in the DMZ zone providing the received ticket. The NAT service establishes a connection to the FWDaemon and passes the received earlier ticket. The FWDaemon, in turn, provides the connection to the required service. It is worth to mention that the FWDaemon is running on the same host as the DISPD dispatcher and neither DISPD nor FWDaemon can work without each other.

The most complicated scenario comes to the picture when an arbitrary UNIX filter program is used as a service provided for the outside NCBI users. The figure below shows all the components involved into the scenario.

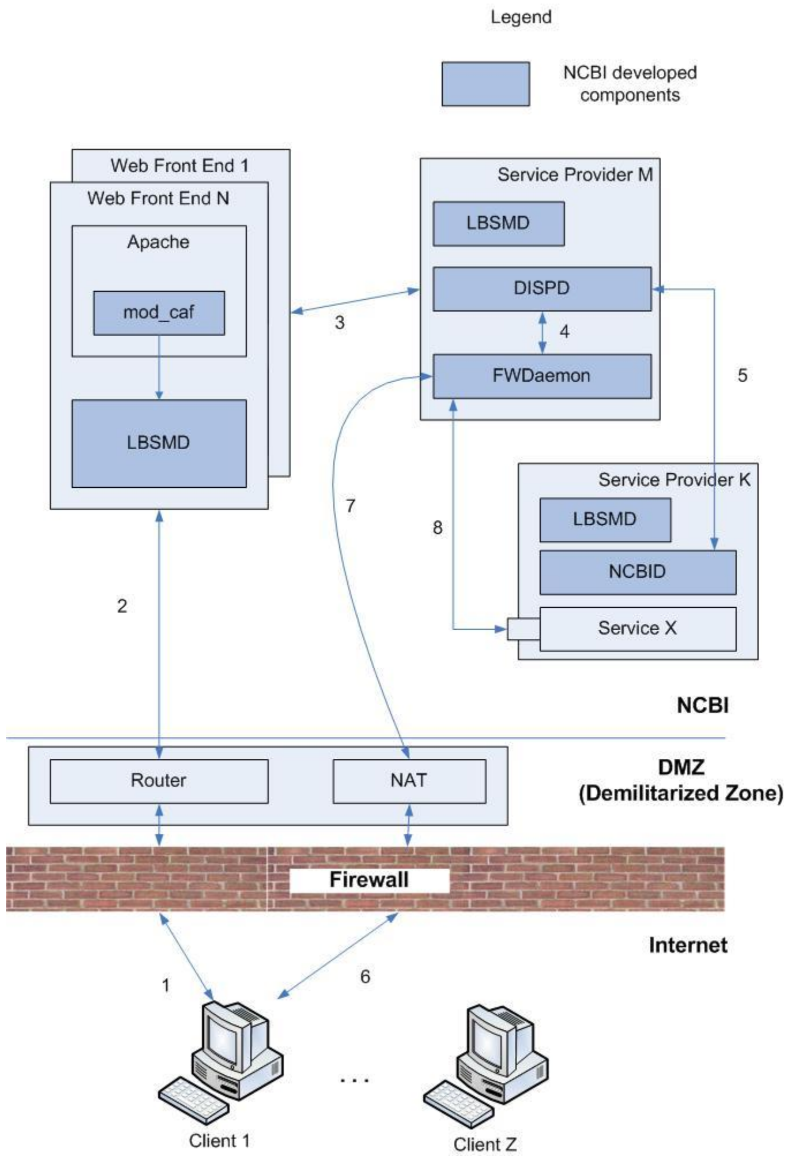


Figure 8. NCBID at Work

The data flow in the scenario is as follows. A request from the client reaches a front end Apache server as it was discussed above. Then the front end server passes the request to the DISPD dispatcher. The DISPD communicates to both the FWDaemon and the NCBID utility on (possibly) the other host and requests to demonize a requested UNIX filter program (Service X on the figure). The demonized service starts listening on the certain port for a network connection. The connection attributes are delivered to the FWDaemon and to the client via the web front end and the router. The client connects to the NAT service and the NAT service passes the request further to the FWDaemon. The FWDaemon passes the request to the demonized Service X on the Service Provider K host. Since that moment the client is able to start data exchange with the service. The described scenario is purposed for long term connections oriented tasks.

Further sections describe all the components in more detail.

Load Balancing Service Mapping Daemon (LBSMD)

Overview

As it was mentioned earlier the purpose of LBSMD daemon is running on each host which carries either public or private servers which, in turn, implement NCBI services. The services include CGI programs or standalone servers to access NCBI data.

Each service has a unique name assigned to it. The “TaxServer” would be an example of such name. The name not only identifies a service. It also implies a protocol which is used for data exchange with the certain service. For example, any client which connects to the “TaxServer” service knows how to communicate with that service regardless the way the service is implemented. In other words the service could be implemented as a standalone server on host X and as a CGI program on the same host or on another host Y (please note, however, that there are exceptions and for some service types it is forbidden to have more than one service type on the same host).

A host can advertize many services. For example, one service (such as “Entrez2”) can operate with binary data only while another one (such as “Entrez2Text”) can operate with text data only. The distinction between those two services could be made by using a content type specifier in the LBSMD daemon configuration file.

The main purpose of the LBSMD daemon is to maintain a table of all services available at NCBI at the moment. In addition the LBSMD daemon keeps track of servers that are found to be nonfunctional (dead servers). The daemon is also responsible for propagating trouble reports, obtained from applications. The application trouble reports are based on their experience with advertised servers (e.g., an advertised server is not technically dead but generates some sort of garbage). Further in this document, the latter kind of feedback is called a penalty.

The principle of load balancing is simple: each server which implements a service is assigned a (calculated) rate. The higher the rate, the better the chance for that server to be chosen when a request for a service comes up. Note that load balancing is thus almost never deterministic.

The LBSMD daemon calculates two parameters for the host on which it is running. The parameters are a normal host status and a BLAST host status (based on the instant load of the system). These parameters are then used to calculate the rate of all (non static) servers on the host. The rates of all other hosts are not calculated but received and stored in the LBSDM table.

The LBSMD daemon is started from crontab every few minutes on all the production hosts to ensure that the daemon is always running. This technique is safe because no more than one instance of the daemon is permitted on a certain host and any attempt to start more than one is rejected.

The main loop of the LBSMD daemon comprises periodic checking of the configuration file and reloading the configuration if necessary, checking and processing the incoming messages from neighbor LBSMD daemons running on other hosts, and generation and broadcasting the messages to the other hosts about the load of the system and configured services. The LBSMD daemon also checks periodically whether the configured servers are alive by trying to connect to them and then disconnect immediately, without sending/receiving any data. This is the only way how the daemon is able to check whether the network port is working.

Clients can redirect services. The LBSMD does not distinguish between direct and redirected services.

Configuration

The LBSMD daemon is configured via command line options and via a configuration file. The full list of command line options can be retrieved by issuing the following command:

/opt/machine/lbsm/sbin/lbsmd --help

The local NCBI users can also visit the following link:

http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmd.cgi

The default name of the LBSMD daemon configuration file is /etc/lbsmd/servrc.cfg. Each line can be one of the following:

- a part of the host environment
- an include directive
- a service definition
- an empty line (entirely blank or containing a comment only)

Empty lines are ignored in the file. Any single configuration line can be split into several physical lines by inserting backslash symbols (\) before the line breaks. A comment is introduced by the pound symbol (#).

A configuration line of the form

```
name=value
```

goes into the host environment. The host environment can be accessed by clients when they perform the service name resolution. The host environment is designed to help the client to know about limitations/options that the host has, and based on this additional information the client can make a decision whether the server (despite the fact that it implements the service) is suitable for carrying out the client's request. For example, the host environment can give the client an idea about what databases are available on the host. The host environment is not interpreted or used in any way by either the daemon or by the load balancing algorithm, except that the name must be a valid identifier. The value may be practically anything, even empty. It is left solely to the client to parse the environment and to look for the information of interest. The host environment can be obtained from the service iterator by a call to SERV_GetNextInfoEx() (http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/ident?i=SERV_GetNextInfoEx), which is documented in the service mapping API

Note: White space characters which surround the name are not preserved but they are preserved in the value i.e. when they appear after the “=” sign.

A configuration line of the form

```
%include filename
```

causes the filename file content be inserted here. The daemon always assumes that relative file names (those with names that do not start with the slash character (/)) are given with the daemon startup directory as a base. This is true for any level of nesting.

Once started, the daemon first assigns the configuration file name as /etc/lbsmd/servrc.cfg and then tries to read it. If the file is not found (or is not readable) the daemon looks for the configuration file servrc.cfg in the directory from which the server has been started. If the file is found then the file is used as a configuration file. This fallback mechanism is not used when the configuration file name is explicitly stated in the command line. The daemon periodically checks the configuration file and all of its descendants and reloads (discards) their contents if some of the files have been either updated, (re-)moved, or added.

A configuration line of the form

```
service_name [check_specifier] server_descriptor [| launcher_info ]
```

introduces a service. The detailed description of the individual fields is given below.

- service_name introduces the service name, for example TaxServer.
- [check_specifier] is an optional parameter (if omitted, the surrounding square brackets must not be used). The parameter is a comma separated list and each element in the list can be one of the following.
  - [-]N[/[-]M] where N and M are integers. This will lead to checking every N seconds with backoff time of M seconds if failed. The “-“ character is used when it is required to check dependencies only but not the primary connection point. "0", which stands for "no check interval", disables checks for the service.
  - [!][host[:port]][+[service]] which describes a dependency. The “!” character means negation. The service is a service name the describing service depends on and runs on host:port. The pair host:port is required if no service is specified. The host, :port, or both can be missing if service is specified (in that case the missing parts are read as “any”). The “+” character alone means “this service” (the one currently being defined). There could be multiple dependency specifications for a service.
  - [~][DOW[-DOW]][@H[-H]] which defines a schedule. The “~” character means negation. The service runs from DOW to DOW (DOW is one of Su, Mo, Tu, We, Th, Fr, Sa) or any if not specified and between hours H to H (9-5 means 9:00am thru 5:59pm, 9-22 means 9:00am thru 10:59pm). Single DOW and / or H are allowed and mean the exact day of week and / or the exact hour. There could be multiple schedule specifications.
  - email@ncbi.nlm.nih.gov which makes the LBSMD daemon to send an e-mail to the specified address whenever this server changes its status (e.g. from up to down). There could be many e-mail specifications. The ncbi.nlm.nih.gov part is fixed and is not allowed to be changed.
  - user which makes the LBSMD daemon add the specified user to the list of users who are authorized to change the server rate on the fly (e.g. post a penalty, issue re-rate command etc.). By default these actions are allowed to the root and lbsmd users. There could be many user specifications.
  - script which specifies a path to a local executable which checks whether the server is operational. The LBSMD daemon starts this script periodically as

specified by the check time parameter(s) above. A single script specification is allowed. See Check Script Specification for more details.

- server_descriptor specifies the address of the server and supplies additional information. An example of the server_descriptor:
  STANDALONE somehost:1234 R=3000 L=yes S=yes B=-20
  See Server Descriptor Specification for more details.
- launcher_info is basically a command line preceded by a pipe symbol (|) which plays a role of a delimiter from the server_descriptor. It is only required for the NCBID type of service which are configured on the local host.

Check Script Specification

The check script file is configured between square brackets '[' and ']' in the service definition line. For example, in this service definition line:

MYSERVICE [5, /bin/user/directory/script.sh] STANDALONE :2222 ...

the period in seconds between script checks is "5" and the check script file is "/bin/user/directory/script.sh". The default period is 15 seconds. You can prepend "-" to the period to indicate that LBSMD should not check the connection point (:2222 in this example) on its own, but should only run the script. The script must finish before the next check run is due. Otherwise, LBSMD will remove the script from the check schedule (and won't use it again).

The following command-line parameters are always passed to the script upon execution:

- argv[0] = name of the executable with preceding '|' character if stdin / stdout are open to the server connection (/dev/null otherwise), NB: '|' is not always readily accessible from within shell scripts, so it's duplicated in argv[2] for convenience;
- argv[1] = name of the service being checked;
- argv[2] = if piped, "|host:port" of the connection point being checked, otherwise "host:port" of the server as per configuration;

The following additional command-line parameters will be passed to the script if it has been run before:

- argv[3] = exit code obtained in the last check script run;
- argv[4] = repetition count for argv[3] (NB: 0 means this is the first occurrence of the exit code given in argv[3]);
- argv[5] = seconds elapsed since the last check script run.

Output to stderr is attached to the LBSMD log file; the CPU limit is set to maximal allowed execution time.

The check script is expected to finish with the following exit codes:

- within the range [0..100], where 0 means the server is running at full throttle (fully available), and 100 means that the server has to be considered down; or
- 123 to request to keep the last exit code if that has been supplied in argv[3] (which is guaranteed to be within [0..100]); or
- 127 to request to turn the server off from LBSMD configuration.

Note that a code from the range [0..100] resets the repetition count even though the resulting exit code may be equal to the previous one. Any other exit code (or code 123 when no previous code is available) will cause the server to be considered fully up (as if 0 has been returned), and will be logged with a warning. Note that upon code 127 no further script runs will occur.

If the check script crashes ungracefully (with or without the coredump), it will be eliminated from further checks, and the server will be considered fully available (i.e. as if 0 had been returned).

Note: The check script operation is complementary to setting a penalty prior to doing any disruptive changes in production. In other words, the script is only reliable as long as the service is expected to work. If there is any scheduled maintenance, it should be communicated to LBSMD via a penalty rather than by an assumption that the failing script will do the job of bringing the service to the down state and excluding it from LB.

Server Descriptor Specification

The server_descriptor, also detailed in connect/ncbi_server_info.h (http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/include/connect/ncbi_server_info.h), consists of the following fields:

server_type [host][:port] [arguments] [flags]

where:

- server_type is one of the following keywords (more info):
  - NCBID for servers launched by ncbid.cgi
  - STANDALONE for standalone servers listening to incoming connections on dedicated ports
  - HTTP_GET for servers, which are the CGI programs accepting only the GET request method
  - HTTP_POST for servers, which are the CGI programs accepting only the POST request method
  - HTTP for servers, which are the CGI programs accepting either GET or POST request methods
  - DNS for introduction of a name (fake service), which can be used later in load-balancing for domain name resolution
  - NAMEHOLD for declaration of service names that cannot be defined in any other configuration files except for the current configuration file. Note: The FIREWALL server specification may not be used in a configuration file (i.e., may neither be declared as services nor as service name holders).
- both host and port parameters are optional. Defaults are local host and port 80, except for STANDALONE and DNS servers, which do not have a default port value. If host is specified (by either of the following: keyword localhost, localhost IP address 127.0.0.1, real host name, or IP address) then the described server is not a subject for variable load balancing but is a static server. Such server always has a constant rate, independent of any host load.
- arguments are required for HTTP* servers and must specify the local part of the URL of the CGI program and, optionally, parameters such as /somepath/somecgi.cgi?param1&param2=value2&param3=value3. If no parameters are to be supplied, then the question mark (?) must be omitted, too. For NCBID servers, arguments are parameters to pass to the server and are formed as arguments for CGI programs, i.e., param1&param2&param3=value. As a special rule, '' (two single quotes) may be used to denote an empty argument for the NCBID server. STANDALONE and DNS servers do not take any arguments.

- flags can come in any order (but no more than one instance of a flag is allowed) and essentially are the optional modifiers of values used by default. The following flags are recognized (see ncbi_server_info.h):
  - load calculation keyword:
    - Blast to use special algorithm for rate calculation acceptable for BLAST (http://www.ncbi.nlm.nih.gov/blast/Blast.cgi) applications. The algorithm uses instant values of the host load and thus is less conservative and more reactive than the ordinary one.
    - Regular to use an ordinary rate calculation (default, and the only load calculation option allowed for static servers).
  - base rate:
    - R=value sets the base server reachability rate (as a floating point number); the default is 1000. Any negative value makes the server unreachable, and a value 0 is used. The range of the base rate is between 0.001 and 100000.
  - locality markers (Note: If necessary, both L and P markers can be combined in a particular service definition):
    - L={yes|no} sets (if yes) the server to be local only. The default is no. The service mapping API returns local only servers in the case of mapping with the use of LBSMD running on the same - local - host (direct mapping), or if the dispatching (indirect mapping) occurs within the NCBI Intranet. Otherwise, if the service mapping occurs using a non-local network (certainly indirectly, by exchange with dispd.cgi) then servers that are local only are not seen.
    - P={yes|no} sets (if yes) the server to be private. The default is no. Private servers are not seen by the outside NCBI users (exactly like local servers), but in addition these servers are not seen from the NCBI Intranet if requested from a host, which is different from one where the private server runs. This flag cannot be used for DNS servers.
  - Stateful server:
    - S={yes|no}. The default is no.
      Indication of stateful server, which allows only dedicated socket (stateful) connections. This tag is not allowed for HTTP* and DNS servers.
  - Note: If several configuration lines for a particular service have Q=value flag, then the quorum is the minimal value among those specified. Q=no or Q=0 defines an active service entry (as if the Q flag were not specified at all).
  - Content type indication:
    - C=type/subtype [no default]
      specification of Content-Type (including encoding), which server accepts. The value of this flag gets added automatically to any HTTP packet sent to the server by SERVICE connector. However, in order to communicate, a client still has to know and generate the data type accepted by the server, i.e. a protocol, which server understands. This flag just helps insure that HTTP packets all get proper content type, defined at service configuration. This tag is not allowed in DNS server specifications.
  - Bonus coefficient:

- ♦ B=double [0.0 = default]
  specifies a multiplicative bonus given to a server run locally, when calculating reachability rate. Special rules apply to negative/zero values: 0.0 means not to use the described rate increase at all (default rate calculation is used, which only slightly increases rates of locally run servers). Negative value denotes that locally run server should be taken in first place, regardless of its rate, if that rate is larger than percent of expressed by the absolute value of this coefficient of the average rate coefficient of other servers for the same service. That is -5 instructs to ignore locally run server if its status is less than 5% of average status of remaining servers for the same service.

— Validity period:

- ♦ T=integer [0 = default]
  specifies the time in seconds this server entry is valid without update. (If equal to 0 then defaulted by the LBSM Daemon to some reasonable value.)

Server descriptors of type NAMEHOLD are special. As arguments, they have only a server type keyword. The namehold specification informs the daemon that the service of this name and type is not to be defined later in any configuration file except for the current one. Also, if the host is specified, then this protection works only for the service name on the particular host. The port number is ignored (if specified).

Note: it is recommended that a dummy port number (such as :0) is always put in the namehold specifications to avoid ambiguities with treating the server type as a host name. The following example disables TestService of type DNS from being defined in all other configuration files included later, and TestService2 to be defined as a NCBID service on host foo:

```
TestService NAMEHOLD :0 DNS
TestService2 NAMEHOLD foo:0 NCBID
```

Signals

The table below describes the LBSMD daemon signal processing.

Signal	Reaction
SIGHUP	reload the configuration
SIGINT	quit
SIGTERM	quit
SIGUSR1	toggle the verbosity level between less verbose (default) and more verbose (when every warning generated is stored) modes

Automatic Configuration Distribution

The configuration files structure is unified for all the hosts in the NCBI network. It is shown on the figure below.

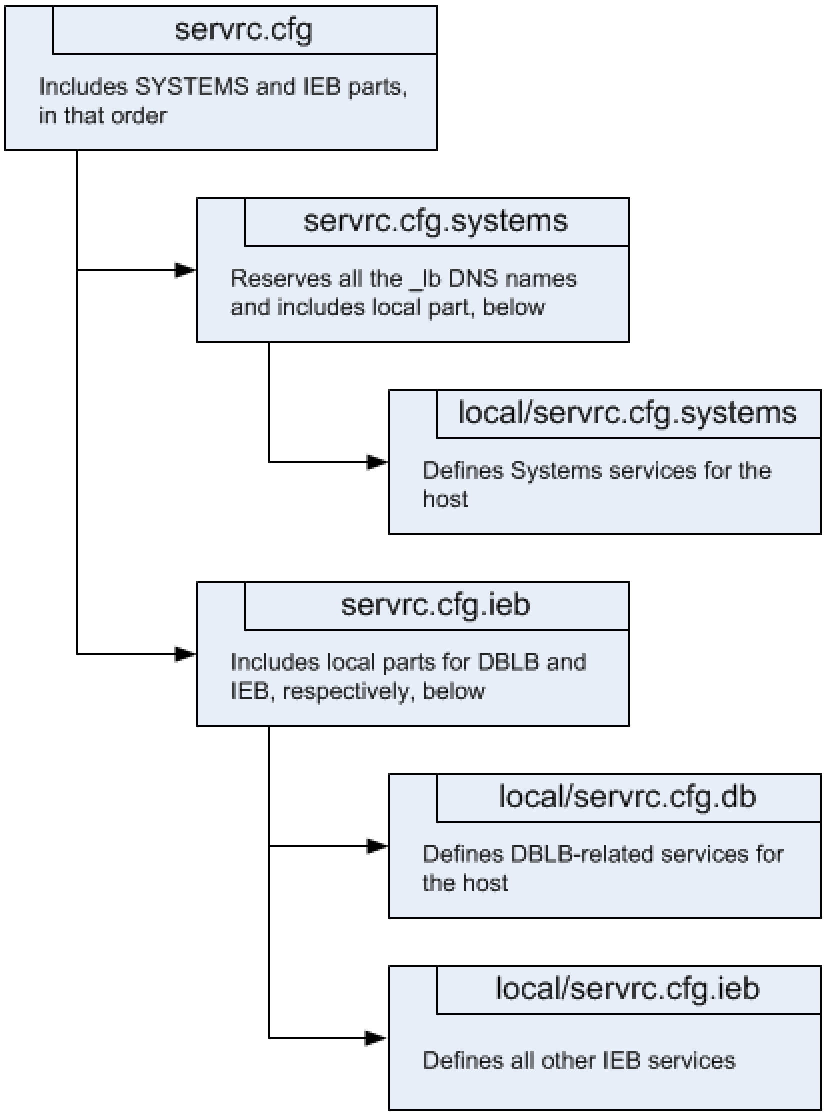

servrc.cfg
Includes SYSTEMS and IEB parts, in that order
servrc.cfg.systems
Reserves all the _lb DNS names and includes local part, below
local/servrc.cfg.systems
Defines Systems services for the host
servrc.cfg.ieb
Includes local parts for DBLB and IEB, respectively, below
local/servrc.cfg.db
Defines DBLB-related services for the host
local/servrc.cfg.ieb
Defines all other IEB services

Figure 9. LBSMD Configuration Files Structure

The common for all the configuration file prefix /etc/lbsmd is omitted on the figure. The arrows on the diagram show how the files are included.

The files servrc.cfg and servrc.cfg.systems have fixed structure and should not be changed at all. The purpose of the file local/servrc.cfg.systems is to be modified by the systems group while the purpose of the file local/servrc.cfg.ieb isto be modified by the delegated members of the respected groups. To make it easier for changes all the local/servrc.cfg.ieb files from all the hosts in the NCBI network are stored in a centralized SVN repository. The repository can be received by issuing the following command:

```
svn co svn+ssh://subvert.be-md.ncbi.nlm.nih.gov/export/home/LBSMD_REPO
```

The file names in that repository match the following pattern:

```
hostname.{be-md|st-va}[.qa]
```

where be-md is used for Bethesda, MD site and st-va is used for Sterling, VA site. The optional .qa suffix is used for quality assurance department hosts.

So, if it is required to change the /etc/lbsmd/local/servrc.cfg.ieb file on the sutils1 host in Bethesda the sutils1.be-md file is to be changed in the repository.

As soon as the modified file is checked in the file will be delivered to the corresponding host with the proper name automatically. The changes will take effect in a few minutes. The process of the configuration distribution is illustrated on the figure below.

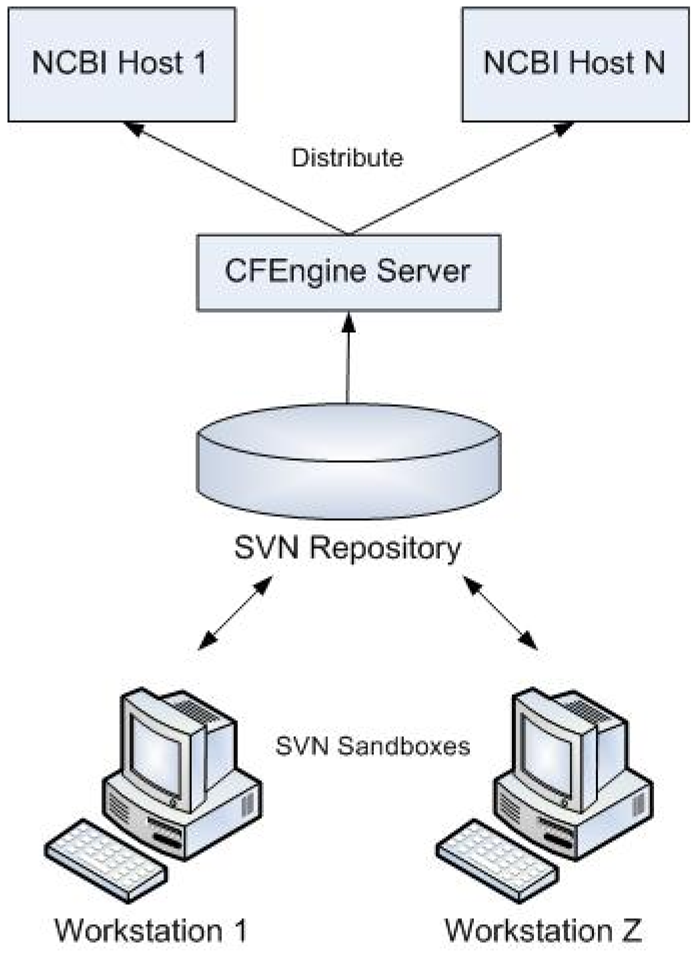


Figure 10. Automatic Configuration Distribution

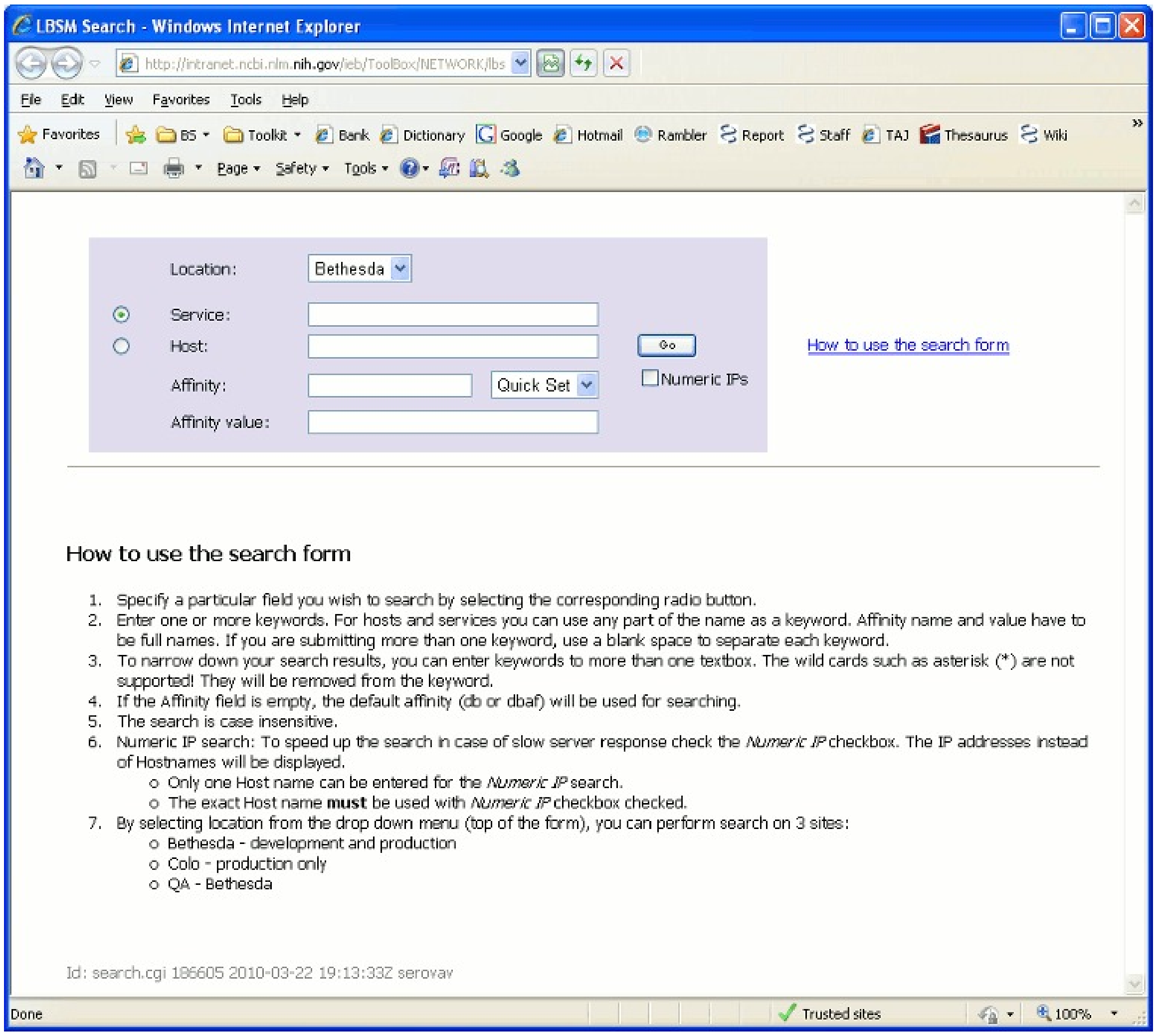
Monitoring and Control

Service Search

The following web page can be used to search for a service:

http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmc/search.cgi

The following screen will appear


LBSM Search - Windows Internet Explorer
http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbs
File Edit View Favorites Tools Help
Favorites BS Toolkit Bank Dictionary Google Hotmail Rambler Report Staff TAJ Thesaurus Wiki
Page Safety Tools
Go
Quick Set
How to use the search form
Done
Trusted sites
100%

Location:	Bethesda
● Service:	
○ Host:	
Affinity:	
Affinity value:	

☐ Numeric IPs

How to use the search form

1. Specify a particular field you wish to search by selecting the corresponding radio button.
2. Enter one or more keywords. For hosts and services you can use any part of the name as a keyword. Affinity name and value have to be full names. If you are submitting more than one keyword, use a blank space to separate each keyword.
3. To narrow down your search results, you can enter keywords to more than one textbox. The wild cards such as asterisk (*) are not supported! They will be removed from the keyword.
4. If the Affinity field is empty, the default affinity (db or dbaf) will be used for searching.
5. The search is case insensitive.
6. Numeric IP search: To speed up the search in case of slow server response check the *Numeric IP* checkbox. The IP addresses instead of Hostnames will be displayed.
   - Only one Host name can be entered for the *Numeric IP* search.
   - The exact Host name **must** be used with *Numeric IP* checkbox checked.
7. By selecting location from the drop down menu (top of the form), you can perform search on 3 sites:
   - Bethesda - development and production
   - Colo - production only
   - QA - Bethesda

Id: search.cgi 186605 2010-03-22 19:13:33Z serovav

Figure 11. NCBI Service Search Page

As an example of usage a user might enter the partial name of the service like "TaxService" and click on the “Go” button. The search results will display "TaxService", "TaxService3" and "TaxService3Test" if those services are available (see http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmc/search.cgi?key=rb_svc&service=TaxService&host=&button=Go&db=).

lbsmc Utility

Another way of monitoring the LBSMD daemon is using the lbsmc (http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/CPP_DOC/lxr/source/src/connect/daemons/lbsmc.c) utility. The utility periodically dumps onto the screen a table which represents the current content of the LBSMD daemon table. The utility output can be controlled by a number of command line options. The full list of available options and their description can be obtained by issuing the following command:

```
lbsmc -h
```

The NCBI intranet users can also get the list of options by clicking on this link: http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmc.cgi?-h.

For example, to print a list of hosts which names match the pattern "sutil*" the user can issue the following command:

```
 >./lbsmc -h sutil* 0
LBSMC - Load Balancing Service Mapping Client R100432
03/13/08 16:20:23 ====== widget3.be-md.ncbi.nlm.nih.gov (00:00) ======= [2]
V1.2
Hostname/IPaddr Task/CPU LoadAv LoadBl Joined Status StatBl
sutils1 151/4 0.06 0.03 03/12 13:04 397.62 3973.51
sutils2 145/4 0.17 0.03 03/12 13:04 155.95 3972.41
sutils3 150/4 0.20 0.03 03/12 13:04 129.03 3973.33
-----------------------------------------------------------------------------
---
Service T Type Hostname/IPaddr:Port LFS B.Rate Coef Rating
bounce +25 NCBID sutils1:80 L 1000.00 397.62
bounce +25 HTTP sutils1:80 1000.00 397.62
bounce +25 NCBID sutils2:80 L 1000.00 155.95
bounce +25 HTTP sutils2:80 1000.00 155.95
bounce +27 NCBID sutils3:80 L 1000.00 129.03
bounce +27 HTTP sutils3:80 1000.00 129.03
dispatcher_lb 25 DNS sutils1:80 1000.00 397.62
dispatcher_lb 25 DNS sutils2:80 1000.00 155.95
dispatcher_lb 27 DNS sutils3:80 1000.00 129.03
MapViewEntrez 25 STANDALONE sutils1:44616 L S 1000.00 397.62
MapViewEntrez 25 STANDALONE sutils2:44616 L S 1000.00 155.95
MapViewEntrez 27 STANDALONE sutils3:44616 L S 1000.00 129.03
MapViewMeta 25 STANDALONE sutils2:44414 L S 0.00 0.00
MapViewMeta 27 STANDALONE sutils3:44414 L S 0.00 0.00
MapViewMeta 25 STANDALONE sutils1:44414 L S 0.00 0.00
sutils_lb 25 DNS sutils1:80 1000.00 397.62
sutils_lb 25 DNS sutils2:80 1000.00 155.95
sutils_lb 27 DNS sutils3:80 1000.00 129.03
TaxService 25 NCBID sutils1:80 1000.00 397.62
TaxService 25 NCBID sutils2:80 1000.00 155.95
TaxService 27 NCBID sutils3:80 1000.00 129.03
TaxService3 +25 HTTP_POST sutils1:80 1000.00 397.62
TaxService3 +25 HTTP_POST sutils2:80 1000.00 155.95
TaxService3 +27 HTTP_POST sutils3:80 1000.00 129.03
test +25 HTTP sutils1:80 1000.00 397.62
test +25 HTTP sutils2:80 1000.00 155.95
test +27 HTTP sutils3:80 1000.00 129.03
testgenomes_lb 25 DNS sutils1:2441 1000.00 397.62
testgenomes_lb 25 DNS sutils2:2441 1000.00 155.95
testgenomes_lb 27 DNS sutils3:2441 1000.00 129.03
testsutils_lb 25 DNS sutils1:2441 1000.00 397.62
testsutils_lb 25 DNS sutils2:2441 1000.00 155.95
testsutils_lb 27 DNS sutils3:2441 1000.00 129.03
-----------------------------------------------------------------------------
---
```

```
* Hosts:4\747, Srvrs:44/1223/23 | Heap:249856, used:237291/249616, free:240 *
LBSMD PID: 17530, config: /etc/lbsmd/servrc.cfg
```

NCBI Intranet Web Utilities

The NCBI intranet users can also visit the following quick reference links:

- Dead servers list: http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmc.cgi?-h+none+-w+-d
- Search engine for all available hosts, all services and database affiliation: http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmc/search.cgi?key=rb_svc&service=&host=&button=Go&db=

If the lbsmc utility is run with the -f option then the output contains two parts:

- The host table. The table is accompanied by raw data which are printed in the order they appear in the LBSMD daemon table.
- The service table

The output is provided in either long or short format. The format depends on whether the -w option was specified in the command line (the option requests the long (wide) output). The wide output occupies about 130 columns, while the short (normal) output occupies 80 which is the standard terminal width.

In case if the service name is more than the allowed number of characters to display the trailing characters will be replaced with “>”. When there is more information about the host / service to be displayed the “+” character is put beside the host / service name (this additional information can be retrieved by adding the -i option). When both “+” and “>” are to be shown they are replaced with the single character “*”. In the case of wide-output format the “#” character shown in the service line means that there is no host information available for the service (similar to the static servers). The “!” character in the service line denotes that the service was configured / stored with an error (this character actually should never appear in the listings and should be reported whenever encountered). Wide output for hosts contains the time of bootup and startup. If the startup time is preceded by the “~” character then the host was gone for a while and then came back while the lbsmc utility was running. The “+” character in the times is to show that the date belongs to the past year(s).

Server Penalizer API and Utility

The utility allows to report problems of accessing a certain server to the LBSMD daemon, in the form of a penalty which is a value in the range [0..100] that shows, in percentages, how bad the server is. The value 0 means that the server is completely okay, whereas 100 means that the server (is misbehaving and) should **not** be used at all. The penalty is not a constant value: once set, it starts to decrease in time, at first slowly, then faster and faster until it reaches zero. This way, if a server was penalized for some reason and later the problem has been resolved, then the server becomes available gradually as its penalty (not being reset by applications again in the absence of the offending reason) becomes zero. The figure below illustrates how the value of penalty behaves.

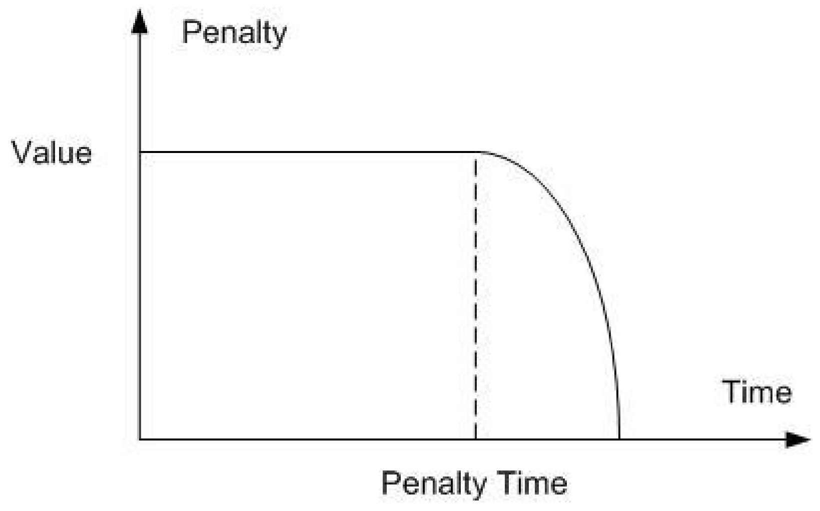


Figure 12. Penalty Value Characteristics

Technically, the penalty is maintained by a daemon, which has the server configured, i.e., received by a certain host, which may be different from the one where the server was put into the configuration file. The penalty first migrates to that host, and then the daemon on that host announces that the server was penalized.

Note: Once a daemon is restarted, the penalty information is lost.

Service mapping API has a call SERV_Penalize() (http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/ident?i=SERV_Penalize) declared in connect/ncbi_service.h (http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/include/connect/ncbi_service.h), which can be used to set the penalty for the last server obtained from the mapping iterator.

For script files (similar to the ones used to start/stop servers), there is a dedicated utility program called lbsm_feedback (http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/CPP_DOC/lxr/source/src/connect/daemons/lbsm_feedback.c), which sets the penalty from the command line. This command should be used with extreme care because it affects the load-balancing mechanism substantially,.

lbsm_feedback is a part of the LBSM set of tools installed on all hosts which run LBSMD. As it was explained above, penalizing means to make a server less favorable as a choice of the load balancing mechanism. Because of the fact that the full penalty of 100% makes a server unavailable for clients completely, at the time when the server is about to shut down (restart), it is wise to increase the server penalty to the maximal value, i.e. to exclude the server from the service mapping. (Otherwise, the LBSMD daemon might not immediately notice that the server is down and may continue dispatching to that server.) Usually, the penalty takes at most 5 seconds to propagate to all participating network hosts. Before an actual server shutdown, the following sequence of commands can be used:

```
> /opt/machine/lbsm/sbin/lbsm_feedback 'Servicename STANDALONE host 100 120'
> sleep 5
now you can shutdown the server
```

The effect of the above is to set the maximal penalty 100 for the service Servicename (of type STANDALONE) running on host host for at least 120 seconds. After 120 seconds the penalty value will start going down steadily and at some stage the penalty becomes 0. The default hold time equals 0. It takes some time to deliver the penalty value to the other hosts on the network so ‘sleep 5’ is used. Please note the single quotes surrounding the penalty specification: they are required in a command shell because lbsm_feedback takes only one argument which is the entire penalty specification.

As soon as the server is down, the LBSMD daemon detects it in a matter of several seconds (if not instructed otherwise by the configuration file) and then does not dispatch to the server until it is back up. In some circumstances, the following command may come in handy:

```
 > /opt/machine/lbsm/sbin/lbsm_feedback 'Servicename STANDALONE host 0'
```

The command resets the penalty to 0 (no penalty) and is useful when, as for the previous example, the server is restarted and ready in less than 120 seconds, but the penalty is still held high by the LBSMD daemon on the other hosts.

The formal description of the lbsm_feedback utility parameters is given below.

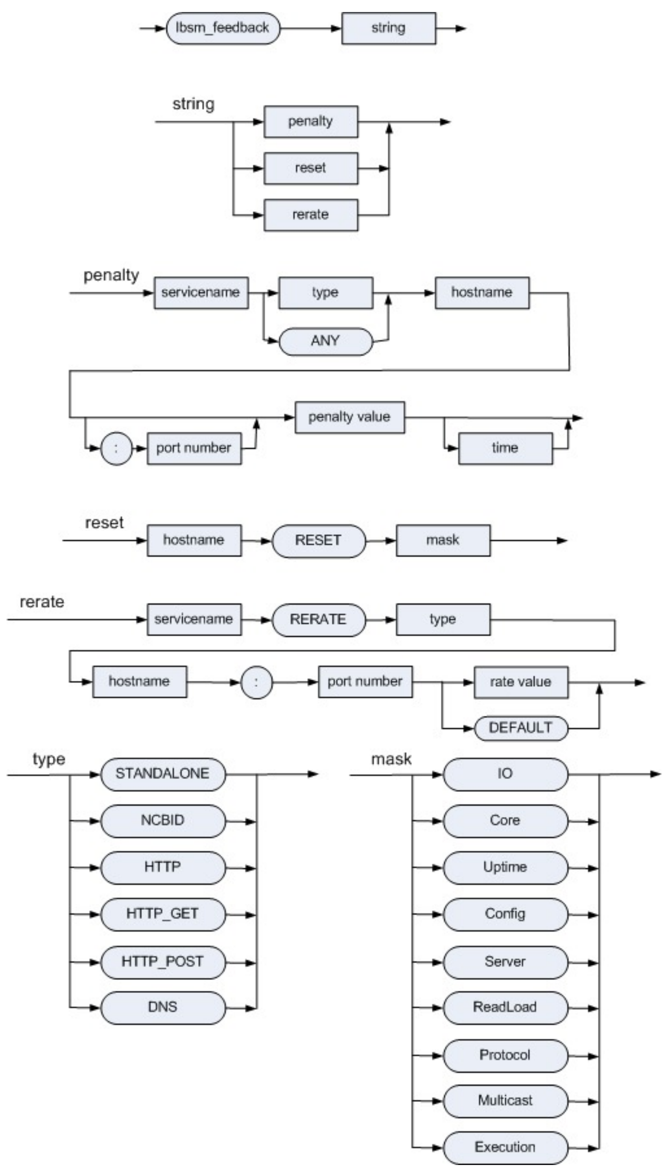


Figure 13. lbsm_feedback Arguments

The servicename can be an identifier with '*' for any symbols and / or '?' for a single character. The penalty value is an integer value in the range 0 ... 100. The port number and time are integers. The host name is an identifier and the rate value is a floating point value.

SVN Repository

The SVN repository where the LBSMD daemon source code is located can be retrieved by issuing the following command:

```
svn co https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/c++
```

The daemon code is in this file:

```
c++/src/connect/daemons/lbsmd.c
```

Log Files

The LBSMD daemon stores its log files at the following location:

/var/log/lbsmd

The file is formed locally on a host where LBSMD daemon is running. The log file size is limited to prevent the disk being flooded with messages. A standard log rotation is applied to the log file so you may see the files:

/var/log/lbsmd.X.gz

where X is a number of the previous log file.

The log file size can be controlled by the -s command line option. By default, -s 0 is the active flag, which provides a way to create (if necessary) and to append messages to the log file with no limitation on the file size whatsoever. The -s -1 switch instructs indefinite appending to the log file, which must exist. Otherwise, log messages are not stored. -s positive_number restricts the ability to create (if necessary) and to append to the log file until the file reaches the specified size in kilobytes. After that, message logging is suspended, and subsequent messages are discarded. Note that the limiting file size is only approximate, and sometimes the log file can grow slightly bigger. The daemon keeps track of log files and leaves a final logging message, either when switching from one file to another, in case the file has been moved or removed, or when the file size has reached its limit.

NCBI intranet users can get few (no more than 100) recent lines of the log file on an NCBI internal host. It is also possible to visit the following link:

http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmd.cgi?log

Configuration Examples

Here is an example of a LBSMD configuration file:

```
# $Id$
#
# This is a configuration file of new NCBI service dispatcher
#
#
# DBLB interface definitions
%include /etc/lbsmd/servrc.cfg.db
# IEB's services
testHTTP /Service/test.cgi?Welcome L=no
Entrez2[0] HTTP_POST www.ncbi.nlm.nih.gov /entrez/eutils/entrez2server.fcgi \
 C=x-ncbi-data/x-asn-binary L=no
Entrez2BLAST[0] HTTP_POST www.ncbi.nlm.nih.gov /entrez/eutils/
entrez2server.cgi \
 C=x-ncbi-data/x-asn-binary L=yes
CddSearch [0] HTTP_POST www.ncbi.nlm.nih.gov /Structure/cdd/c_wrpsb.cgi \
 C=application/x-www-form-urlencoded L=no
CddSearch2 [0] HTTP_POST www.ncbi.nlm.nih.gov /Structure/cdd/wrpsb.cgi \
 C=application/x-www-form-urlencoded L=no
StrucFetch [0] HTTP_POST www.ncbi.nlm.nih.gov /Structure/mmdb/mmdbsrv.cgi \
 C=application/x-www-form-urlencoded L=no
bounce[60]HTTP /Service/bounce.cgi L=no C=x-ncbi-data/x-unknown
```

```
# Services of old dispatcher
bounce[60]NCBID '' L=yes C=x-ncbi-data/x-unknown | \
..../web/public/htdocs/Service/bounce
```

NCBI intranet users can also visit the following link to get a sample configuration file:

http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/NETWORK/lbsmd.cgi?cfg

Database Load Balancing

Database load balancing is an important part of the overall load balancing function. Please see the Database Load Balancer section in the Database Access chapter for more details.

Cookie / Argument Affinity Module (MOD_CAF)

Overview

The cookie / argument affinity module (CAF module in the further discussion) helps to virtualize and to dispatch a web site by modifying the way how Apache resolves host names. It is done by superseding conventional gethostbyname*() API. The CAF module is implemented as an Apache web server module and uses the LBSMD daemon collected data to make a decision how to dispatch a request. The data exchange between the CAF module and the LBSMD daemon is done via a shared memory segment as shown on the figure below.

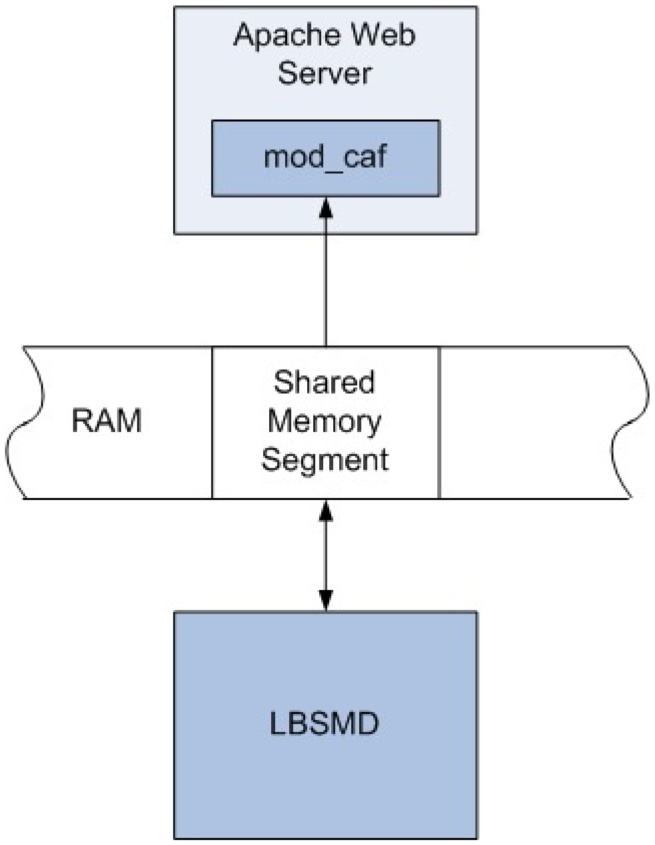


Figure 14. CAF Module and LBSMD daemon data exchange

The LBSMD daemon stores all the collected data in a shared memory segment and the CAF module is able to read data from that segment.

The CAF module looks for special cookies and query line arguments, and analyses the LBSMD daemon data to resolve special names which can be configured in ProxyPass directives of mod_proxy.

The CAF module maintains a list of proxy names, cookies, and arguments (either 4 predefined, see below, or set forth via Apache configuration file by CAF directives) associated with cookies. Once a URL is translated to the use of one of the proxies (generally, by ProxyPass of mod_proxy) then the information from related cookie (if any) and argument (if any) is used to find the best matching real host that corresponds to the proxy. Damaged cookies and arguments, if found in the incoming HTTP request, are ignored.

A special host name is meant under proxy and the name contains a label followed by string ".lb" followed by an optional domain part. Such names trigger gethostbyname() substitute, supplied by the module, to consult load-balancing daemon's table, and to use both the constraints on the arguments and the preferred host information, found in the query string and the cookie, respectively.

For example, the name "pubmed.lb.nlm.nih.gov" is an LB proxy name, which would be resolved by looking for special DNS services ("pubmed_lb" in this example) provided by the LBSMD daemon. Argument matching (see also a separate section below) is done by searching the host environment of target hosts (corresponding to the LB proxy name) as supplied by the LBSMD daemon. That is, "db=PubMed" (to achieve PubMed database affinity) in the query that transforms into a call to an LB proxy, which in turn is configured to use the argument "DB", instructs to search only those target hosts that declare the proxy and have "db=... PubMed ..." configured in their LBSMD environments (and yet to remember to accommodate, if it is possible, a host preference from the cookie, if any found in the request).

The CAF module also detects internal requests and allows them to use the entire set of hosts that the LB names are resolved to. For external requests, only hosts whose DNS services are not marked local (L=yes, or implicitly, by lacking "-i" flag in the LBSMD daemon launch command) will be allowed to serve requests. "HTTP_CAF_PROXIED_HOST" environment is supplied (by means of an HTTP header tag named "CAF-Proxied-Host") to contain an address of the actual host posted the request. Impostor's header tags (if any) of this name are always stripped, so that backends always have correct information about the requesters. Note that all internal requests are trusted, so that an internal resource can make a request to execute on behalf of an outside client by providing its IP in the "Client-Host" HTTP header. The "Client-Host" tag gets through for internal requests only; to maintain security the tag is dropped for all external requests.

The CAF module has its own status page that can be made available in the look somewhat resembling Apache status page. The status can be in either raw or HTML formatted, and the latter can also be sorted using columns in interest. Stats are designed to be fast, but sometimes inaccurate (to avoid interlocking, and thus latencies in request processing, there are no mutexes being used except for the table expansion). Stats are accumulated between server restarts (and for Apache 2.0 can survive graceful restarts, too). When the stat table is full (since it has a fixed size), it is cleaned in a way to get room for 1% of its capacity, yet trying to preserve the most of recent activity as well as the most of heavily used stats from the past. There are two cleaning algorithms currently implemented, and can be somehow tuned by means of CAFexDecile, CAFexPoints, and CAFexSlope directives which are described below.

The CAF module can also report the number of slots that the Apache server has configured and used up each time a new request comes in and is being processed. The information resides in a shared memory segment that several Apache servers can use cooperatively on the same machine. Formerly, this functionality has been implemented in a separate SPY module, which is now fully integrated into this module. Using a special compile-time macro it is possible to obtain the former SPY-only functionality (now called LBSMD reporter feature) without any other CAF features. Note that no CAF* directives will be recognized in Apache configuration, should the reduced functionality build be chosen.

Configuration

The table below describes Apache configuration directives which are taken into account by the CAF module.

Directive	Description
LBSMD { On | Off }	It can appear outside any paired section of the configuration file, and enables ["On", default in mod_spy mode] or disables ["Off", default in full-fledged mod_caf mode] the LBSMD reporter feature. When the module is built exclusively with the LBSMD reporter feature, this is the only directive, which is available for the use by the module. Please note that the directive is extremely global, and works across configuration files. Once "Off" is found throughout the configuration, it takes the effect.
CAF { On | Off }	It can appear outside any paired section of the configuration file, and enables ["On", default] or disables ["Off"] the entire module. Please note that this directive is extremely global, and works across Apache configuration files, that is the setting "Off" anywhere in the configuration causes the module to go out of business completely.
CAFQAMap name path	It can appear outside any paired section of the configuration file but only once in the entire set of the configuration files per "name", and if used, defines a path to the map file, which is to be loaded at the module initialization phase (if the path is relative, it specifies the location with respect to the daemon root prefix as defined at the time of the build, much like other native configuration locations do). The file is a text, line-oriented list (w/o line continuations). The pound symbol (#) at any position introduces a comment (which is ignored by the parser). Any empty line (whether resulted from cutting off a comment, or just blank by itself) is skipped. Non-empty lines must contain a pair of words, delimited by white space(s) (that is, tab or space character(s)). The first word defines an LB group that is to be replaced with the second word, in the cases when the first word matches the LB group used in proxy passing of an internally-originating request. The matching is done by previewing a cookie named "name" that should contain a space-separated list of tokens, which must comprise a subset of names loaded from the left-hand side column of the QA file. Any unmatched token in the cookie will result the request to fail, so will do any duplicate name. That is, if the QA map file contains a paired rule "tpubmed tpubmedqa", and an internal (i.e. originating from within NCBI) request has the NCBIQA cookie listing "tpubmed", then the request that calls for use of the proxy-pass "tpubmed.lb" will actually use the name "tpubmedqa.lb" as if it appeared in the ProxyPass rule of mod_proxy. Default is not to load any QA maps, and not to proceed with any substitutions. Note that if the module is disabled (CAF Off), then the map file, even if specified, need not to exist, and won't be loaded.
CAFFailoverIP address	It defines hostname / IP to return on LB proxy names that cannot be resolved. Any external requests and local ones, in which argument affinity has to be taken into account, will fall straight back to use this address whenever the LB name is not known or LBSMD is not operational. All other requests will be given a chance to use regular DNS first, and if that fails, then fall back to use this IP. When the failover IP address is unset, a failed LB proxy name generally causes the Apache server to throw either "Bad gateway" (502) or "Generic server error" (500) to the client. This directive is global across the entire configuration, and the last setting takes the actual effect.
CAFForbiddenIP address	It is similar to CAFFailoverIP described above yet applies only to the cases when the requested LB DNS name exists but cannot be returned as it would cause the name access violation (for example, an external access requires an internal name to be used to proxy the request). Default is to use the failover IP (as set by CAFFailoverIP), if available.
CAFThrottleIP address	It is similar to CAFFailoverIP described above but applies only to abusive requests that should be throttled out. Despite this directive exists, the actual throttling mechanism is not yet in production. Default is to use the failover IP (as set by CAFFailoverIP), if available.
CAFBusyIP address	It is similar to CAFFailoverIP described above but gets returned to clients when it is known that the proxy otherwise serving the request is overloaded. Default is to use the failover IP, if available.

CAFDebug { Off | On | 2 | 3 }	It controls whether to print none ("Off"), some ("On"), more ("2"), or all ("3") debugging information into Apache log file. Per-request logging is automatically on when debugging is enabled by the native LogLevel directive of Apache (LogLevel debug), or with a command line option -e (Apache 2). This directive controls whether mod_caf produces additional logging when doing maintenance cleaning of its status information (see CAFMaxNStats below). Debug level 1 (On) produces cleanup synopsis and histogram, level 2 produces per-stat eviction messages and the synopsis, and debug level 3 is a combination of the above. Default is "Off". The setting is global, and the last encounter has the actual effect. NOTE: per-stat eviction messages may cause latencies in request processing; so debug levels "2" and "3" should be used carefully, and only when actually needed.
CAFTiming { Off | On | TOD }	It controls whether the module timing profile is done while processing requests. For this to work, though, CAFMaxNStats must first enable collection of statistics. Module's status page then will show how much time is being spent at certain stages of a request processing. Since proxy requests and non-proxy requests are processed differently they are accounted separately. "On" enables to make the time marks using the gettimeofday(2) syscall (accurate up to 1us) without reset upon each stat cleanup (note that tick count will wrap around rather frequently). Setting "TOD" is same as "On" but extends it so that counts do get reset upon every cleanup. Default is "Off". The setting is global, and the last encounter in the configuration file has the actual effect.
CAFMaxNStats number	The number defines how many statistics slots are allocated for CAF status (aka CAF odometer). Value "0" disables the status page at all. Value "-1" sets default number of slots (which currently corresponds to the value of 319). Note that the number only sets a lower bound, and the actual number of allocated slots may be automatically extended to occupy whole number of pages (so that no "memory waste" occurs). The actual number of stats (and memory pages) is printed to the log file. To access the status page, a special handler must be installed for a designated location, as in the following example: <Location /caf-status> SetHandler CAF-status Order deny,allow Deny from all Allow from 130.14/16 </Location> 404 (Document not found) gets returned from the configured location if the status page has been disabled (number=0), or if it malfunctions. This directive is global across the entire configuration, and the last found setting takes the actual effect. CAF stats can survive server restarts [graceful and plain "restart"], but not stop / start triggering sequence. Note: "CAF Off" does not disable the status page if it has been configured before -- it just becomes frozen. So [graceful] restart with "CAF Off" won't prevent from gaining access to the status page, although the rest of the module will be rendered inactive.
CAFUrlList url1 url2 ...	By default, CAF status does not distinguish individual CGIs as they are being accessed by clients. This option allows separating statistics on a per-URL basis. Care must be taken to remember of "combinatorial explosion", and thus the appropriate quantity of stats is to be pre-allocated with CAFMaxNStats if this directive is used, or else the statistics may renew too often to be useful. Special value "*" allows to track every (F)CGI request by creating individual stat entries for unique (F)CGI names (with or without the path part, depending on a setting of CAFStatPath directive, below). Otherwise, only those listed are to be accounted for, leaving all others to accumulate into a nameless stat slot. URL names can have .cgi or .fcgi file name extensions. Alternatively, a URL name can have no extension to denote a CGI, or a trailing period (.) to denote an FCGI. A single dot alone (.) creates a specially named stat for all non-matching CGIs (both .cgi or .fcgi), and collects all other non-CGI requests in a nameless stat entry. (F)CGI names are case sensitive. When path stats are enabled (see CAFStatPath below), a relative path entry in the list matches any (F)CGI that has the trailing part matching the request (that is, "query.fcgi" matches any URL that ends in "query.fcgi", but "/query.fcgi" matches only the top-level ones). There is an internal limit of 1024 URLs that can be explicitly listed. Successive directives add to the list. A URL specified as a minus sign alone ("-") clears the list, so that no urls will be registered in stats. This is the default. This directive is only allowed at the top level, and applies to all virtual hosts.

CAFUrlKeep url1 url2 ...	CAF status uses a fixed-size array of records to store access statistics, so whenever the table gets full, it has to be cleaned up by dropping some entries, which have not been updated too long, have fewer count values, etc. The eviction algorithm can be controlled by CAFexDecile, CAFexPoints, and CAFexSlope directives, described below, but even when finely tuned, can result in some important entries being pre-emptied, especially when per-URL stats are enabled. This directive helps avoid losing the important information, regardless of other empirical characteristics of a candidate-for-removal. The directive, like CAFUrlList above, lists individual URLs which, once recorded, have to be persistently kept in the table. Note that as a side effect, each value (except for "-") specified in this directive implicitly adds an entry as if it were specified with CAFUrlList. Special value "-" clears the keep list, but does not affect the URL list, so specifying "CAFUrlKeep a b -" is same as specifying "CAFUrlList a b" alone, that is, without obligation for CAF status to keep either "a" or "b" permanently. There is an internal limit of 1024 URLs that can be supplied by this directive. Successive uses add to the list. The directive is only allowed at the top level, and applies to all virtual hosts.
CAFexDecile digit	It specifies the top decile(s) of the total number of stat slots, sorted by the hit count and subject for expulsion, which may not be made available for stat's cleanup algorithms should it be necessary to arrange a new slot by removing old/stale entries. Decile is a single digit 0 through 9, or a special value "default" (which currently translates to 1). Note that each decile equals 10%.
CAFexPoints { value | percentage% }	The directive specifies how many records, as an absolute value, or as a percentage of total stat slots, are to be freed each time the stat table gets full. Keyword "default" also can be used, which results in eviction of 1% of all records (or just 1 record, whatever is greater). Note that if CAFUrlKeep is in use, the cleanup may not be always possible. The setting is global and the value found last takes the actual effect.
CAFexSlope { value | "quad" }	The directive can be used to modify cleanup strategy used to vacate stat records when the stat table gets full. The number of evicted slots can be controlled by CAFexPoints directive. The value, which is given by this directive, is used to plot either circular ("quad") or linear (value >= 0) plan of removal. The linear plan can be further fine-tuned by specifying a co-tangent value of the cut-off line over a time-count histogram of statistics, as a binary logarithm value, so that 0 corresponds to the co-tangent of 1 (=2^0), 1 (default) corresponds to the co-tangent of 2 (=2^1), 2 - to the co-tangent of 4 (=2^2), 3 - to 8 (=2^3), and so forth, up to a maximal feasible value 31 (since 2^32 overflows an integer, this results in the infinite co-tangent, causing a horizontal cut-off line, which does not take into account times of last updates, but counts only). The default co-tangent (2) prices the count of a stats twice higher than its longevity. The cleanup histogram can be viewed in the log if CAFDebug is set as 2 (or 3). The setting is global and the value found last takes the actual effect.
CAFStatVHost { Off | On }	It controls whether VHosts of the requests are to be tracked on the CAF status page. By default, VHost separation is not done. Note that preserving graceful restart of the server may leave some stats VHost-less, when switching from VHost-disabled to VHost-enabled mode, with this directive. The setting is global and the setting found last has the actual effect.
CAFStatPath { On | Off }	It controls whether the path part of URLs is to be stored and shown on the CAF status page. By default, the path portion is stripped. Keep in mind the relative path specifications as given in CAFUrlList directive, as well as the number of possible combinations of Url/VHost/Path, that can cause frequent overflows of the status table. When CAFStatPath is "Off", the path elements are stripped from all URLs provided in the CAFUrlList directive (and merging the identical names, if any result). This directive is global, and the setting found last having the actual effect.
CAFOkDnsFallback { On | Off }	It controls whether it is okay to fallback for consulting regular DNS on the unresolved names, which are not constrained with locality and/or affinities. Since shutdown of SERVNSD (which provided the fake .lb DNS from the load balancer), fallback to system DNS looks painfully slow (at it has now, in the absence of the DNS server, to reach the timeout), so the default for this option is "Off". The setting is global, and the value found last takes the actual effect.
CAFNoArgOnGet { On | Off }	It can appear outside any paired section of the configuration, "On" sets to ignore argument assignment in GET requests that don't have explicit indication of the argument. POST requests are not affected. Default is "Off", VHost-specific.
CAFArgOnCgiOnly { On | Off }	It controls whether argument is taken into account when an FCGI or CGI is being accessed. Default is "Off". The setting is per-VHost specific.

CAFCookies { Cookie | Cookie2 | Any }	It instructs what cookies to search for: "Cookie" stands for RFC2109 cookies (aka Netscape cookies), this is the default. "Cookie2" stands for new RFC2965 cookies (new format cookies). "Any" allows searching for both types of cookies. This is a per-server option that is not shared across virtual host definitions, and allowed only outside any <Directory> or <Location>. Note that, according to the standard, cookie names are not case-sensitive.
CAFArgument argument	It defines argument name to look for in the URLs. There is no default. If set, the argument becomes default for any URL and also for proxies whose arguments are not explicitly set with CAFProxyArgument directives. The argument is special case sensitive: first, it is looked up "as-is" and, if that fails, in all uppercase then. This directive can appear outside any <Directory> or <Location> and applies to virtual hosts (if any) independently.
CAFHtmlAmp { On | Off }	It can appear outside any paired section of configuration, set to On enables to recognize "&amp;" for the ampersand character in request URLs (caution: "&amp;" in URLs is not standard-conforming). Default is "Off", VHost-specific.
CAFProxyCookie proxy cookie	It establishes a correspondence between LB DNS named proxy and a cookie. For example, "CAFProxyCookie pubmed.lb MyPubMedCookie" defines that "MyPubMedCookie" should be searched for preferred host information when "pubmed.lb" is being considered as a target name for proxying the incoming request. This directive can appear anywhere in configuration, but is hierarchy complying.
CAFProxyNoArgOnGet proxy { On | Off | Default }	The related description can be seen at the CAFNoArgOnGet directive description above. The setting applies only to the specified proxy. "Default" (default) is to use the global setting.
CAFProxyArgOnCgiOnly proxy { On | Off | Default }	The related description can be seen at the CAFArgOnCgiOnly directive description above. The setting applies only to the specified proxy. "Default" (default) is to use the global setting.
CAFProxyArgument proxy argument	It establishes a correspondence between LB DNS named proxy and a query line argument. This directive overrides any default that might have been set with global "CAFArgument" directive. Please see the list of predefined proxies below. The argument is special case sensitive: first, it is looked up "as-is" and, if that fails, in all uppercase then. The first argument occurrence is taken into consideration. It can appear anywhere in configuration, but is hierarchy complying.
CAFProxyAltArgument proxy altargument	It establishes a correspondence between LB DNS named proxy and an alternate query line argument. The alternate argument (if defined) is used to search (case-insensitively) query string for the argument value, but treating the value as if it has appeared to argument set forth by CAFProxyArgument or CAFArgument directives for the location in question. If no alternate argument value is found, the regular argument search is performed. Please see the list of predefined proxies below. Can appear anywhere in configuration, but is hierarchy complying, and should apply for existing proxies only. Altargument "-" deletes the alternate argument (if any). Note again that unlike regular proxy argument (set forth by either CAFArgument (globally) or CAFProxyArgument (per-proxy) directives) the alternate argument is entirely case-insensitive.
CAFProxyDelimiter proxy delimiter	It sets a one character delimiter that separates host[:port] field in the cookie, corresponding to the proxy, from some other following information, which is not pertinent to cookie affinity business. Default is '|'. No separation is performed on a cookie that does not have the delimiter -- it is then thought as been found past the end-of-line. It can appear anywhere in configuration, but is hierarchy complying.
CAFProxyPreference proxy preference	It sets a preference (floating point number from the range [0..100]) that the proxy would have if a host matching the cookie is found. The preference value 0 selects the default value which is currently 95. It can appear anywhere in configuration, but is hierarchy complying.
CAFProxyCryptKey proxy key	It sets a crypt key that should be used to decode the cookie. Default is the key preset when a cookie correspondence is created [via either "CAFProxyCookie" or "CAFProxyArgument"]. To disable cookie decrypting (e.g. if the cookie comes in as a plain text) use "". Can appear anywhere in configuration, but is hierarchy complying.

All hierarchy complying settings are inherited in directories that are deeper in the directory tree, unless overridden there. The new setting then takes effect for that and all descendant directories/locations.

There are 4 predefined proxies that may be used [or operated on] without prior declaration by either "CAFProxyCookie" or "CAFProxyArgument" directives:

LB name	CookieName	Preference	Delimiter	Crypted?	Argument	AltArg
tpubmed.lb	LB-Hint-Pubmed	95	|	yes	db	<none>
eutils.lb	LB-Hint-Pubmed	95	|	yes	db	DBAF
mapview.lb	LB-Hint-MapView	95	|	yes	<none>	<none>
blastq.lb	LB-Hint-Blast	95	|	yes	<none>	<none>

NOTE: The same cookie can be used to tie up an affinity for multiple LB proxies. On the other hand, LB proxy names are all unique throughout the configuration file.

NOTE: It is very important to keep in mind that arguments and alt-arguments are treated differently, case-wise. Alt-args are case insensitive, and are screened before the main argument (but appear as if the main argument has been found). On the other hand, main arguments are special case-sensitive, and are checked twice: "as is" first, then in all CAPs. So having both "DB" for alt-argument and "db" for the main, hides the main argument, and actually makes it case-insensitive. CAF will warn on some occurrences when it detects whether the argument overloading is about to happen (take a look at the logs).

The CAF module is also able to detect if a request comes from a local client. The /etc/ncbi/local_ips file describes the rules for making the decision.

The file is line-oriented, i.e. supposes to have one IP spec per one line. Comments are introduced by either "#" or "!", no continuation lines allowed, the empty lines are ignored.

An IP spec is a word (no embedded whitespace characters) and is either:

- a host name or a legitimate IP address
- a network specification in the form "networkIP / networkMask"
- an IP range (explained below).

A networkIP / networkMask specification can contain an IP prefix for the network (with or without all trailing zeroes present), and the networkMask can be either in CIDR notation or in the form of a full IP address (all 4 octets) expressing contiguous high-bit ranges (all the records below are equivalent):

```
130.14.29.0/24
130.14.29/24
130.14.29/255.255.255.0
130.14.29.0/255.255.255.0
```

An IP range is an incomplete IP address (that is, having less than 4 full octets) followed by exactly one dot and one integer range, e.g.:

```
130.14.26.0-63
```

denotes a host range from 130.14.26.0 thru 130.14.26.63 (including the ends),

```
130.14.8-9
```

denotes a host range from 130.14.8.0 thru 130.14.9.255 (including the ends).

Note that 127/8 gets automatically added, whether or not it is explicitly included into the configuration file. The file loader also warns if it encounters any specifications that overlap each other. Inexistent (or unreadable) file causes internal hardcoded defaults to be used - a warning is issued in this case.

Note that the IP table file is read once per Apache daemon's life cycle (and it is *not* reloaded upon graceful restarts). The complete stop / start sequence should be performed to force the IP table be reloaded.

Configuration Examples

- To define that "WebEnv" cookie has an information about "pubmed.lb" preference in "/Entrez" and all the descendant directories one can use the following:

```
<Location /Entrez>
 CAFProxyCookie pubmed.lb WebEnv
 CAFPreference pubmed.lb 100
</Location>
```

The second directive in the above example sets the preference to 100% -- this is a preference, not a requirement, so meaning that using the host from the cookie is the most desirable, but not blindly instructing to go to in every case possible.

- To define new cookie for some new LB name the following fragment can be used:

```
<Directory /SomeDir>
 CAFProxyCookie myname.lb My-Cookie
 CAFProxyCookie other.lb My-Cookie
</Directory>
<Directory /SomeDir/SubDir>
 CAFProxyCookie myname.lb My-Secondary-Cookie
</Directory>
```

The effect of the above is that "My-Cookie" will be used in LB name searches of "myname.lb" in directory "/SomeDir", but in "/SomeDir/SubDir" and all directories of that branch, "My-Secondary-Cookie" will be used instead. If an URL referred to "/SomeDir/AnotherDir", then "My-Cookie" would still be used.

Note that at the same time "My-Cookie" is used under "/SomeDir" everywhere else if "other.lb" is being resolved there.

- The following fragment disables cookie for "tpubmed.lb" [note that no "CAFProxyCookie" is to precede this directive because "tpubmed.lb" is predefined]:

```
CAFProxyPreference tpubmed.lb 0
```

- The following directive associates proxy "systems.lb" with argument "ticket":

```
CAFProxyArgument systems.lb ticket
```

The effect of the above is that if an incoming URL resolves to use "systems.lb", then "ticket", if found in the query string, would be considered for lookup of "systems.lb" with the load-balancing daemon.

Arguments Matching

Suppose that the DB=A is a query argument (explicit DB selection, including just "DB" (as a standalone argument, treated as missing value), "DB=" (missing value)). That will cause the following order of precedence in selecting the target host:

Match	Description
DB=A	Best. "A" may be "" to match the missing value
DB=*	Good. "*" stands for "any other"
DB not defined	Fair
DB=-	Poor. "-" stands for "missing in the request"
DB=B	Mismatch. It is used for fallbacks only as the last resort

No host with an explicit DB assignment (DB=B or DB=-) is being selected above if there is an exclamation point "!" [stands for "only"] in the assignment. DB=~A for the host causes the host to be skipped from selection as well. DBs are screened in the order of appearance, the first one is taken, so "DB=~A A" skips all requests having DB=A in their query strings.

Suppose that there is no DB selection in the request. Then the hosts are selected in the following order:

Match	Description
DB=-	Best "-" stands for "missing from the request"
DB not defined	Good
DB=*	Fair. "*" stands for "any other"
DB=B	Poor

No host with a non-empty DB assignment (DB=B or DB=*) is being selected in the above scenario if there is an exclamation point "!" [stands for "only"] in the assignment. DB=~- defined for the host causes the host not to be considered.

Only if there are no hosts in the best available category of hosts, the next category is used. That is, no "good" matches will ever be used if there are "best" matches available. Moreover, if all "best" matches have been used up but are known to exist, the search fails.

"~" may not be used along with "*": "~*" combination will be silently ignored entirety, and will not modify the other specified affinities. Note that "~" alone has a meaning of 'anything but empty argument value, "". Also note that formally, "~A" is an equivalent to "~A *" as well as "~-" is an equivalent to "*".

Argument Matching Examples

Host affinity

DB=A ~B

makes the host to serve requests having either DB=A or DB=<other than B> in their query strings. The host may be used as a failover for requests that have DB=C in them (or no DB) if there is no better candidate available. Adding "!" to the affinity line would cause the host not to be used for any requests, in which the DB argument is missing.

Host affinity

DB=A -

makes the host to serve requests with either explicit DB=A in their query strings, or not having DB argument at all. Failovers from searches not matching the above may occur. Adding "!" to the line disables the failovers.

Host affinity

DB=- *

makes the host to serve requests that don't have any DB argument in their query strings, or when their DB argument failed to literally match affinity lines of all other hosts. Adding "!" to the line doesn't change the behavior.

Log File

The CAF module uses the Apache web server log files to put CAF module's messages into.

Monitoring

The status of the CAF modules can be seen via a web interface using the following links:

http://web1.be-md.ncbi.nlm.nih.gov/caf-status

http://web2.be-md.ncbi.nlm.nih.gov/caf-status

http://web3.be-md.ncbi.nlm.nih.gov/caf-status

http://web4.be-md.ncbi.nlm.nih.gov/caf-status

http://webdev1.be-md.ncbi.nlm.nih.gov/caf-status

http://webdev2.be-md.ncbi.nlm.nih.gov/caf-status

http://web91.be-md.qa.ncbi.nlm.nih.gov/caf-status

DISPD Network Dispatcher

Overview

The DISPD dispatcher is a CGI/1.0-compliant program (the actual file name is dispd.cgi). Its purpose is mapping a requested service name to an actual server location when the client has no direct access to the LBSMD daemon. This mapping is called dispatching. Optionally, the DISPD dispatcher can also pass data between the client, who requested the mapping, and the server, which implements the service, found as a result of dispatching. This combined mode is called a connection. The client may choose any of these modes if there are no special requirements on data transfer (e.g., firewall connection). In some cases, however, the requested connection mode implicitly limits the request to be a dispatching-only request, and the actual data flow between the client and the server occurs separately at a later stage.

Protocol Description

The dispatching protocol is designed as an extension of HTTP/1.0 and is coded in the HTTP header parts of packets. The request (both dispatching and connection) is done by sending an HTTP packet to the DISPD dispatcher with a query line of the form:

```
dispd.cgi?service=<name>
```

which can be followed by parameters (if applicable) to be passed to the service. The <name> defines the name of the service to be used. The other parameters take the form of one or more of the following constructs:

```
&<param>[=<value>]
```

where square brackets are used to denote an optional value part of the parameter.

In case of a connection request the request body can contain data to be passed to the first-found server. A connection to this server is automatically initiated by the DISPD dispatcher. On the contrary, in case of a dispatching-only request, the body is completely ignored, that is, the connection is dropped after the header has been read and then the reply is generated without consuming the body data. That process may confuse an unprepared client.

Mapping of a service name into a server address is done by the LBSMD daemon which is run on the same host where the DISPD dispatcher is run. The DISPD dispatcher never dispatches a non-local client to a server marked as local-only (by means of L=yes in the configuration of the LBSMD daemon). Otherwise, the result of dispatching is exactly what the client would get from the service mapping API if run locally. Specifying capabilities explicitly the client can narrow the server search, for example, by choosing stateless servers only.

Client Request to DISPD

The following additional HTTP tags are recognized in the client request to the DISPD dispatcher.

Tag	Description
Accepted-Server-Types: <list>	The <list> can include one or more of the following keywords separated by spaces: • NCBID • STANDALONE • HTTP • HTTP_GET • HTTP_POST • FIREWALL The keyword describes the server type which the client is capable to handle. The default is any (when the tag is not present in the HTTP header), and in case of a connection request, the dispatcher will accommodate an actual found server with the connection mode, which the client requested, by relaying data appropriately and in a way suitable for the server. Note: FIREWALL indicates that the client chooses a firewall method of communication. Note: Some server types can be ignored if not compatible with the current client mode

Client-Mode: <client-mode>	The <client-mode> can be one of the following: • STATELESS_ONLY - specifies that the client is not capable of doing full-duplex data exchange with the server in a session mode (e.g., in a dedicated connection). • STATEFUL_CAPABLE - should be used by the clients, which are capable of holding an opened connection to a server. This keyword serves as a hint to the dispatcher to try to open a direct TCP channel between the client and the server, thus reducing the network usage overhead. The default (when the tag is not present at all) is STATELESS_ONLY to support Web browsers.
Dispatch-Mode: <dispatch-mode>	The <dispatch-mode> can be one of the following: • INFORMATION_ONLY - specifies that the request is a dispatching request, and no data and/or connection establishment with the server is required at this stage, i.e., the DISPD dispatcher returns only a list of available server specifications (if any) corresponding to the requested service and in accordance with client mode and server acceptance. • NO_INFORMATION - is used to disable sending the above-mentioned dispatching information back to the client. This keyword is reserved solely for internal use by the DISPD dispatcher and should **not** be used by applications. • STATEFUL_INCLUSIVE - informs the DISPD dispatcher that the current request is a connection request, and because it is going over HTTP it is treated as stateless, thus dispatching would supply stateless servers only. This keyword modifies the default behavior, and dispatching information sent back along with the server reply (resulting from data exchange) should include stateful servers as well, allowing the client to go to a dedicated connection later. • OK_DOWN or OK_SUPPRESSED or PROMISCUOUS - defines a dispatch only request without actual data transfer for the client to obtain a list of servers which otherwise are not included such as, currently down servers (OK_DOWN), currently suppressed by having 100% penalty servers (OK_SUPPRESSED) or both (PROMISCUOUS) The default (in the absence of this tag) is a connection request, and because it is going over HTTP, it is automatically considered stateless. This is to support calls for NCBI services from Web browsers.
Skip-Info-<n>: <server-info>	<n> is a number of <server-info> strings that can be passed to the DISPD dispatcher to ignore the servers from being potential mapping targets (in case if the client knows that the listed servers either do not work or are not appropriate). Skip-Info tags are enumerated by numerical consequent suffices (<n>), starting from 1. These tags are optional and should only be used if the client believes that the certain servers do not match the search criteria, or otherwise the client may end up with an unsuccessful mapping.
Client-Host: <host>	The tag is used by the DISPD dispatcher internally to identify the <host>, where the request comes from, in case if relaying is involved. Although the DISPD dispatcher effectively disregards this tag if the request originates from outside NCBI (and thus it cannot be easily fooled by address spoofing), in-house applications **should** **not** use this tag when connecting to the DISPD dispatcher because the tag **is trusted and considered** within the NCBI Intranet.
Server-Count: {N|ALL}	The tag defines how many server infos to include per response (default N=3, ALL causes everything to be returned at once). N is an integer and ALL is a keyword.

DISPD Client Response

The DISPD dispatcher can produce the following HTTP tags in response to the client.

Tag	Description
Relay-Path: <path>	The tag shows how the information was passed along by the DISPD dispatcher and the NCBID utility. This is essential for debugging purposes
Server-Info-<n>: <server-info>	The tag(s) (enumerated increasingly by suffix <n>, starting from 1) give a list of servers, where the requested service is available. The list can have up to five entries. However, there is only one entry generated when the service was requested either in firewall mode or by a Web browser. For a non-local client, the returned server descriptors can include FIREWALL server specifications. Despite preserving information about host, port, type, and other (but not all) parameters of the original servers, FIREWALL descriptors are not specifications of real servers, but they are created on-the-fly by the DISPD dispatcher to indicate that the connection point of the server cannot be otherwise reached without the use of either firewalling or relaying.
Connection-Info: <host> <port> <ticket>	The tag is generated in a response to a stateful-capable client and includes a host (in a dotted notation) and a port number (decimal value) of the connection point where the server is listening (if either the server has specifically started or the FWDaemon created that connection point because of the client's request). The ticket value (hexadecimal) represents the 4-byte ticket that must be passed to the server as binary data at the very beginning of the stream. If instead of a host, a port, and ticket information there is a keyword TRY_STATELESS, then for some reasons (see Dispatcher-Failures tag below) the request failed but may succeed if the client would switch into a stateless mode.

Dispatcher-Failures: <failures>	The tag value lists all transient failures that the dispatcher might have experienced while processing the request. A fatal error (if any) always appears as the last failure in the list. In this case, the reply body would contain a copy of the message as well. Note: Fatal dispatching failure is also indicated by an unsuccessful HTTP completion code.
Used-Server-Info-n: <server_info>	The tag informs the client end of server infos that having been unsuccessfully used during current connection request (so that the client will be able to skip over them if needs to). n is an integral suffix, enumerating from 1.
Dispatcher-Messages:	The tag is used to issue a message into standard error log of a client. The message is intercepted and delivered from within Toolkit HTTP API.

Communication Schemes

After making a dispatching request and using the dispatching information returned, the client can usually connect to the server on its own. Sometimes, however, the client has to connect to the DISPD dispatcher again to proceed with communication with the server. For the DISPD dispatcher this would then be a connection request which can go one of two similar ways, relaying and firewalling.

The figures (Figure7, Figure8) provided at the very beginning of the "Load Balancing" chapter can be used for better understanding of the communication schemes described below.

- In the relay mode, the DISPD dispatcher passes data from the client to the server and back, playing the role of a middleman. Data relaying occurs when, for instance, a Web browser client wants to communicate with a service governed by the DISPD dispatcher itself.
- In the firewall mode, DISPD sends out only the information about where the client has to connect to communicate with the server. This connection point and a verifiable ticket are specified in the Connection-Info tag in the reply header. Note: firewalling actually pertains only to the stateful-capable clients and servers.

The firewall mode is selected by the presence of the FIREWALL keyword in the Accepted-Server-Types tag set by the client sitting behind a firewall and not being able to connect to an arbitrary port.

These are scenarios of data flow between the client and the server, depending on the "stateness" of the client:

A. Stateless client

1 Client is **not using firewall** mode

- **I** The client has to connect to the server by its own, using dispatching information obtained earlier; or
- **II** The client connects to the DISPD dispatcher with a connection request (e.g., the case of Web browsers) and the DISPD dispatcher facilitates data relaying for the client to the server.

2 If the client chooses to use the firewall mode then the only way to communicate with the server is to connect to the DISPD dispatcher (making a connection request) and use the DISPD dispatcher as a relay.

Note: Even if the server is stand-alone (but lacking S=yes in the configuration file of the LBSMD daemon) then the DISPD dispatcher initiates a microsession to the server and wraps its output into an HTTP/1.0-compliant reply. Data from both HTTP and NCBID servers are simply relayed one-to-one.

B. Stateful-capable client

1 A client which is **not using the firewall** mode has to connect directly to the server, using the dispatcher information obtained earlier (e.g., with the use of INFORMATION_ONLY in Dispatch-Mode tag) if local; for external clients the connection point is provided by the Connection-Info tag (port range 4444-4544).

2 If the firewall mode is selected, then the client has to expect Connection-Info to come back from the DISPD dispatcher pointing out where to connect to the server. If TRY_STATELESS comes out as a value of the former tag, then the client has to switch into a stateless mode (e.g., by setting STATELESS_ONLY in the Client-Mode tag) for the request to succeed.

Note: TRY_STATELESS could be induced by many reasons, mainly because all servers for the service are stateless ones or because the FWDaemon is not available on the host, where the client's request was received.

Note: Outlined scenarios show that no prior dispatching information is required for a stateless client to make a connection request, because the DISPD dispatcher can always be used as a data relay (in this way, Web browsers can access NCBI services). But for a stateful-capable client to establish a dedicated connection an additional step of obtaining dispatching information must precede the actual connection.

To support requests from Web browsers, which are unaware of HTTP extensions comprising dispatching protocol the DISPD dispatcher considers an incoming request that does not contain input dispatching tags as a connection request from a stateless-only client.

The DISPD dispatcher uses simple heuristics in analyzing an HTTP header to determine whether the connection request comes from a Web browser or from an application (a service connector, for instance). In case of a Web browser the chosen data path could be more expensive but more robust including connection retries if required, whereas on the contrary with an application, the dispatcher could return an error, and the retry is delegated to the application.

The DISPD dispatcher always preserves original HTTP tags User-Agent and Client-Platform when doing both relaying and firewalling.

NCBID Server Launcher

Overview

The LBSMD daemon supports services of type NCBID which are really UNIX filter programs that read data from the stdin stream and write the output into the stdout stream without having a common protocol. Thus, HTTP/1.0 was chosen as a framed protocol for wrapping both requests and replies, and the NCBID utility CGI program was created to pass a request from the HTTP body to the server and to put the reply from the server into the HTTP body and send it back to the client. The NCBID utility also provides a dedicated connection between the server and the client, if the client supports the stateful way of communication. Former releases of the NCBID utility were implemented as a separate CGI program however the latest releases integrated the NCBID utility and the DISPD dispatcher into a single component (ncbid.cgi is a hard link to dispd.cgi).

The NCBID utility determines the requested service from the query string in the same way as the DISPD dispatcher does, i.e., by looking into the value of the CGI parameter service. An executable file which has to be run is then obtained by searching the configuration file (shared with the LBSMD daemon; the default name is servrc.cfg): the path to the executable along with optional command-line parameters is specified after the bar character ("|") in the line containing a service definition.

The NCBID utility can work in either of two connection modes, stateless and stateful, as determined by reading the following HTTP header tag:

Connection-Mode: <mode>

where <mode> is one of the following:

- STATEFUL
- STATELESS

The default value (when the tag is missing) is STATELESS to support calls from Web browsers.

When the DISPD dispatcher relays data to the NCBID utility this tag is set in accordance with the current client mode.

The STATELESS mode is almost identical to a call of a conventional CGI program with an exception that the HTTP header could hold tags pertaining to the dispatching protocol, and resulting from data relaying (if any) by the DISPD dispatcher.

In the STATEFUL mode, the NCBID utility starts the program in a more tricky way, which is closer to working in a firewall mode for the DISPD dispatcher, i.e. the NCBID utility loads the program with its stdin and stdout bound to a port, which is switched to listening. The program becomes a sort of an Internet daemon (the only exception is that only one incoming connection is allowed). Then the client is sent back an HTTP reply containing the Connection-Info tag. The client has to use port, host, and ticket from that tag to connect to the server by creating a dedicated TCP connection.

Note: the NCBID utility never generates TRY_STATELESS keyword.

For the sake of the backward compatibility the NCBID utility creates the following environment variables (in addition to CGI/1.0 environment variables created by the HTTP daemon when calling NCBID) before starting the service executables:

Name	Description
NI_CLIENT_IPADDR	The variable contains an IP address of the remote host. It could also be an IP address of the firewall daemon if the NCBID utility was started as a result of firewalling.
NI_CLIENT_PLATFORM	The variable contains the client platform extracted from the HTTP tag Client-Platform provided by the client if any.

Firewall Daemon (FWDaemon)

Overview

The NCBI Firewall Daemon (FWDaemon) is essentially a network multiplexer listening at an advertised network address.

The FWDaemon works in a close cooperation with the DISPD dispatcher which informs FWDaemon on how to connect to the “real” NCBI server and then instructs the network client to connect to FWDaemon (instead of the “real” NCBI server). Thus, the FWDaemon serves as a middleman that just pumps the network traffic from the network client to the NCBI server and back.

The FWDaemon allows a network client to establish a persistent TCP/IP connection to any of publicly advertised NCBI services, provided that the client is allowed to make an outgoing network connection to any of the following FWDaemon addresses (on front-end NCBI machines):

```
ports 5860..5870 at both 130.14.29.112 and 165.112.7.12
```

Note: One FWDaemon can simultaneously serve many client/server pairs.

FWDaemon Behind a "Regular" Firewall

If a network client is behind a regular firewall, then a system administrator should open the above addresses (only!) for outgoing connections and set your client to "firewall" mode. Now the network client can use NCBI network services in a usual way (as if there were no firewall at all).

FWDaemon Behind a "Non-Transparent" Firewall

Note: If a firewall is "non-transparent" (this is an extremely rare case), then a system administrator must "map" the corresponding ports on your firewall server to the advertised FWDaemon addresses (shown above). In this case, you will have to specify the address of your firewall server in the client configuration.

The mapping on your non-transparent firewall server should be similar to the following:

```
CONN_PROXY_HOST:5860..5870 --> 130.14.29.112:5860..5870
```

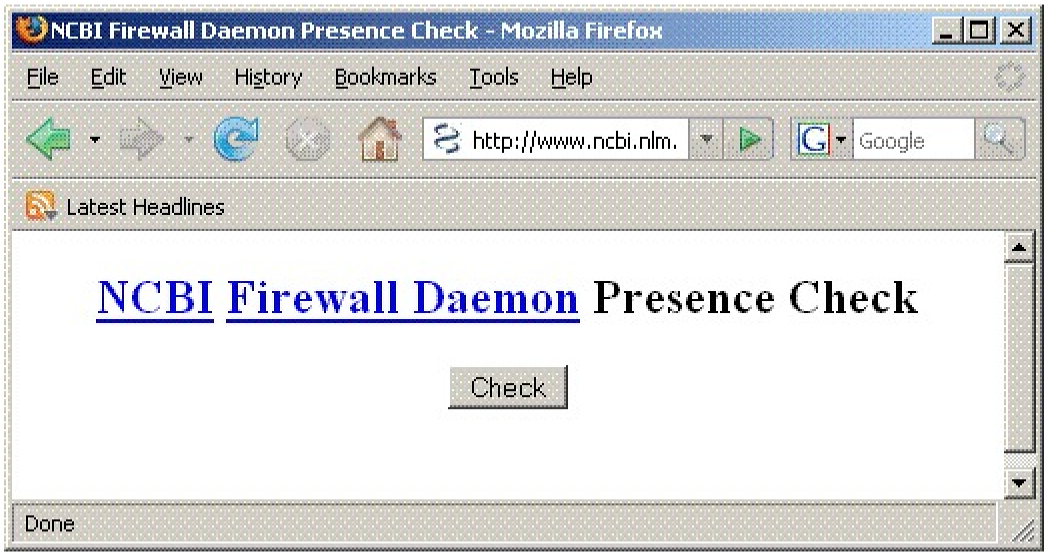
Please note that there is a port range that might not be presently used by any clients and servers, but it is reserved for future extensions. Nevertheless, it is recommended that you have this range configured on firewalls to allow the applications to function seamlessly in the future.

Monitoring

The FWDaemon could be monitored using the following web page:

http://www.ncbi.nlm.nih.gov/IEB/ToolBox/NETWORK/fwd_check.cgi

Having the page loaded into a browser the user will see the following.

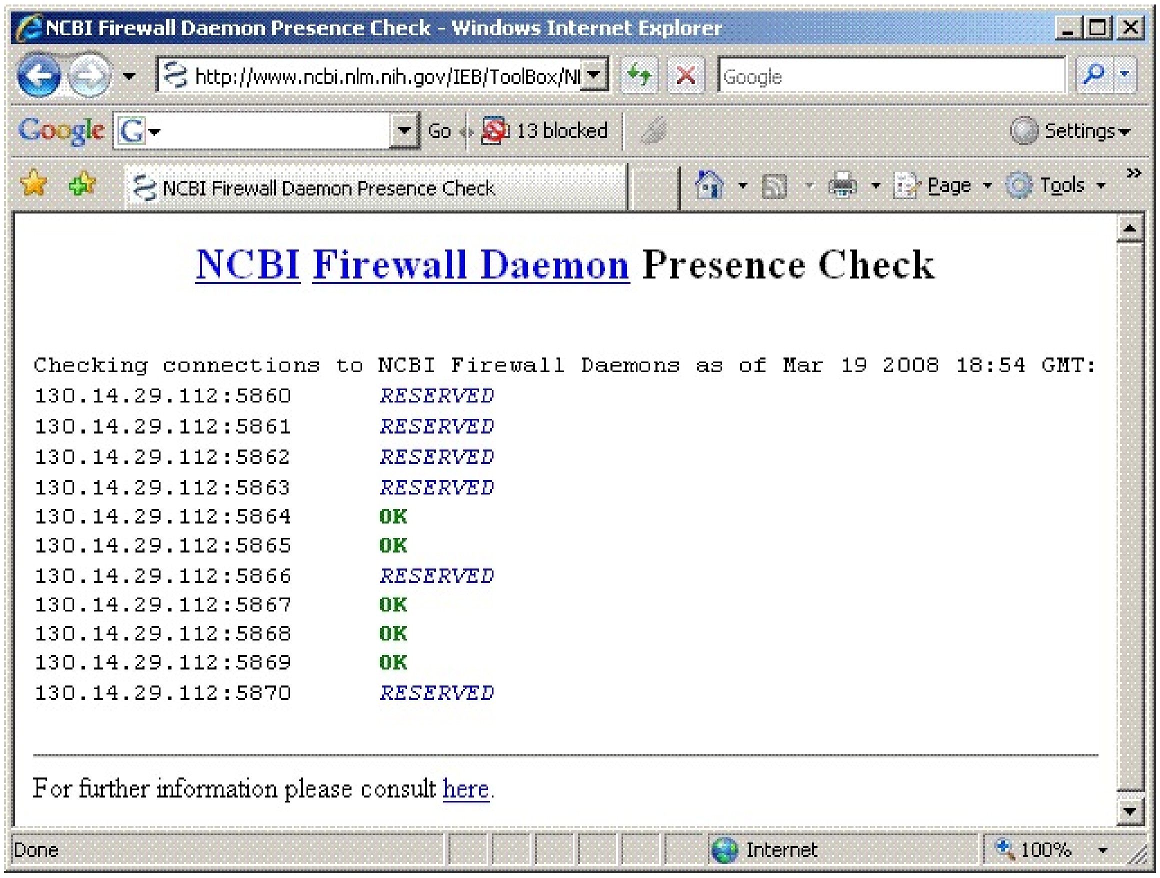

NCBI Firewall Daemon Presence Check - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.ncbi.nlm.
Google
Latest Headlines
Done

NCBI Firewall Daemon Presence Check

Check

Figure 15. FWDaemon Checking Web Page

By clicking the “Check” button a page similar to the following will appear.


NCBI Firewall Daemon Presence Check - Windows Internet Explorer
http://www.ncbi.nlm.nih.gov/IEB/ToolBox/NI
Google
Go
13 blocked
Settings
NCBI Firewall Daemon Presence Check
Page
Tools
Done
Internet
100%

NCBI Firewall Daemon Presence Check

```
Checking connections to NCBI Firewall Daemons as of Mar 19 2008 18:54 GMT:
130.14.29.112:5860      RESERVED
130.14.29.112:5861      RESERVED
130.14.29.112:5862      RESERVED
130.14.29.112:5863      RESERVED
130.14.29.112:5864      OK
130.14.29.112:5865      OK
130.14.29.112:5866      RESERVED
130.14.29.112:5867      OK
130.14.29.112:5868      OK
130.14.29.112:5869      OK
130.14.29.112:5870      RESERVED
```

For further information please consult here.

Figure 16. FWDaemon Presence Check

The outside NCBI network users can check the connection to the NAT service following the below steps:

- Run the FWDaemon presence check as described above.
- Take connection properties from any line where the status is “OK”. For example 130.14.29.112:5864
- Establish a telnet session using those connection properties. The example of a connection session is given below (a case when a connection was successfully established).

```
 > telnet 130.14.29.112 5864
Trying 130.14.29.112...
Connected to 130.14.29.112.
Escape character is '^]'.
NCBI Firewall Daemon: Invalid ticket. Connection closed.
See http://www.ncbi.nlm.nih.gov/cpp/network/firewall.html.
Connection closed by foreign host.
```

Log Files

The FWDaemon stores its log files at the following location:

/opt/machine/fwdaemon/log/fwdaemon

which is usually a link to /var/log/fwdaemon.

The file is formed locally on a host where FWDaemon is running.

FWDaemon and NCBID Server Data Exchange

One of the key points in the communications between the NCBID server and the FWDaemon is that the DISPD dispatcher instructs the FWDaemon to expect a new client connection. This instruction is issued as a reaction on a remote client request. It is possible that the remote client requested a service but did not use it. To prevent resource leaking and facilitate the usage monitoring the FWDaemon keeps a track of those requested but not used connections in a special file. The NCBID dispatcher is able to read that file before requesting a new connection from the FWDaemon and if the client was previously marked as the one who left connections not used then the NCBID dispatcher refuses the connection request.

The data exchange is illustrated on the figure below.

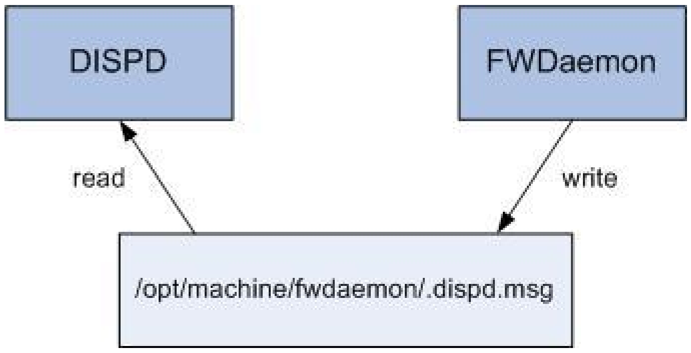


Figure 17. DISPD FWDaemon Data Exchange

The location of the .dispd.msg file is detected by the DISPD dispatcher as follows. The dispatcher determines the user name who owns the dispd.cgi executable. Then the dispatcher looks to the home directory for that user. The directory is used to look for the .dispd.msg file. The FWDaemon is run under the same user and the .dispd.msg file is saved by the daemon in its home directory.

Launcherd Utility

The purpose of the launcherd utility is to replace the NCBID services on hosts where there is no Apache server installed and there is a need to have UNIX filter programs to be daemonized.

The launcherd utility is implemented as a command line utility which is controlled by command line arguments. The list of accepted arguments can be retrieved with the -h option:

```
service1:~> /export/home/service/launcherd -h
Usage:
launcherd [-h] [-q] [-v] [-n] [-d] [-i] [-p #] [-l file] service command [parameters...]
-h = Print usage information only; ignore anything else
-q = Quiet start [and silent exit if already running]
```

```
-v = Verbose logging [terse otherwise]
-n = No statistics collection
-d = Debug mode [do not go daemon, stay foreground]
-i = Internal mode [bind to localhost only]
-p # = Port # to listen on for incoming connection requests
-l = Set log file name [use `-' or `+' to run w/o logger]
Note: Service must be of type STANDALONE to auto-get the port.
Note: Logging to `/dev/null' is treated as logging to a file.
Signals: HUP, INT, QUIT, TERM to exit
```

The launcherd utility accepts the name of the service to be daemonized. Using the service name the utility checks the LBSMD daemon table and retrieves port on which the service requests should be accepted. As soon as an incoming request is accepted the launched forks and connects the socket with the standard streams of the service executable.

One of the launcherd utility command line arguments is a path to a log file where the protocol messages are stored.

The common practice for the launcherd utility is to be run by the standard UNIX cron daemon. Here is an example of a cron schedule which runs the launcherd utility every 3 minutes:

```
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/export/home/service/UPGRADE/crontabs/service1/crontab
# installed on Thu Mar 20 20:48:02 2008)
# (Cron version -- $Id: crontab.c,v 2.13 1994/01/17 03:20:37 vixie Exp $)
MAILTO=ncbiduse@ncbi
*/3 * * * * test -x /export/home/service/launcherd && /export/home/service/launcherd -q -l /
export/home/service/bounce.log -- Bounce /export/home/service/bounce >/dev/null
MAILTO=grid-mon@ncbi,taxhelp@ncbi
*/3 * * * * test -x /export/home/service/launcherd && /export/home/service/launcherd -q -l /
var/log/taxservice -- TaxService /export /home/service/taxservice/taxservice >/dev/null
```

Monitoring Tools

There are various ways to monitor the services available at NCBI. These are generic third party tools and specific NCBI developed utilities. The specific utilities are described above in the sections related to a certain component.

The system load and statistics could be visualized by using ORCA graphs. It can be reached at:

http://viz.ncbi.nlm.nih.gov/orca/

The NCBI ORCA Pages shows all the available hosts.

Click on the certain server leads to another page with the related graphs.

One more web based tool to monitor servers / services statuses is Nagios. It can be reached at:

http://nagios.ncbi.nlm.nih.gov

Quality Assurance Domain

The quality assurance (QA) domain uses the same equipment and the same network as the production domain. Not all the services which are implemented in the production domain are implemented in the QA one. When a certain service is requested with the purpose of testing a service from QA should be called if it is implemented or a production one otherwise. The dispatching is implemented transparently. It is done by the CAF module running on production front ends. To implement that the CAFQAMap directive is put into the Apache web server configuration file as following:

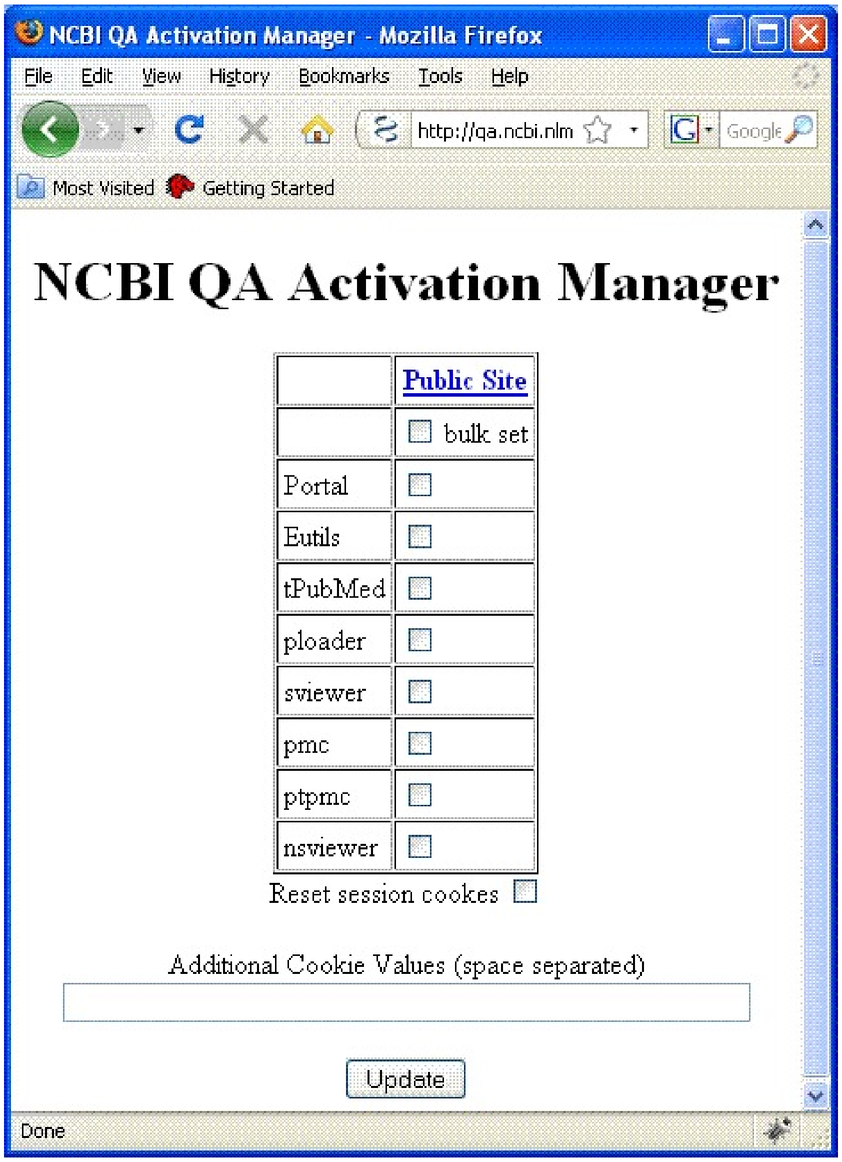
CAFQAMap NCBIQA /opt/machine/httpd/public/conf/ncbiqa.mapping

The directive above defines the NCBIQA cookie which triggers names substitutions found in the /opt/machine/httpd/public/conf/ncbiqa.mapping file.

To set the cookie the user can visit the following link:

http://qa.ncbi.nlm.nih.gov/portal/sysutils/qa_status.cgi

A screen similar to the following will appear:


NCBI QA Activation Manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://qa.ncbi.nlm
Google
Most Visited
Getting Started
Done

NCBI QA Activation Manager

	Public Site
	☐ bulk set
Portal	☐
Eutils	☐
tPubMed	☐
ploader	☐
sviewer	☐
pmc	☐
ptpmc	☐
nsviewer	☐

Reset session cookes ☐

Additional Cookie Values (space separated)

Update

Figure 18. QA Cookie Manager.

While connecting to a certain service the cookie is analyzed by the CAF module and if the QA cookie is detected then name mapping is triggered. The mapping is actually a process of replacing one name with another. The replacement rules are stored in the /opt/machine/httpd/public/conf/ncbiqa.mapping file. The file content could be similar to the following:

portal portalqa

eutils eutilsqa

tpubmed tpubmedqa

which means to replace portal with portalqa etc.

So the further processing of the request is done using the substituted name. The process is illustrated on the figure below.

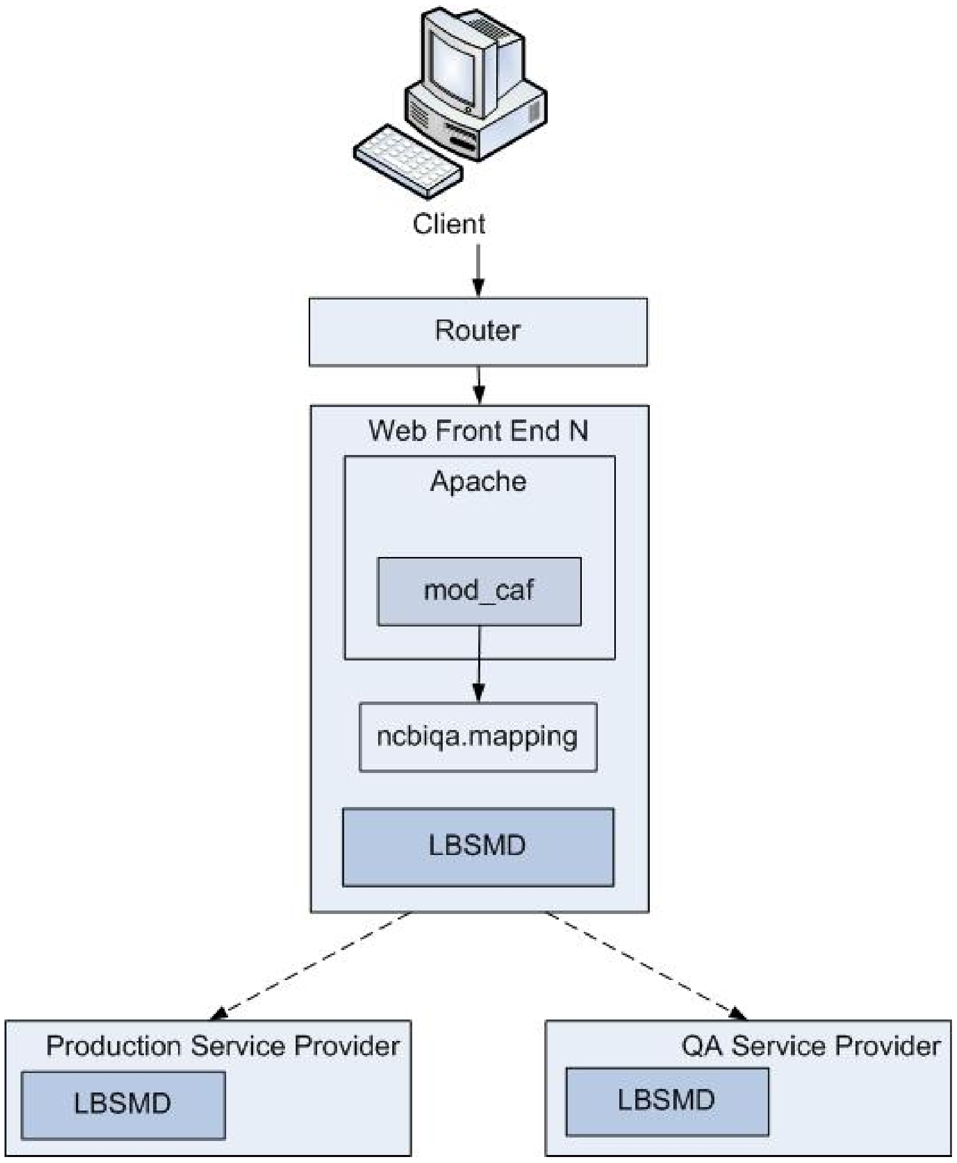


Figure 19. NCBI QA

NCBI Genome Workbench

The NCBI Genome Workbench is an integrated sequence visualization and analysis platform. This application runs on Windows, Unix, and Macintosh OS X.

The following topics are discussed in this section:

- Design goals
- Design

Design Goals

The primary goal of Genome Workbench is to provide a flexible platform for development of new analytic and visualization techniques. To this end, the application must facilitate easy modification and extension. In addition, we place a large emphasis on cross-platform development, and Genome Workbench should function and appear identically on all supported platforms.

Design

The basic design of Genome Workbench follows a modified Model-View-Controller (MVC) architecture. The MVC paradigm provides a clean separation between the data being dealt with (the model), the user's perception of this data (provided in views), and the user's interaction with this data (implemented in controllers). For Genome Workbench, as with many other implementations of the MVC architecture, the View and Controller are generally combined.

Central to the framework is the notion of the data being modeled. The model here encompasses the NCBI data model, with particular emphasis on sequences and annotations. The Genome Workbench framework provides a central repository for all managed data through the static class interface in CDocManager. CDocManager owns the single instance of the C++ Object Manager that is maintained by the application. CDocManager marshals individual CDocument classes to deal with data as the user requests. CDocument, at its core, wraps a CScope class and thus provides a hook to the object manager.

The View/Controller aspect of the architecture is implemented through the abstract class CView. Each CView class is bound to a single document. Each CView class, in turn, represents a view of some portion of the data model or a derived object related to the document. This definition is intentionally vague; for example, when viewing a document that represents a sequence alignment, a sequence in that alignment may not be contained in the document itself but is distinctly related to the alignment and can be presented in the context of the document. In general, the views that use the framework will define a top-level FLTK window; however, a view could be defined to be a CGI context such that its graphical component is a Web browser.

To permit maximal extensibility, the framework delegates much of the function of creating and presenting views and analyses to a series of plugins. In fact, most of the basic components of the application itself are implemented as plugins. The Genome Workbench framework defines three classes of plugins: data loaders, views, and algorithms. Technically, a plugin is simply a shared library defining a standard entry point. These libraries are loaded on demand; the entry point returns a list of plugin factories, which are responsible for creating the actual plugin instances.

Cross-platform graphical development presents many challenges to proper encapsulation. To alleviate a lot of the difficulties seen with such development, we use a cross-platform GUI toolkit (FLTK) in combination with OpenGL for graphical development.

NCBI NetCache Service

What is NetCache?

NetCache is a service that provides to distributed hosts a reliable and uniform means of accessing temporary storage. Using NetCache, distributed applications can store data temporarily without having to manage distributed access or handle errors. Applications on different hosts can access the same data simply by using the unique key for the data.

CGI applications badly need this functionality to store session information between successive HTPP requests. Some session information could be embedded into URLs or cookies, however it is generally not a good idea because:

- Some data should not be transmitted to the client, for security reasons.
- Both URLs and cookies are quite limited in size.
- Passing data via either cookie or URL generally requires additional encoding and decoding steps.
- It makes little sense to pass data to the client only so it can be passed back to the server.

Thus it is better to store this information on the server side. However, this information cannot be stored locally because successive HTTP requests for a given session are often processed on different machines. One possible way to handle this is to create a file in a shared network directory. But this approach can present problems to client applications in any of the standard operations:

- Adding a blob
- Removing a blob
- Updating a blob
- Automatically removing expired blobs
- Automatically recovering after failures

Therefore, it's better to provide a centralized service that provides robust temporary storage, which is exactly what NetCache does.

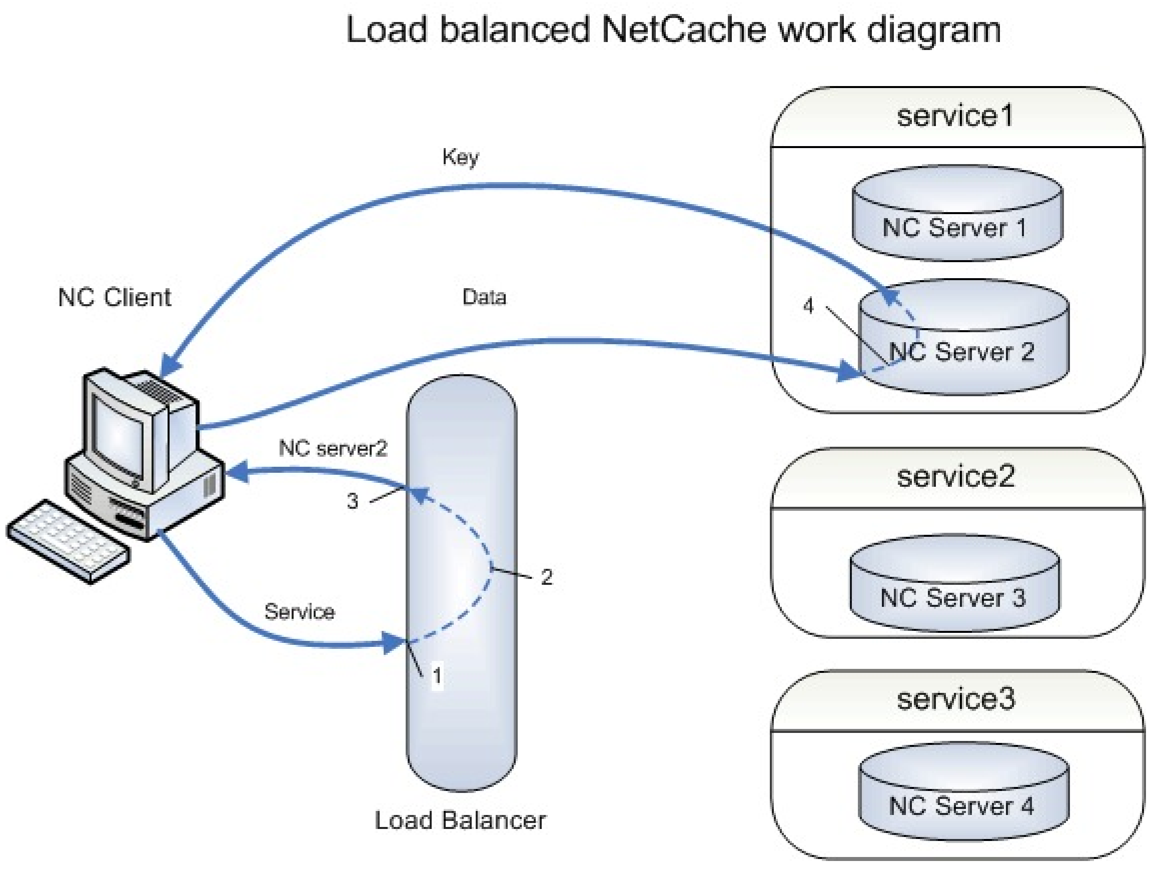
NetCache is load-balanced and has high performance and virtually unlimited scalability. Any Linux, UNIX or Windows machine can be a NetCache host, and any application can use it. For example, the success with which NetCache solves the problem of distributed access to temporary storage enables the NCBI Grid framework to rely on it for passing data between its components.

What can NetCache be used for?

Programs can use NetCache for data exchange. For example, one application can put a blob into NetCache and pass the blob key to another application, which can then access (retrieve, update, remove) the data. Some typical use cases are:

- Store CGI session info
- Store CGI-generated graphics
- Cache results of computations
- Cache results of expensive DBMS or search system queries
- Pass messages between programs

The diagram below illustrates how NetCache works.



1 Client requests a named service from the Load Balancer.

2 Load Balancer chooses the least loaded server (on this diagram Server 2) corresponding to the requested service.

3 Load Balancer returns the chosen server to the client.

4 Client connects to the selected NetCache server and sends the data to store.

5 NetCache generates and returns a unique key which can then be used to access the data.

How to use NetCache

All new applications developed within NCBI should use NetCache together with the NCBI Load Balancer. It is not recommended to use an unbalanced NetCache service.

The following topics explain how to use NetCache from an application:

- The basic ideas
- Set up your program to use NetCache
- Establish the NetCache service name
- Initialize the client API
- Store data
- Retrieve data
- Samples and other resources

The basic ideas

Two classes provide an interface to NetCache - CNetCacheAPI and CNetICacheClient. These classes share most of the basic ideas of using NetCache, but might be best suited for slightly different purposes. CNetCacheAPI might be a bit better for temporary storage in scenarios where the data is not kept elsewhere, whereas CNetICacheClient implements the ICache interface and might be a bit better for scenarios where the data still exists elsewhere but is also cached for performance reasons. CNetCacheAPI will probably be more commonly used because it automatically generates unique keys for you and it has a slightly simpler interface. CNetCacheAPI also supports stream insertion and extraction operators.

There are multiple ways to write data to NetCache and read it back, but the basic ideas are:

- NetCache stores data in blobs. There are no constraints on the format, and the size can be anything from one byte to "big" - that is, the size is specified using size_t and the practical size limit is the lesser of available storage and organizational policy.
- Blob identification is usually associated with a unique purpose.
  - With CNetCacheAPI, a blob is uniquely identified by a key that is generated by NetCache and returned to the calling code. Thus, the calling code can limit use of the blob to a given purpose. For example, data can be passed from one instance of a CGI to the next by storing the data in a NetCache blob and passing the key via cookie.
  - With CNetICacheClient, blobs are identified by the combination { key, version, subkey, cache name }, which isn't guaranteed to be unique. It is possible that two programs could choose the same combination and one program could change or delete the data stored by the other.
- With CNetICacheClient, the cache name can be specified in the registry and is essentially a convenient way of simulating namespaces.
- When new data is written using a key that corresponds to existing data:
  - API calls that use a buffer pointer replace the existing data.
  - API calls that use a stream or writer append to the existing data.
- Data written with a stream or writer won't be accessible from the NetCache server until the stream or writer is deleted or until the writer's Close() method is called.

- A key must be supplied to retrieve data.
- Reading data doesn't delete it - it will be removed automatically when its "time-to-live" has expired, or it can be removed explicitly.
- When you create a blob, you must either pass a non-zero time-to-live (TTL) value or accept the server default. If you don't pass a value, you can find the server default by calling GetBlobInfo() for a blob. Unless TTL prolongation is disabled for the NetCache server, the TTL gets reset each time a blob is accessed - either to the value you pass or the server default, not to the blob's former TTL. Therefore, if you specify a TTL, you must do so every time you access the blob, otherwise the blob will be given the server's default TTL.

Set up your program to use NetCache

To use NetCache from your application, you must use the NCBI application framework by deriving you application class from CNcbiApplication. If your application is a CGI, you can derive from CCgiApplication.

You will need at least the following libraries in your Makefile.<appname>.app:

```
# For CNcbiApplication-derived programs:
LIB = xconnserv xthrserv xconnect xutil xncbi

# For CCgiApplication-derived programs:
LIB = xcgi xconnserv xthrserv xconnect xutil xncbi

# If you're using CNetICacheClient, also add ncbi_xcache_netcache to LIB.

# All apps need this LIBS line:
LIBS = $(NETWORK_LIBS) $(DL_LIBS) $(ORIG_LIBS)
```

Your source should include:

```
#include <corelib/ncbiapp.hpp> // for CNcbiApplication-derived programs
#include <cgi/cgiapp.hpp> // for CCgiApplication-derived programs

#include <connect/services/netcache_api.hpp> // if you use CNetCacheAPI
#include <connect/services/neticache_client.hpp> // if you use
CNetICacheClient
```

An even easier way to get a new CGI application started is to use the new_project script:

```
new_project mycgi app/netcache
```

Establish the NetCache service name

All applications using NetCache must use a service name. A service name is essentially just an alias for a group of NetCache servers from which the load balancer can choose when connecting the NetCache client and server. For applications with minimal resource requirements, the selected service may be relatively unimportant, but applications with large resource requirements may need their own dedicated NetCache servers. But in all cases, developers should contact grid-core@ncbi.nlm.nih.gov and ask what service name to use for new applications.

Service names must match the pattern [A-Za-z_][A-Za-z0-9_]*, must not end in _lb, and are not case-sensitive. Limiting the length to 18 characters is recommended, but there is no hard limit.

Service names are typically specified on the command line or stored in the application configuration file. For example:

```
[netcache_api]
service=the_svc_name_here
```

Initialize the client API

Initializing the NetCache API is extremely easy - simply create a CNetCacheAPI or CNetICacheClient object, selecting the constructor that automatically configures the API based on the application registry. Then, define the client name in the application registry using the client entry in the [netcache_api] section. The client name should be unique if the data is application-specific, or it can be shared by two or more applications that need to access the same data. The client name is added to AppLog entries, so it is helpful to indicate the application in this string.

For example, put this in your source code:

```
// To configure automatically based on the config file, using CNetCacheAPI:
CNetCacheAPI nc_api(GetConfig());

// To configure automatically based on the config file, using
CNetICacheClient:
CNetICacheClient ic_client(CNetICacheClient::eAppRegistry);
```

and put this in your configuration file:

```
[netcache_api]
client=your_app_name_here
```

If you are using CNetICacheClient, you either need to use API methods that take a cache name or, to take advantage of automatic configuration based on the registry, specify a cache name in the [netcache_api] section, for example:

```
[netcache_api]
cache_name=your_cache_name_here
```

For a complete reference of NetCache configuration parameters, please see the NetCache and NetSchedule section in the Library Configuration chapter:

Store data

There are ancillary multiple ways to save data, whether you're using CNetCacheAPI or CNetICacheClient.

With all the storage methods, you can supply a "time-to-live" parameter, which specifies how long (in seconds) a blob will be accessible. See the basic ideas section for more information on time-to-live.

Storing data using CNetCacheAPI

If you are saving a new blob using CNetCacheAPI, it will create a unique blob key and pass it back to you. Here are several ways to store data using CNetCacheAPI (see the class reference for additional methods):

```
CNetCacheAPI nc_api(GetConfig());

// Write a simple object (and get the new blob key).
key = nc_api.PutData(message.c_str(), message.size());

// Or, overwrite the data by writing to the same key.
nc_api.PutData(key, message.c_str(), message.size());

// Or, create an ostream (and get a key), then insert into the stream.
auto_ptr<CNcbiOstream> os(nc_api.CreateOStream(key));
*os << "line one\n";
*os << "line two\n";
// (data written at stream deletion or os.reset())

// Or, create a writer (and get a key), then write data in chunks.
auto_ptr<IEmbeddedStreamWriter> writer(nc_api.PutData(&key));
while(...) {
 writer->Write(chunk_buf, chunk_size);
 // (data written at writer deletion or writer.Close())
```

Storing data using CNetICacheClient

If you are saving a new blob using CNetICacheClient, you must supply a unique { blob key / version / subkey / cache name } combination. Here are two ways (with the cache name coming from the registry) to store data using CNetICacheClient (see the class reference for additional methods):

```
CNetICacheClient ic_client(CNetICacheClient::eAppRegistry);

// Write a simple object.
ic_client.Store(key, version, subkey, message.c_str(), message.size());

// Or, create a writer, then write data in chunks.
auto_ptr<IEmbeddedStreamWriter>
 writer(ic_client.GetNetCacheWriter(key, version, subkey));
while(...) {
 writer->Write(chunk_buf, chunk_size);
 // (data written at writer deletion or writer.Close())
```

Retrieve data

Retrieving data is more or less complementary to storing data.

If an attempt is made to retrieve a blob after its time-to-live has expired, an exception will be thrown.

Retrieving data using CNetCacheAPI

The following code snippet demonstrates three ways of retrieving data using CNetCacheAPI (see the class reference for additional methods):

```
// Read a simple object.
nc_api.ReadData(key, message);

// Or, extract words from a stream.
auto_ptr<CNcbiIstream> is(nc_api.GetIStream(key));
while (!is->eof()) {
 *is >> message; // get one word at a time, ignoring whitespace

// Or, retrieve the whole stream buffer.
NcbiCout << "Read: '" << is->rdbuf() << "'" << NcbiEndl;

// Or, read data in chunks.
while (...) {
 ERW_Result rw_res = reader->Read(chunk_buf, chunk_size, &bytes_read);
 chunk_buf[bytes_read] = '\0';
 if (rw_res == eRW_Success) {
 NcbiCout << "Read: '" << chunk_buf << "'" << NcbiEndl;
 } else {
 NCBI_USER_THROW("Error while reading BLOB");
 }
```

Retrieving data using CNetICacheClient

The following code snippet demonstrates two ways to retrieve data using CNetICacheClient, with the cache name coming from the registry (see the class reference for additional methods):

```
// Read a simple object.
ic_client.Read(key, version, subkey, chunk_buf, kMyBufSize);

// Or, read data in chunks.
size_t remaining(ic_client.GetSize(key, version, subkey));
auto_ptr<IReader> reader(ic_client.GetReadStream(key, version, subkey));
while (remaining > 0) {
 size_t bytes_read;
 ERW_Result rw_res = reader->Read(chunk_buf, chunk_size, &bytes_read);
 if (rw_res != eRW_Success) {
 NCBI_USER_THROW("Error while reading BLOB");
 }
 // do something with the data
 ...
 remaining -= bytes_read;
}
```

Samples and other resources

Here is a sample client application that demonstrates a variety of ways to use NetCache:

src/sample/app/netcache/netcache_client_sample.cpp

Here is a sample application that uses NetCache from a CGI application:

src/sample/app/netcache/netcache_cgi_sample.cpp

Here are test applications for CNetCacheAPI and CNetICacheClient:

src/connect/services/test/test_netcache_api.cpp

src/connect/services/test/test_ic_client.cpp

Please see the NetCache and NetSchedule section of the Library Configuration chapter for documentation on the NetCache configuration parameters.

The grid_cli command-line tool (available on both Windows and UNIX) provides convenient sub-commands for manipulating blobs, getting their status, checking servers, etc.

You can also email grid-core@ncbi.nlm.nih.gov if you have questions.

Questions and answers

Q:What exactly is netcache's architecture, it is memory-based (like memcached), or does it use filesystem/sql/whatever?

A:It keeps its database on disk, memory-mapped; it also has a (configurable) "write-back buffer" - to use when there is a lot of data coming in, and a lot of this data gets re-written quickly (this is to help avoid thrashing the disk with relatively transient blob versions - when the OS's automatic memory swap mechanism may become sub-optimal).

Q:Is there an NCBI "pool" of netcache servers that we can simply tie in to, or do we have to set up netcache servers on our group's own machines?

A:We usually (except for PubMed) administer NC servers, most of which are shared. Depends on your load (hit rate, blob size distribution, blob lifetime, redundancy, etc.) we can point you to the shared NC servers or create a new NC server pool.

Q:I assume what's in c++/include/connect/services/*hpp is the api to use for a client?

A:Yes, also try the sample under src/sample/app/netcache - for example:

```
new_project pc_nc_client app/netcache
cd pc_nc_client
make
./netcache_client_sample -service NC_test
```

Q:Is there a way to build in some redundancy, e.g. so that if an individual server/host goes down, we don't lose data?

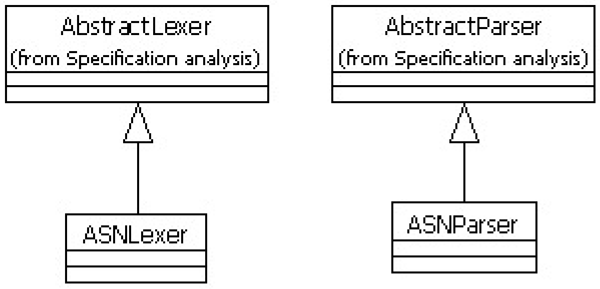
A:Yes, you can mirror NC servers, master-master style, including between BETH and COLO sites. Many NC users use mirrored instances nowadays, including PubMed.

Q:Is there a limit to the size of the data blobs that can be stored?

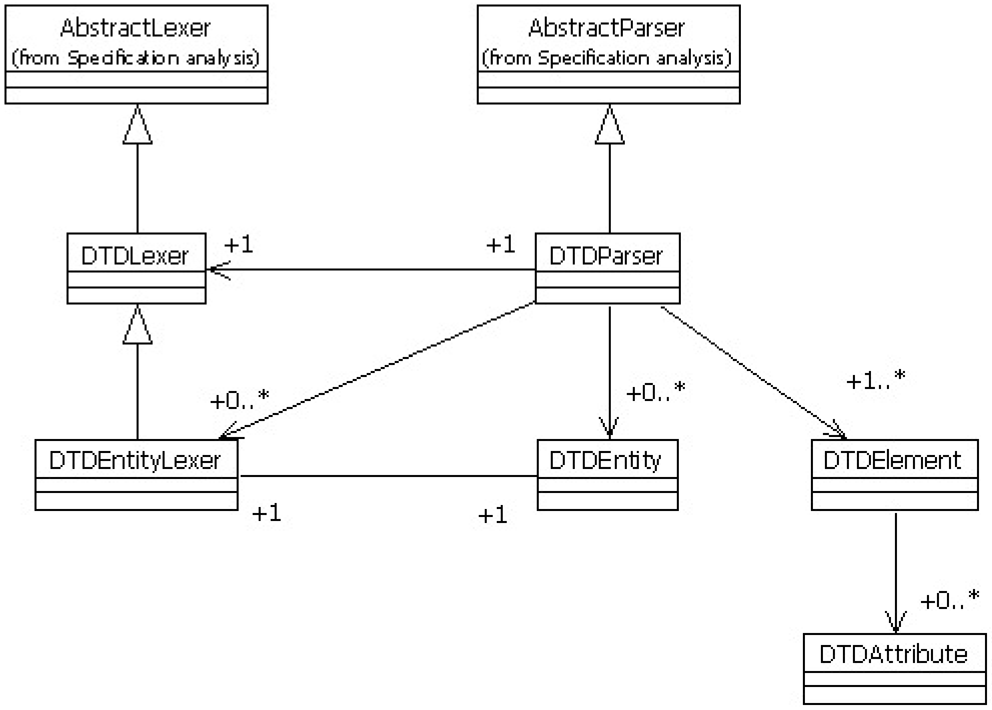
A:I have seen 400MB blobs there being written and read without an incident a thousand times a day (http://mini.ncbi.nlm.nih.gov/1k3ru). We can do experiments to see how your load will be handled. As a general rule, you should ask grid-core@ncbi.nlm.nih.gov for guidance when changing your NC usage.

Q:How is the expiration of BLOBs handled by NetCache? My thinking is coming from two directions. First, I wouldn’t want BLOBs deleted out from under me, but also, if the expiration is too long, I don’t want to be littering the NetCache. That is: do I need to work hard to remove all of my BLOBs or can I just trust the automatic clean-up?

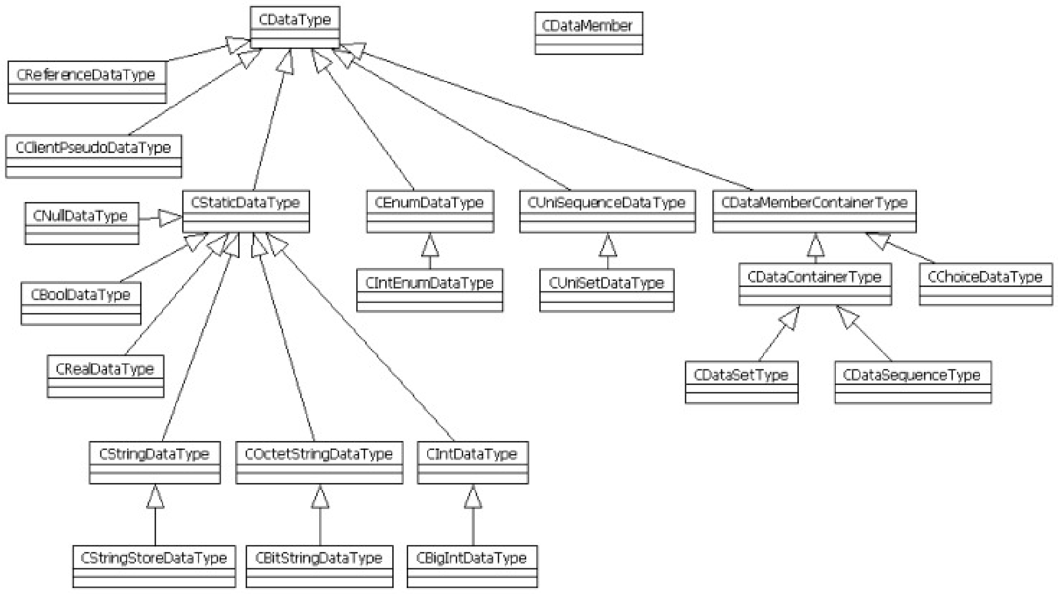
A:You can specify a "time-to-live" when you create a blob. If you don't specify a value, you can find the service's default value by calling GetBlobInfo(). See the basic ideas section for more details.



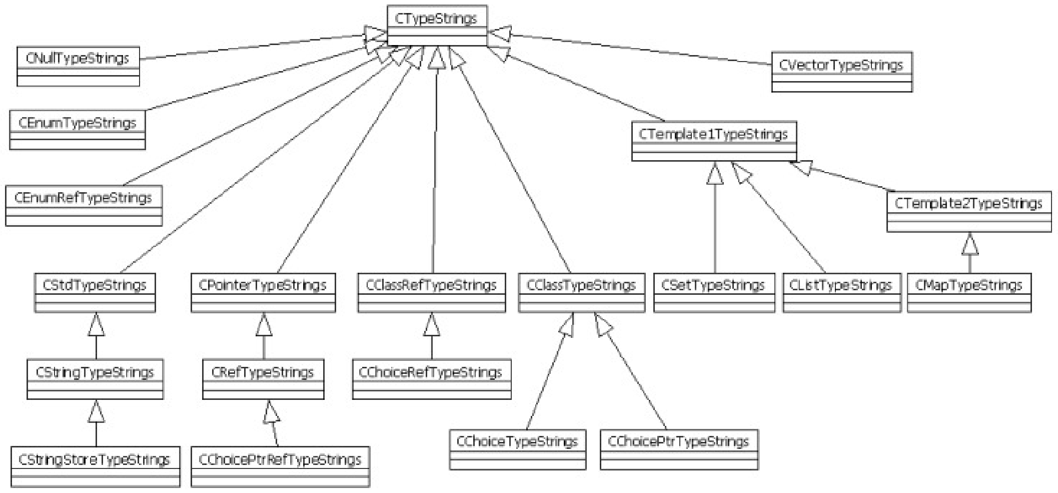
1. ASN.1 specification analysis.



2. DTD specification analysis.



3. Data values.



4. Code generation.

1. Main arguments

Argument	Effect	Comments
-h	Display the DATATOOL arguments	Ignores other arguments
-m <file>	ASN.1 or DTD module file(s)	Required argument
-M <file>	External module file(s)	Is used for IMPORT type resolution
-i	Ignore unresolved types	Is used for IMPORT type resolution
-f <file>	Write ASN.1 module file	
-fx <file>	Write DTD module file	"-fx m" writes modular DTD file
-fxs <file>	Write XML Schema file	
-fd <file>	Write specification dump file in datatool internal format	
-ms <string>	Suffix of modular DTD or XML Schema file name	
-dn <string>	DTD module name in XML header	No extension. If empty, omit DOCTYPE declaration.
-v <file>	Read value in ASN.1 text format	
-vx <file>	Read value in XML format	
-F	Read value completely into memory	
-p <file>	Write value in ASN.1 text format	
-px <file>	Write value in XML format	
-pj <file>	Write value in JSON format	
-d <file>	Read value in ASN.1 binary format	-t argument required
-t <type>	Binary value type name	See -d argument
-e <file>	Write value in ASN.1 binary format	
-xmlns	XML namespace name	When specified, also makes XML output file reference Schema instead of DTD
-sxo	No scope prefixes in XML output	
-sxi	No scope prefixes in XML input	
-logfile <File_Out>	File to which the program log should be redirected	
conffile <File_In>	Program's configuration (registry) data file	
-version	Print version number	Ignores other arguments

2. Code generation arguments

Argument	Effect	Comments
-od <file>	C++ code definition file	See Definition file
-ods	Generate an example definition file (e.g. MyModuleName._sample_def)	Must be used with another option that generates code such as -oA.
-odi	Ignore absent code definition file	
-odw	Issue a warning about absent code definition file	
-oA	Generate C++ files for all types	Only types from the main module are used (see -m and -mx arguments).
-ot <types>	Generate C++ files for listed types	Only types from the main module are used (see -m and -mx arguments).
-ox <types>	Exclude types from generation	
-oX	Turn off recursive type generation	
-of <file>	Write the list of generated C++ files	
-oc <file>	Write combining C++ files	
-on <string>	Default namespace	The value "-" in the Definition file means don't use a namespace at all and overrides the -on option specified elsewhere.
-opm <dir>	Directory for searching source modules	
-oph <dir>	Directory for generated *.hpp files	
-opc <dir>	Directory for generated *.cpp files	
-or <prefix>	Add prefix to generated file names	
-orq	Use quoted syntax form for generated include files	
-ors	Add source file dir to generated file names	
-orm	Add module name to generated file names	
-orA	Combine all -or* prefixes	
-ocvs	create ".cvsignore" files	
-oR <dir>	Set -op* and -or* arguments for NCBI directory tree	
-oDc	Turn ON generation of Doxygen-style comments	The value "-" in the Definition file means don't generate Doxygen comments and overrides the -oDc option specified elsewhere.
-odx <string>	URL of documentation root folder	For Doxygen
-lax_syntax	Allow non-standard ASN.1 syntax accepted by asntool	The value "-" in the Definition file means don't allow non-standard syntax and overrides the -lax_syntax option specified elsewhere.
-pch <string>	Name of the precompiled header file to include in all *.cpp files	
-oex <export>	Add storage-class modifier to generated classes	Can be overriden by [-]._export in the definition file.

25: Examples and Demos

Last Update: July 12, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

See Getting Started for basic information on using the NCBI C++ Toolkit.

Chapter Outline

ID1_FETCH - the ID1 and Entrez2 client

Location: c++/src/app/id1_fetch/id1_fetch.cpp (compiled executable is $NCBI/c++/Release/bin/id1_fetch on NCBI systems)

Synopsis:

- Accept a list of sequences, specified either directly by ID or indirectly by an Entrez query.
- Produce more information about the sequences, either as data from the ID server or as Entrez document summaries.

This program corresponds to idfetch from the C Toolkit.

Invocation

See Table 1.

Note: You must specify exactly one of the options indicating what to look up: -gi, -in, -flat, -fasta, -query, -qf.

Output Data Formats

The possible values of the -fmt argument are shown in Table 2.

Lookup Types

The possible values of the -lt argument are shown in Table 3.

Output Complexity Levels

The possible values of the -maxplex argument are shown in Table 4.

Flattened Sequence ID Format

A flattened sequence ID has one of the following three formats, where square brackets [...] surround optional elements:

- type([name or locus][,[accession][,[release][,version]]])
- type=accession[.version]
- type:number

The first format requires quotes in most shells.

The type is a number, indicating who assigned the ID, as follows:

- Local use
- GenInfo backbone sequence ID
- GenInfo backbone molecule type
- GenInfo import ID
- GenBank
- The European Molecular Biology Laboratory (EMBL)
- The Protein Information Resource (PIR)

- SWISS-PROT
- A patent
- RefSeq
- General database reference
- GenInfo integrated database (GI)
- The DNA Data Bank of Japan (DDBJ)
- The Protein Research Foundation (PRF)
- The Protein DataBase (PDB)
- Third-party annotation to GenBank
- Third-party annotation to EMBL
- Third-party annotation to DDBJ
- TrEMBL

FASTA Sequence ID Format

This format consists of a two- or three-letter tag indicating the ID's type, followed by one or more data fields, which are separated from the tag and each other by vertical bars (|). The vertical bar is a special character in most command-line shells, so command-line arguments containing ID's usually must be quoted. Table 5 shows the specific possibilities.

Example Usage

```
id1_fetch -query '5-delta4 isomerase' -lt none -db Nucleotide

34

id1_fetch -fmt genbank -gi 34

LOCUS BT3BHSD 1632 bp mRNA MAM 12-SEP-1993
DEFINITION Bovine mRNA for 3 beta hydroxy-5-ene steroid dehydrogenase/delta
 5-delta4 isomerase (EC 1.1.1.145, EC 5.3.3.1).
ACCESSION X17614
VERSION X17614.1 GI:34
KEYWORDS 3 beta-hydroxy-delta5-steroid dehydrogenase;
 steroid delta-isomerase.
...
FEATURES Location/Qualifiers
...
 CDS 105..1226
 /codon_start=1
 /transl_table=1
 /function="3 beta-HSD (AA 1-373)"
 /protein_id="CAA35615.1"
 /db_xref="GI:35"
 /translation="MAGWSCLVTGGGGFLGQRIICLLVEEKDLQEIRVLDKVFRPEVR
 EEFSKLQSKIKLTLLEGDILDEQCLKGACQGTSVVIHTASVIDVRNAVPRETIMNVNV
 KGTQLLLEACVQASVPVFIHTSTIEVAGPNSYREIIQDGREEEHHESAWSSPYPYSKK
 LAEKAVLGANGWALKNGGTLYTCALRPMYIYGEGSPFLSAYMHGALNNNGILTNHCKF
 SRVNPVYVGNVAWAHILALRALRDPKKVPNIQGQFYYISDDTPHQSYDDLNYTLSKEW
 GFCLDSRMSLPISLQYWLAFLLEIVSFLLSPIYKYNPCFNRHLVTLSNSVFTFSYKKA
 QRDLGYEPLYTWEEAKQKTKEWIGSLVKQHKETLKTKIH"
 /db_xref="SWISS-PROT:P14893"
...
 1441 ggacagacaa ggtgatttgc tgcagctgct ggcaccaaaa tctcagtggc agattctgag
 1501 ttatttgggc ttcttgtaac ttcgagtttt gcctcttagt cccactttct ttgttaaatg
 1561 tggaagcatt tcttttaaaa gttcatattc cttcatgtag ctcaataaaa atgatcaaca
 1621 ttttcatgac tc
//

id1_fetch -fmt genpept -gi 35

LOCUS CAA35615 373 aa MAM 12-SEP-1993
DEFINITION Bovine mRNA for 3 beta hydroxy-5-ene steroid dehydrogenase/delta
 5-delta4 isomerase (EC 1.1.1.145, EC 5.3.3.1), and translated
 products.
ACCESSION CAA35615
VERSION CAA35615.1 GI:35
PID g35
SOURCE cow.
 ORGANISM Bos taurus
 Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
 Mammalia; Eutheria; Cetartiodactyla; Ruminantia; Pecora; Bovoidea;
 Bovidae; Bovinae; Bos.
...
ORIGIN
 1 magwsclvtg gggflgqrii cllveekdlq eirvldkvfr pevreefskl qskikltlle
 61 gdildeqclk gacqgtsvvi htasvidvrn avpretimnv nvkgtqllle acvqasvpvf
 121 ihtstievag pnsyreiiqd greeehhesa wsspypyskk laekavlgan gwalknggtl
 181 ytcalrpmyi ygegspflsa ymhgalnnng iltnhckfsr vnpvyvgnva wahilalral
 241 rdpkkvpniq gqfyyisddt phqsyddlny tlskewgfcl dsrmslpisl qywlafllei
 301 vsfllspiyk ynpcfnrhlv tlsnsvftfs ykkaqrdlgy eplytweeak qktkewigsl
 361 vkqhketlkt kih
//

id1_fetch -fmt fasta -gi 35 -maxplex bioseq

>emb|CAA35615.1||gi|35 Bovine mRNA for 3 beta hydroxy-5-ene steroid
dehydrogenase/delta
 5-delta4 isomerase (EC 1.1.1.145, EC 5.3.3.1), and translated products
MAGWSCLVTGGGGFLGQRIICLLVEEKDLQEIRVLDKVFRPEVREEFSKLQSKIKLTLLEGDILDEQCLK
GACQGTSVVIHTASVIDVRNAVPRETIMNVNVKGTQLLLEACVQASVPVFIHTSTIEVAGPNSYREIIQD
GREEEHHESAWSSPYPYSKKLAEKAVLGANGWALKNGGTLYTCALRPMYIYGEGSPFLSAYMHGALNNNG
ILTNHCKFSRVNPVYVGNVAWAHILALRALRDPKKVPNIQGQFYYISDDTPHQSYDDLNYTLSKEWGFCL
DSRMSLPISLQYWLAFLLEIVSFLLSPIYKYNPCFNRHLVTLSNSVFTFSYKKAQRDLGYEPLYTWEEAK
QKTKEWIGSLVKQHKETLKTKIH

id1_fetch -lt ids -gi 35

ID1server-back ::= ids {
 embl {
 accession "CAA35615",
 version 1
 },
 general {
 db "NCBI_EXT_ACC",
 tag str "FPAA037960"
 },
 gi 35
}

id1_fetch -lt state -fasta 'emb|CAA35615' -fmt xml

<?xml version="1.0"?>
<!DOCTYPE ID1server-back PUBLIC "-//NCBI//NCBI ID1Access/EN"
"NCBI_ID1Access.dtd">
<ID1server-back>
 <ID1server-back_gistate>40</ID1server-back_gistate>
</ID1server-back>

id1_fetch -lt state -flat '5=CAA35615.1' -fmt asnb | od -t u1

0000000 166 128 002 001 040 000 000
0000007

id1_fetch -lt state -flat '5(,CAA35615)' -fmt fasta

gi = 35, states: LIVE

id1_fetch -lt history -flat '12:35' -fmt fasta

GI Loaded DB Retrieval No.
-- ------ -- -------------
35 03/08/1999 EMBL 274319

id1_fetch -lt revisions -gi 35 -fmt fasta

GI Loaded DB Retrieval No.
-- ------ -- -------------
35 03/08/1999 EMBL 274319
35 06/06/1996 OLD02 84966
35 05/27/1995 OLDID 1524022
35 11/29/1994 OLDID 966346
35 08/31/1993 OLDID 426053
35 04/20/1993 OLDID 27

id1_fetch -fmt quality -gi 13508865

>AL590146.2 Phrap Quality (Length: 121086, Min: 31, Max: 99)
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
...
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 54 54 56 56 54 54 54 56 56 56 56 65 65 57 60 56 56 59 59
 56 56 56 49 99 31 31 49 49 54 63 63 54 51 53 55 51 51 49 58
 58 58 58 53 52 49 51 51 51 52 55 51 51 51 49 49 49 63 63 60
 65 65 59 54 54 54 54 54 56 60 60 65 65 65 65 70 70 65 65 65
 65 65 65 65 60 59 59 66 66 66 67 65 65 63 46 65 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
...
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 99 99 99 99 99
```

Examples from the Programming Manual

- applic.cpp
- smart.cpp
- An Example of a Web-based CGI Application - Source Files
  - car.cpp
  - car.hpp
  - car_cgi.cpp
  - Makefile.car_app
  - car.html

applic.cpp

```
// File name: applic.cpp
// Description: Using the CNcbiApplication class with CNcbiDiag, CArgs
// and CArgDescription classes

#include <corelib/ncbistd.hpp>
#include <corelib/ncbiutil.hpp>
#include <corelib/ncbiargs.hpp>
#include <corelib/ncbiapp.hpp>
#include <corelib/ncbienv.hpp>

USING_NCBI_SCOPE;

class CTestApp : public CNcbiApplication {
public:
 virtual int Run();
};
int CTestApp::Run() {
 auto_ptr<CArgs> args;

 // create a CArgDescriptions object to constrain the input arguments;
 // Argument descriptions are added using:

 // void AddKey(string& name, string& synopsis, string& comment, EType,
TFlags);
 // void AddOptionalKey(string& name, string& synopsis, string& comment,
EType,
 // string& default, TFlags);
 // void AddFlag(string& name, string& comment);

 {
 CArgDescriptions argDesc;

 // Required arguments:
 argDesc.AddKey("n","integer","integer between 1 and
10",argDesc.eInteger);
 argDesc.AddKey("f","float","float between 0.0 and 1.0",argDesc.eDouble);
 argDesc.AddKey("i","inputFile","data file
in",CArgDescriptions::eInputFile);

 // optional arguments:
 argDesc.AddOptionalKey("l","logFile","log errors to <logFile>",
 argDesc.eOutputFile);

 // optional flags
 argDesc.AddFlag("it", "input text");
 argDesc.AddFlag("ib", "input binary");

 try {
 args.reset(argDesc.CreateArgs(GetArguments()));
 }
 catch (exception& e) {
 string a;
 argDesc.SetUsageContext(GetArguments()[0],
 "CArgDescriptions demo program");

 cerr << e.what() << endl;
 cerr << argDesc.PrintUsage(a);
 return (-1);
 }
 }

 int intIn = (*args)["n"].AsInteger();
 float floatIn = (*args)["f"].AsDouble();
 string inFile = (*args)["i"].AsString();

 // process optional args
 if ( args->Exest("l") ) {
 SetDiagStream(&(*args)["l"].AsOutputFile());
 }

 bool textIn = args->Exist("it");
 bool binIn = (*args)["ib"].AsBoolean();

 if (! (textIn ^ binIn) ) {
 ERR_POST_X(1, Error << "input type must be specified using -it or -ib");
 }

 string InputType;
 if ( textIn ) {
 InputType = "text";
 } else if ( binIn ) {
 InputType = "binary";
 }

 ERR_POST_X(2, Info << "intIn = " << intIn << " floatIn = " << floatIn
 << " inFile = " << inFile << " input type = " << InputType);

 return 0;
}
int main(int argc, const char* argv[])
{
 CNcbiOfstream diag("moreApp.log");
 SetDiagStream(&diag);

 // Set the global severity level to Info
 SetDiagPostLevel(eDiag_Info);

 CTestApp theTestApp;
 return theTestApp.AppMain(argc, argv);
}
```

smart.cpp

```
// File name: smart.cpp
// Description: Memory management using auto_ptr versus CRef

#include <corelib/ncbiapp.hpp>
#include <corelib/ncbiobj.hpp>

USING_NCBI_SCOPE;

class CTestApp : public CNcbiApplication {
public:
 virtual int Run(void);
};
```

```
///////////////////////////////////////////////////////////////////
//
// 1. Install an auto_ptr to an int and make a copy - then try to
// reference the value from the original auto_ptr.
//
// 2. Do the same thing using CRefs instead of auto_ptrs.
//
////////////////////////////////////////////////////////////////////

int CTestApp::Run()
{
 auto_ptr<int> orig_ap;
 orig_ap.reset(new int(5));
 {
 auto_ptr<int> copy_ap = orig_ap;

 if ( !orig_ap.get() ) {
 cout << "orig_ap no longer exists - copy_ap = " << *copy_ap << endl;
 } else {
 cout << "orig_ap = " << *orig_ap << ", copy_ap = "
 << *copy_ap << endl;
 }
 }
 if ( orig_ap.get() ) {
 cout << "orig_ap = " << *orig_ap << endl;
 }

 CRef< CObjectFor<int> > orig(new CObjectFor<int>);
 *orig = 5;
 {
 CRef< CObjectFor<int> > copy = orig;

 if ( !orig ) {
 cout << "orig no longer exists - copy = " << (int&) *copy << endl;
 } else {
 cout << "orig = " << (int&) *orig << ", copy = "
 << (int&) *copy << endl;
 }
 }
 if ( orig ) {
 cout << "orig = " << (int&) *orig << endl;
 }
 return 0;
}


int main(int argc, const char* argv[])
{
 CTestApp theTestApp;
 return theTestApp.AppMain(argc, argv);
}
```

An Example of a Web-based CGI Application - Source Files

car.cpp

```
// File name: car.cpp
// Description: Implement the CCarAttr class

#include "car.hpp"

BEGIN_NCBI_SCOPE

///////////////////////////////////////////////////////////////////////////
///
// CCarAttr::

set<string> CCarAttr::sm_Features;
set<string> CCarAttr::sm_Colors;


CCarAttr::CCarAttr(void)
{
 // make sure there is only one instance of this class
 if ( !sm_Features.empty() ) {
 _TROUBLE;
 return;
 }

 // initialize static data
 sm_Features.insert("Air conditioning");
 sm_Features.insert("CD Player");
 sm_Features.insert("Four door");
 sm_Features.insert("Power windows");
 sm_Features.insert("Sunroof");

 sm_Colors.insert("Black");
 sm_Colors.insert("Navy");
 sm_Colors.insert("Silver");
 sm_Colors.insert("Tan");
 sm_Colors.insert("White");
}

// dummy instance of CCarAttr -- to provide initialization of
// CCarAttr::sm_Features and CCarAttr::sm_Colors
static CCarAttr s_InitCarAttr;
```

```
END_NCBI_SCOPE
```

car.hpp

```
// File name: car.hpp
// Description: Define the CCar and CCarAttr classes

#ifndef CAR_HPP
#define CAR_HPP

#include <coreilib/ncbistd.hpp>
#include <set>

BEGIN_NCBI_SCOPE

/////////////////////
// CCar

class CCar
{
public:
 CCar(unsigned base_price = 10000) { m_Price = base_price; }

 bool HasFeature(const string&amp; feature_name) const
 { return m_Features.find(feature_name) != m_Features.end(); }
 void AddFeature(const string& feature_name)
 { m_Features.insert(feature_name); }

 void SetColor(const string& color_name) { m_Color = color_name; }
 string GetColor(void) const { return m_Color; }

 const set<string>& GetFeatures() const { return m_Features; }
 unsigned GetPrice(void) const
 { return m_Price + 1000 * m_Features.size(); }

private:
 set<string> m_Features;
 string m_Color;
 unsigned m_Price;
};



/////////////////////
// CCarAttr -- use a dummy all-static class to store available car
attributes
```

```
class CCarAttr {
public:
 CCarAttr(void);
 static const set<string>& GetFeatures(void) { return sm_Features; }
 static const set<string>& GetColors (void) { return sm_Colors; }
private:
 static set<string> sm_Features;
 static set<string> sm_Colors;
};


END_NCBI_SCOPE

#endif /* CAR__HPP */
```

car_cgi.cpp

```
// File name: car_cgi.cpp
// Description: Implement the CCarCgi class and function main

#include <cgi/cgiapp.hpp>
#include <cgi/cgictx.hpp>
#include <html/html.hpp>
#include <html/page.hpp>

#include "car.hpp"

USING_NCBI_SCOPE;

//////////////////////////////////////////////////////////////////////////
///
// CCarCgi:: declaration

class CCarCgi : public CCgiApplication
{
public:
 virtual int ProcessRequest(CCgiContext& ctx);

private:
 CCar* CreateCarByRequest(const CCgiContext& ctx);

 void PopulatePage(CHTMLPage& page, const CCar& car);

 static CNCBINode* ComposeSummary(const CCar& car);
 static CNCBINode* ComposeForm (const CCar& car);
 static CNCBINode* ComposePrice (const CCar& car);

 static const char sm_ColorTag[];
 static const char sm_FeatureTag[];
};


///////////////////////////////////////////////////////////////////////////
///
// CCarCgi:: implementation


const char CCarCgi::sm_ColorTag[] = "color";
const char CCarCgi::sm_FeatureTag[] = "feature";


int CCarCgi::ProcessRequest(CCgiContext& ctx)
{
 // Create new "car" object with the attributes retrieved
 // from the CGI request parameters
 auto_ptr<CCar> car;
 try {
 car.reset( CreateCarByRequest(ctx) );
 } catch (exception& e) {
 ERR_POST_X(1, "Failed to create car: " << e.what());
 return 1;
 }

 // Create an HTML page (using the template file "car.html")
 CRef<CHTMLPage> page;
 try {
 page = new CHTMLPage("Car", "car.html");
 } catch (exception& e) {
 ERR_POST_X(2, "Failed to create the Car HTML page: " << e.what());
 return 2;
 }

 // Register all substitutions for the template parameters <@XXX@>
 // (fill them out using the "car" attributes)
 try {
 PopulatePage(*page, *car);
 } catch (exception& e) {
 ERR_POST_X(3, "Failed to populate the Car HTML page: " << e.what());
 return 3;
 }

 // Compose and flush the resultant HTML page
 try {
 const CCgiResponse& response = ctx.GetResponse();
 response.WriteHeader();
 page->Print(response.out(), CNCBINode::eHTML);
 response.Flush();
 } catch (exception& e) {
 ERR_POST_X(4, "Failed to compose and send the Car HTML page: " << e.what
())；
 return 4;
 }

 return 0;
}


CCar* CCarCgi::CreateCarByRequest(const CCgiContext& ctx)
{
 auto_ptr<CCar> car(new CCar);

 // Get the list of CGI request name/value pairs
 const TCgiEntries& entries = ctx.GetRequest().GetEntries();

 TCgiEntriesCI it;

 // load the car with selected features
 pair<TCgiEntriesCI,TCgiEntriesCI> feature_range =
 entries.equal_range(sm_FeatureTag);
 for (it = feature_range.first; it != feature_range.second; ++it) {
 car->AddFeature(it->second);
 }

 // color
 if ((it = entries.find(sm_ColorTag)) != entries.end()) {
 car->SetColor(it->second);
 } else {
 car->SetColor(*CCarAttr::GetColors().begin());
 }

 return car.release();
}


 /************ Create a form with the following structure:
 <form>
 <table>
 <tr>
 <td> (Features) </td>
 <td> (Colors) </td>
 <td> (Submit) </td>
 </tr>
 </table>
 </form>
 ********************/

CNCBINode* CCarCgi::ComposeForm(const CCar& car)
{
 CRef<CHTML_table> Table = new CHTML_table();
 Table->SetCellSpacing(0)->SetCellPadding(4)
 ->SetBgColor("#CCCCCC")->SetAttribute("border", "0");

 CRef<CHTMLNode> Row = new CHTML_tr();

 // features (check boxes)
 CRef<CHTMLNode> Features = new CHTML_td();
 Features->SetVAlign("top")->SetWidth("200");
 Features->AppendChild(new CHTMLText("Options: <br>"));

 ITERATE(set<string>, it, CCarAttr::GetFeatures()) {
 Features->AppendChild
 (new CHTML_checkbox
 (sm_FeatureTag, *it, car.HasFeature(*it), *it));
 Features->AppendChild(new CHTML_br());
 }

 // colors (radio buttons)
 CRef<CHTMLNode> Colors = new CHTML_td();
 Colors->SetVAlign("top")->SetWidth("128");
 Colors->AppendChild(new CHTMLText("Color: <br>"));

 ITERATE(set<string>, it, CCarAttr::GetColors()) {
 Colors->AppendChild
 (new CHTML_radio
 (sm_ColorTag, *it, !NStr::Compare(*it, car.GetColor()), *it));
 Colors->AppendChild(new CHTML_br());
 }

 Row->AppendChild(&*Features);
 Row->AppendChild(&*Colors);
 Row->AppendChild
 ((new CHTML_td())->AppendChild
 (new CHTML_submit("submit", "submit")));
 Table->AppendChild(&*Row);

 // done
 return (new CHTML_form("car.cgi", CHTML_form::eGet))->AppendChild
(&*Table);
}


CNCBINode* CCarCgi::ComposeSummary(const CCar& car)
{
 string summary = "You have ordered a " + car.GetColor() + " model";

 if ( car.GetFeatures().empty() ) {
 summary += " with no additional features.<br>";
 return new CHTMLText(summary);
 }

 summary += " with the following features:<br>";
 CRef<CHTML_ol> ol = new CHTML_ol();

 ITERATE(set<string>, it, car.GetFeatures()) {
 ol->AppendItem(*it);
 }
 return (new CHTMLText(summary))->AppendChild((CNodeRef&)ol);
}


CNCBINode* CCarCgi::ComposePrice(const CCar& car)
{
 return
 new CHTMLText("Total price: $" + NStr::UIntToString(car.GetPrice()));
}


void CCarCgi::PopulatePage(CHTMLPage& page, const CCar& car)
{
 page.AddTagMap("FORM", ComposeForm (car));
 page.AddTagMap("SUMMARY", ComposeSummary (car));
 page.AddTagMap("PRICE", ComposePrice (car));
}



///////////////////////////////////////////////////////////////////////////
///
// MAIN


int main(int argc, char* argv[])
{
 SetDiagStream(&NcbiCerr);
 return CCarCgi().AppMain(argc, argv);
}
```

Makefile.car_app

```
# Makefile: /home/zimmerma/car/Makefile.car_app
# This file was originally generated by shell script "new_project"

### PATH TO A PRE-BUILT C++ TOOLKIT ###
builddir = /netopt/ncbi_tools/c++/GCC-Debug/build
# builddir = $(NCBI)/c++/Release/build


### DEFAULT COMPILATION FLAGS -- DON'T EDIT OR MOVE THESE 4 LINES !!! ###
include $(builddir)/Makefile.mk
```

```
srcdir = .
BINCOPY = @:
LOCAL_CPPFLAGS = -I.


###########################################################################
###
### EDIT SETTINGS FOR THE DEFAULT (APPLICATION) TARGET HERE ###
APP = car.cgi
SRC = car car_cgi

# PRE_LIBS = $(NCBI_C_LIBPATH) .....
LIB = xhtml xcgi xncbi

# LIB = xser xhtml xcgi xncbi xconnect
# LIBS = $(NCBI_C_LIBPATH) -lncbi $(NETWORK_LIBS) $(ORIG_LIBS)

# CPPFLAGS = $(ORIG_CPPFLAGS) $(NCBI_C_INCLUDE)
# CFLAGS = $(ORIG_CFLAGS)
# CXXFLAGS = $(ORIG_CXXFLAGS)
# LDFLAGS = $(ORIG_LDFLAGS)
# ###
###########################################################################
###

### APPLICATION BUILD RULES -- DON'T EDIT OR MOVE THIS LINE !!! ###
include $(builddir)/Makefile.app


### PUT YOUR OWN ADDITIONAL TARGETS (MAKE COMMANDS/RULES) BELOW HERE ###
```

car.html

```
<html>
<head>
<title>Automobile Order Form</title>
</head>
<body>
<h1>Place your order here</h1>
<@FORM@>
<@SUMMARY@>
<@PRICE@>
</body>
</html>
```

Table 1. Invocation flags

Argument	Value	Effect
-h		Print usage message and exit.
-gi N	integer	GenInfo ID of sequence to look up.
-fmt fmt	format type	Output data format; default is asn (ASN.1 text).
-out file	filename	Write output to specified file rather than stdout.
-log file	filename	Write errors and messages to specified file rather than stderr.
-db str	string	Use specified database. **Mandatory for** Entrez **queries**, where it is normally either Nucleotide or Protein. Also specifies satellite database for sequence-entry lookups.
-ent N	integer	Dump specified subentity. Only relevant for sequence-entry lookups.
-lt type	lookup type	Type of lookup; default is entry (sequence entry).
-in file	filename	Read sequence IDs from file rather than command line. May contain raw GI IDs, flattened IDs, and FASTA-format IDs.
-maxplex m	complexity	Maximum output complexity level; default is entry (entire entry).
-flat id	flat ID	Flattened ID of sequence to look up.
-fasta id	FASTA ID	FASTA-style ID of sequence to look up.
-query str	string	Generate ID list from specified Entrez query.
-qf file	file	Generate ID list from Entrez query in specified file.

Table 2. Output data formats

Value	Format	Comments
asn	ASN.1 text (default)	
asnb	ASN.1 binary	
docsum	Entrez document summary	Lookup type is irrelevant.
fasta	FASTA	Produces state as simple text; produces history in tabular form.
genbank	GenBankflat-file format	Lookup type must be entry (default).
genpept	GenPept flat-file format	Lookup type must be entry (default).
quality	Quality scores	Lookup type must be entry (default); data not always available.
xml	XML	Isomorphic to ASN.1 output.

Table 3. Lookup types

Value	**Description**
entry	The actual sequence entry (default)
history	Summary of changes to the sequence data
ids	All of the sequence's IDs
none	Just the GI ID
revisions	Summary of changes to the sequence data or annotations
state	The sequence's status

Table 4. Maximum output complexity level values

Value	Description
bioseq	Just the bioseq of interest
bioseq-set	Minimal bioseq-set
entry	Entire entry (default)
nuc-prot	Minimal nuc-prot
pub-set	Minimal pub-set

Table 5. FASTA sequence ID format values

Type	**Format(s)** [1]	**Example(s)**
local	lcl|integer lcl|string	lcl|123 lcl|hmm271
GenInfo backbone seqid	bbs|integer	bbs|123
GenInfo backbone moltype	bbm|integer	bbm|123
GenInfo import ID	gim|integer	gim|123
GenBank	gb|accession|locus	gb|M73307|AGMA13GT
EMBL	emb|accession|locus	emb|CAM43271.1|
PIR	pir|accession|name	pir||G36364
SWISS-PROT	sp|accession|name	sp|P01013|OVAX_CHICK
patent	pat|country|patent|sequence	pat|US|RE33188|1
pre-grant patent	pgp|country|application-number|seq-number	pgp|EP|0238993|7
RefSeq [2]	ref|accession|name	ref|NM_010450.1|
general database reference	gnl|database|integer gnl|database|string	gnl|taxon|9606 gnl|PID|e1632
GenInfo integrated database	gi|integer	gi|21434723
DDBJ	dbj|accession|locus	dbj|BAC85684.1|
PRF	prf|accession|name	prf||0806162C
PDB	pdb|entry|chain	pdb|1I4L|D
third-party GenBank	tpg|accession|name	tpg|BK003456|
third-party EMBL	tpe|accession|name	tpe|BN000123|
third-party DDBJ	tpd|accession|name	tpd|FAA00017|
TrEMBL	tr|accession|name	tr|Q90RT2|Q90RT2_9HIV1
genome pipeline [3]	gpp|accession|name	gpp|GPC_123456789|
named annotation track [3]	nat|accession|name	nat|AT_123456789.1|

[1] Spaces should not be present in ID's. It's okay to leave off the final vertical bar for most text ID types (such as gb) when the locus is absent; apart from that, vertical bars must be present even if an adjacent field is omitted.

[2] Some RefSeq accessions have additional letters following the underscore. See the RefSeq accession format reference for details.

[3] For NCBI internal use.

26: C Toolkit Resources for C++ Toolkit Users

Last Update: March 10, 2011.

Overview

For certain tasks in the C++ Toolkit environment, it is necessary to use, or at least refer to, material from the NCBI C Toolkit. Here are some links relevant to the C Toolkit:

- C Toolkit Documentation
- C Toolkit Queryable Source Browser

Chapter Outline

The following is an outline of the topics presented in this chapter:

Using NCBI C and C++ Toolkits together

Note: Due to security issues, not all links on this page are accessible by users outside NCBI.

Overview

When using both C and C++ Toolkits together in a single application, it is very important to understand that there are some resources shared between the two. This document describes how to safely use both Toolkits together, and how to gain from their cooperation.

Shared Sources

To maintain a sort of uniformity and ease in source code maintenance, the CONNECT library is the first library of both Toolkits kept the same at the source code level. To provide data interoperability, ASN.1 specifications have to be identical in both Toolkits, too.

CONNECT Library

The CONNECT library is currently the only C code that is kept identical in both Toolkits. The old API of the CONNECT library is still supported by means of a simple wrapper, which is only in the C Toolkit. There are two scripts that perform synchronization between C++ Toolkit and C Toolkit:

sync_c_to_cxx.pl – This script copies the latest changes made in the C Toolkit (which is kept in the CVS repository) to the C++ Toolkit (kept in the Subversion repository). The following files are presently copied: gicache.h and gicache.c. Both are copied from the distrib/network/sybutils/ctlib CVS module to the trunk/c++/src/objtools/data_loaders/genbank/gicache location in the Toolkit repository.

sync_cxx_to_c.pl – This script copies files in the opposite direction: from the C++ Toolkit to the C Toolkit. Most of the files common to both Toolkits are synchronized by this script. Here's the list of C source directories (CVS modules) that are currently copied from Subversion:
- connect
- ctools
- algo/blast/core
- algo/blast/composition_adjustment
- util/tables
- util/creaders

ASN files in the following CVS modules are also synchronized with Subversion:
- network/medarch/client
- network/taxon1/common
- network/id1arch
- network/id2arch
- access
- asn
- biostruc

- biostruc/cdd
- biostruc/cn3d
- tools
- api
- data

ASN.1 Specifications

Unlike the C source files in the CONNECT library, the ASN.1 data specifications are maintained within C Toolkit source structure, and have to be copied over to C++ Toolkit tree whenever they are changed.

However, the internal representations of ASN.1-based objects differ between the two Toolkits. If you need to convert an object from one representation to the other, you can use the template class CAsnConverter<>, defined in ctools/asn_converter.hpp.

Run-Time Resources

The CONNECT library was written for use "as is" in the C Toolkit, but it must also be in the C++ Toolkit tree. Therefore, it cannot directly employ the utility objects offered by the C++ Toolkit such as message logging CNcbiDiag, registry CNcbiRegistry, and MT-locks CRWLock. Instead, these objects were replaced with helper objects coded entirely in C (as tables of function pointers and data).

On the other hand, throughout the code, the CONNECT library refers to predefined objects g_CORE_Log (so called CORE C logger) g_CORE_Registry (CORE C registry), and g_CORE_Lock (CORE C MT-lock), which actually are never initialized by the library, i.e. they are empty objects, which do nothing. It is an application's resposibility to replace these dummies with real working logger, registry, and MT-lock objects. There are two approaches, one for C and another for C++.

C programs can call CORE_SetREG(), CORE_SetLOG(), and CORE_SetLOCK() to set up the registry, logger, and MT-lock (connect/ncbi_util.h must also be included). There are also convenience routines for CORE logger, like CORE_SetLOGFILE(), CORE_SetLOGFILE_NAME(), which facilitate redirecting logging messages to either a C stream (FILE*) or a named file.

In a C++ program, it is necessary to convert native C++ objects into their C equivalents, so that the C++ objects can be used where types LOG, REG or MT_LOCK are expected. This is done using calls declared in connect/ncbi_core_cxx.hpp, as described later in this section.

LOG and CNcbiDiag

The CONNECT library has its own logger, which has to be set by one of the routines declared in connect/ncbi_util.h: CORE_SetLOG(), CORE_SetLOGFILE() etc. On the other hand, the interface defined in connect/ncbi_core_cxx.hpp provides the following C++ function to convert a logging stream of the NCBI C++ Toolkit into a LOG object:

```
LOG LOG_cxx2c (void)
```

This function creates the LOG object on top of the corresponding C++ CNcbiDiag object, and then both C and C++ objects can be manipulated interchangeably, causing exactly the same effect on the underlying logger. Then, the returned C handle LOG can be subsequently used as a CORE C logger by means of CORE_SetLOG(), as in the following nested calls:

```
CORE_SetLOG(LOG_cxx2c());
```

REG and CNcbiRegistry

connect/ncbi_core_cxx.hpp declares the following C++ function to bind C REG object to CNcbiRegistry used in C++ programs built with the use of the NCBI C++ Toolkit:

```
REG REG_cxx2c (CNcbiRegistry* reg, bool pass_ownership = false)
```

Similarly to CORE C logger setting, the returned handle can be used later with CORE_SetREG () declared in connect/ncbi_util.h to set up the global registry object (CORE C registry).

MT_LOCK and CRWLock

There is a function

```
MT_LOCK MT_LOCK_cxx2c (CRWLock* lock, bool pass_ownership = false)
```

declared in connect/ncbi_core_cxx.hpp, which converts an object of class CRWLock into a C object MT_LOCK. The latter can be used as an argument to CORE_SetLOCK() for setting the global CORE C MT-lock, used by a low level code, written in C. Note that passing 0 as the lock pointer will effectively create a new internal CRWLock object, which will then be converted into MT_LOCK and returned. This object gets automatically destroyed when the corresponding MT_LOCK is destroyed. If the pointer to CRWLock is passed a non NULL value then the second argument can specify whether the resulting MT_LOCK acquires the ownership of the lock, thus is able to delete the lock when destructing itself.

CONNECT Library in C++ Code

Setting LOG

To set up the CORE C logger to use the same logging format of messages and destination as used by CNcbiDiag, the following sequence of calls may be used:

```
CORE_SetLOG(LOG_cxx2c());
SetDiagTrace(eDT_Enable);
SetDiagPostLevel(eDiag_Info);
SetDiagPostFlag(eDPF_All);
```

Setting REG

To set the CORE C registry be the same as C++ registry CNcbiRegistry, the following call is necessary:

```
CORE_SetREG(REG_cxx2c(cxxreg, true));
```

Here cxxreg is a CNcbiRegistry registry object created and maintained by a C++ application.

Setting MT-Locking

To set up a CORE lock, which is used throughout the low level code, including places of calls of non-reentrant library calls (if no reentrant counterparts were detected during configure process), one can place the following statement close to the beginning of the program:

```
CORE_SetLOCK(MT_LOCK_cxx2c());
```

Note that the use of this call is extremely important in a multi-threaded environment.

Convenience call CONNECT_Init()

Header file connect/ncbi_core_cxx.hpp provides a convenience call, which sets all shared CONNECT-related resources discussed above for an application written within the C++ Toolkit framework (or linked solely against the libraries contained in the Toolkit):

```
void CONNECT_Init(CNcbiRegistry* reg = NULL);
```

The call takes only one argument, an optional pointer to a registry, which is used by the application, and should also be considered by the CONNECT library. No registry will be used if NULL gets passed. The ownership of the registry is passed along. This fact should be noted by an application making extensive use of CONNECT in static classes, i.e. prior to or after main(), because the registry can get deleted before the CONNECT library stops using it. The call also ties CORE C logger to CNcbiDiag, and privately creates a CORE C MT-lock object (on top of CRWLock) for internal synchronization inside the library.

An example of how to use this call can be found in the test program test_ncbi_conn_stream.cpp. It shows how to properly setup CORE C logger, CORE C registry and CORE C MT-lock so they will use the same data in the C and C++ parts of both the CONNECT library and the application code.

Another good source of information is the set of working application examples in src/app/id1_fetch. Note: In the examples, the convenience routine does not change logging levels or disable/enable certain logging properties. If this is desired, the application still has to use separate calls.

C Toolkit diagnostics redirection

In a C/C++ program linked against both NCBI C++ and NCBI C Toolkits the diagnostics messages (if any) generated by either Toolkit are not necessarily directed through same route, which may result in lost or garbled messages. To set the diagnostics destination be the same as CNcbiDiag's one, and thus to guarantee that the messages from both Toolkits will be all stored sequentially and in the order they were generated, there is a call

```
#include <ctools/ctools.h>
void SetupCToolkitErrPost(void);
```

which is put in a specially designated directory ctools providing back links to the C Toolkit from the C++ Toolkit.

CONNECT Library in C Code

The CONNECT library in the C Toolkit has a header connect/ncbi_core_c.h, which serves exactly the same purpose as connect/ncbi_core_cxx.hpp, described previously. It defines an API to convert native Toolkit objects, like logger, registry, and MT-lock into their abstract equivalents, LOG, REG, and MT_LOCK, respectively, which are defined in connect/ncbi_core.h, and subsequently can used by the CONNECT library as CORE C objects.

Briefly, the calls are:

- LOG LOG_c2c (void); Create a logger LOG with all messages sent to it rerouted via the error logging facility used by the C Toolkit.

- REG REG_c2c (const char* conf_file); Build a registry object REG from a named file conf_file. Passing NULL as an argument causes the default Toolkit registry file to be searched for and used.
- MT_LOCK MT_LOCK_c2c (TNlmRWlock lock, int/*bool*/ pass_ownership); Build an MT_LOCK object on top of TNlmRWlock handle. Note that passing NULL effectively creates an internal handle, which is used as an underlying object. Ownership of the original handle can be passed to the resulting MT_LOCK by setting the second argument to a non-zero value. The internally created handle always has its ownership passed along.

Exactly the same way as described in the previous section, all objects, resulting from the above functions, can be used to set up CORE C logger, CORE C registry, and CORE MT-lock of the CONNECT library using the API defined in connect/ncbi_util.h: CORE_SetLOG(), CORE_SetREG(), and CORE_SetLOCK(), respectively.

Convenience call CONNECT_Init()

As an alternative to using per-object settings as shown in the previous paragraph, the following "all-in-one" call is provided:

```
void CONNECT_Init (const char* conf_file);
```

This sets CORE C logger to go via Toolkit default logging facility, causes CORE C registry to be loaded from the named file (or from the Toolkit's default file if conf_file passed NULL), and creates CORE C MT-lock on top of internally created TNlmRWlock handle, the ownership of which is passed to the MT_LOCK.

Note: Again, properties of the logging facility are not affected by this call, i.e. the selection of what gets logged, how, and where, should be controlled by using native C Toolkit's mechanisms defined in ncbierr.h.

Access to the C Toolkit source tree Using CVS

For a detailed description of the CVS utility see the CVS online manual or run the commands "man cvs" or "cvs --help" on your Unix workstation.

CVS Source Code Retrieval for Public Read-only Access

Public access to the public part of the C Toolkit is available via CVS client. To use it, follow exactly the in-house Unix / Mac OS X instructions with two exceptions:

- The CVSROOT env. variable should be set to:
  :pserver:anoncvs@anoncvs.ncbi.nlm.nih.gov:/vault
- Use empty password to login:
  > cvs login
  Logging in to :pserver:anoncvs@anoncvs.ncbi.nlm.nih.gov:/vault
  CVS password: <just press ENTER here>

Public web access is also available via ViewVC.

CVS Source Code Retrieval for In-House Users with Read-Write Access

You must have a CVS account set up prior to using CVS - email svn-admin@ncbi.nlm.nih.gov to get set up.

The C Toolkit CVS repository is available online and may be searched using LXR.

- Using CVS from Unix or Mac OS X
- Using CVS from Windows

Using CVS from Unix or Mac OS X

To set up a CVS client on Unix or Mac OS X:

- Set the CVSROOT environment variable to: :pserver:${LOGNAME}@cvsvault:/src/NCBI/vault.ncbi. Note that for NCBI Unix users, this may already be set if you specified developer for the facilities option in the .ncbi_hints file in your home directory.
- Run the command: cvs login You will be asked for a password (email svn-admin@ncbi.nlm.nih.gov if you need the password). This command will record your login info into ~/.cvspass file so you won't have to login into CVS in the future. Note: You may need to create an empty ~/.cvspass file before logging in as some CVS clients apparently just cannot create it for you. If you get an authorization error, then send e-mail with the errors to cpp-core@ncbi.nlm.nih.gov.
- If you have some other CVS snapshot which was checked out with an old value of CVSROOT, you should commit all your changes first, then delete completely the old snapshot dir and run: cvs checkout to get it with new CVSROOT value.
- Now you are all set and can use all the usual CVS commands.

Note: When you are in a directory that was created with cvs checkout by another person, a local ./CVS/ subdirectory is also created in that directory. In this case, the cvs command ignores the current value of the CVSROOT environment variable and picks up a value from ./CVS/Root file. Here is an example of what this Root file looks like:

```
:pserver:username@cvsvault:/src/NCBI/vault.ncbi
```

Here the *username* is the user name of the person who did the initial CVS checkout in that directory. So CVS picks up the credentials of the user who did the initial check-in and ignores the setting of the CVSROOT environment variable, and therefore the CVS commands that require authorization will fail. There are two possible solutions to this problem:

- Create your own snapshot of this area using the cvs get command.
- Impersonate the user who created the CVS directory by creating in the ~/.cvspass file another string which is a duplicate of the existing one, and in this new string change the username to that of the user who created the directory. This hack will allow you to work with the CVS snapshot of the user who created the directory. However, this type of hack is not recommended for any long term use as you are impersonating another user.

Using CVS from Windows

The preferred CVS client is TortoiseCVS. If this is not installed on your PC, ask PC Systems to have it installed. Your TortoiseCVS installation should include both a CVS command-line client and integration into Windows Explorer.

To use TortoiseCVS as integrated into Windows Explorer:

- Navigate to the directory where you want the source code to be put.
- Right-click and select "CVS Checkout".
- Set the CVSROOT text field to :pserver:%USERNAME%@cvsvault:/src/NCBI/vault.ncbi (where %USERNAME% is replaced with your Windows user name).

- Set the module text field to the portion of the C Toolkit you want to retrieve. If you want the whole Toolkit, use distrib. If you want just one library, for example the CONNECT library, use distrib/connect. There are also non C Toolkit modules (you can see them here). You can work with those as well by using their names instead of distrib (e.g. internal).
- Click OK. If you are asked for a password and don't know what to use, email svn-admin@ncbi.nlm.nih.gov.
- From the context menu (right-click) you can now perform CVS functions, such as updating, committing, tagging, diffing, etc.
- You may also change global preferences (such as external tools) using the Preferences application available from the Start menu.

For command-line use, follow the in-house Unix / Mac OS X instructions with these exceptions:

- Make sure you have your "home" directory set up -- i.e. the environment variables HOMEDRIVE and HOMEPATH should be set. In NCBI, HOMEDRIVE usually set to C:, and HOMEPATH is usually set to something like \Documents and Settings\%USERNAME% (where %USERNAME% is replaced with your Windows user name).
- Create an empty file named .cvspass in your "home" directory.
- The CVS root needs to be specified.
  - Either set an environment variable:
    %CVSROOT%=:pserver:%USERNAME%@cvsvault:/src/NCBI/vault.ncbi
  - or use a command-line argument for each CVS command:
    -d :pserver:%USERNAME%@cvsvault:/src/NCBI/vault.ncbi
- Open a command shell, verify the above environment variables are set properly, and execute the command "cvs login". You will be asked for a password (email svn-admin@ncbi.nlm.nih.gov if you need the password). This command will record your login info in the .cvspass file so you won't have to log into CVS in the future. If you get an authorization error, send e-mail with the errors to cpp-core@ncbi.nlm.nih.gov.

Part 6: Help and Support

Part 6 discusses the different source code browsers, FAQs, XML Authoring, and other useful documentation. The following is a list of chapters in this part:

27: NCBI C++ Toolkit Source Browser

Source Browsers

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The NCBI C++ Toolkit source code is highly browseable and can be searched in a variety of useful ways. To that end we provide two source browsers, one based on the LXR Engine and another based on Doxygen. These are complementary approaches that allow the Toolkit source to be searched and navigated according to its file hierarchy and present an alphabetical list of all classes, macros, variables, typedefs, etc. named in the Toolkit, as well as a summary of the parent-child relationships among the classes.

Chapter Outline

The following is an outline of the topics presented in this chapter:

LXR

The LXR Engine enables search-driven browsing together with a more conventional navigation of the Toolkit's source. In source mode, LXR provides navigation of the source tree through a Web-based front end. The LXR search modes ident, find and search will generate a list to identify all occurrences in the Toolkit where an identifier, file name, or specified free text, respectively, are found.

An identifier in an LXR search is the name of a class, function, variable, macro, typedef, or other named entity within the Toolkit source. This search can be especially handy when attempting to determine, for example, which header has been left out when a symbol reference cannot be found.

Some hints for using LXR:

- For free-text LXR searches, patterns, wildcards, and regular expression syntax are allowed. See the Search Help page for details.
- The identifier ("ident") and file ("find") LXR search modes attempt an **exact** and **case-sensitive** match to your query.
- LXR indexes files from a root of $NCBI/c++; matches will be found not only in src and include but also in any resident build tree and the compilers and scripts directories as well.
- Note: The documentation itself is not searched by LXR.

Doxygen Browser

The Doxygen tool has been used to generate a Toolkit source code browser from the source code files. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code. Doxygen has been configured to extract the code structure directly from the source code files. This feature is very useful because it quickly enables you to find your way in large source distributions. You can also visualize the relations between the various elements by means of dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

28: Software Development Tools

Created: April 1, 2003.
Last Update: March 19, 2004.

Software Development Tools

The overview for this chapter consists of the following topics:

Introduction

The tools used in-house by software developers at NCBI -- such as debuggers, memory checkers, profilers, etc. are discussed in C++ Toolkit Wiki (see links below).

Chapter Outline

The following is an outline of the topics presented in this chapter:

Section Placeholder

This section is only here for technical reasons. All meaningful content is above

29: XML Authoring using Word

Last Update: March 28, 2012.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter describes writing new chapters and editing the existing chapters for C++Toolkit book using the Bookshelf Authoring Template. This template allows creating documents that can automatically be converted to XML for use on the NCBI Bookshelf. Although the NCBI Bookshelf uses XML internally, the authoring template is based on Microsoft Word and does not require any prior knowledge of XML. This approach has the advantage of being able to use Word's spelling and grammar checking and avoid editing the actual XML document.

Chapter Outline

Writing a new chapter

Before writing a new chapter, please email us at cpp-core@ncbi.nlm.nih.gov with a description of the new chapter and to coordinate the addition of the new chapter with other work we are performing on the book. You also will need a version of Microsoft Word that can open, edit, and save files in Word 2003 format. Closely following the Bookshelf Authoring Template guideline is essential to ensure the proper Word-to-XML conversion.

Editing Existing Chapters

To edit an existing chapter, please email us at cpp-core@ncbi.nlm.nih.gov to obtain the correct Word file for the chapter. You also will need version of Microsoft Word that can open, edit, and save files in Word 2003 format. Closely following the Bookshelf Authoring Template guideline is essential to ensure the proper Word-to-XML conversion.

Editing Part Overviews

Each major book part has a short overview / contents page just for that part (for example, see the Part 1 page). All overviews are collectively maintained in the booktoolkit.xml file.

Documentation styles

The basic documentation styles are described in the table.

Name	Description	Example
nc-ncbi-app	Program name	datatool
nc-ncbi-class	Class	CTime
nc-ncbi-cmd	Command, script source code	configure --help
nc-ncbi-code	Source code	count = 0
nc-ncbi-func	Function	SetAttribute()
nc-ncbi-highlight	Highlights	NOTE:
nc-intro-subsect-head	Introduction and outline headings	Introduction
nc-ncbi-lib	Librariy name	connect
nc-ncbi-macro	Macro	_TRACE, ERR_POST, BEGIN_NCBI_SCOPE
nc-ncbi-monospace	Monospace font	C++ type: int, short, unsigned, long, etc.
nc-ncbi-path	Directories, file names, paths	/internal/c++
nc-ncbi-type	Type	int, FILE*, TMode, EType
nc-ncbi-var	Variables	i, count, m_Name, eDiag_Warning
nc-ncbi-pageobject	Tab, button, other page objects	Preview/Index

30: FAQs, Useful Documentation Links, and Mailing Lists

Last Update: May 16, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter contains frequently asked questions and useful links.

Chapter Outline

FAQs

Security

Following are some of the common questions regarding security. If you have a different question, or if these questions don't fully address your concern, please email your question to cpp-core@ncbi.nlm.nih.gov.

- How do I prevent SQL Injection attacks?
- How do I prevent Cross-Site Scripting (XSS) attacks?

How do I prevent SQL Injection attacks?

Summary:

1. No SQL EVER passed in by a user is allowed.
2. Use stored procedures whenever possible.
3. If stored procedures are not possible, and if the SQL statement needs to be constructed from user input, use parameterized SQL whenever possible.
4. If constructing dynamic SQL from user input is unavoidable, you MUST sanitize the user input.

Please see the NCBI document "SQL Security and Hygiene" for more details.

Sample code is also available for SDBAPI and DBAPI.

For more information on using a database from your application, see the "Database Access" chapter of the C++ Toolkit book.

How do I prevent Cross-Site Scripting (XSS) attacks?

NEVER trust user-supplied strings - always sanitize them before using them.

Before including a user-supplied string in:	**Sanitize the string with:**
a URL	NStr::URLEncode()
JavaScript	NStr::JavaScriptEncode()
XML	NStr::XmlEncode()
HTML	NStr::HtmlEncode()
JSON	NStr::JsonEncode()
SQL	NStr::SQLEncode()

Note: In addition to sanitizing URLs with NStr::URLEncode(), the CUrl class can be used to take apart and reassemble a URL. If the original URL was malformed an error would be produced. At the very least, improper URL segments would be mangled.

General

How do I prepare my development environment for using the C++ Toolkit?

That depends on your development environment and whether you are inside or outside of NCBI:

- Unix or Mac OS X inside NCBI
- Unix or Mac OX X outside NCBI
- Windows inside NCBI
- Windows outside NCBI

Unix or Mac OS X inside NCBI

All developer Unix accounts should be automatically prepared for using the C++ Toolkit. You should have a ~/.ncbi_hints file with a non-trivial facilities line that will be sourced when logging in. If everything is set up properly, the following commands should provide meaningful output:

```
svn --version
new_project
echo $NCBI
```

Unix or Mac OX X outside NCBI

After downloading the Toolkit source, set environment variable NCBI to <toolkit_root> (where <toolkit_root> is the top-level directory containing configure) and add $NCBI/scripts/common to your PATH.

Once the Toolkit is configured and built, then you can use it.

Windows inside NCBI

A supported version of MSVC must be installed.

A Subversion client must be installed. For help on that, please see \\snowman\win-coremake\App\ThirdParty\Subversion. To make sure subversion is working, enter svn --version in your cmd.exe shell.

Your PATH should include \\snowman\win-coremake\Scripts\bin.

If you want to step into the source for the C++ Toolkit libraries while debugging, then drive S: must be mapped to \\snowman\win-coremake\Lib. You can map it or let the new_project script map it for you.

Windows outside NCBI

A supported version of MSVC must be installed.

Download the Toolkit source.

Once the Toolkit is configured and built, then you can use it.

The GetTypeInfo() method is not declared or defined in any of the objects for which it is part of the interface

The macro DECLARE_INTERNAL_TYPE_INFO() is used in the *.hpp files to declare the GetTypeInfo(). There are several macros that are used to implement GetTypeInfo() methods in *.cpp files. These macros are generally named and used as follows:

```
BEGIN_*_INFO(...)
{
 ADD_*(...)
 ...
}
```

See User-defined Type Information in the Programming Manual for more information.

*Which include file should be used in *.cpp files, class.hpp or class_.hpp?*

Include the **class**.hpp (file without underscore). Never instantiate or use a class of the form C*_Base directly. Instead use the C* form which inherits from the C*_Base class (e.g., don't use CSeq_id_Base directly -- use CSeq_id instead).

How can I disable the XML declaration or DOCTYPE with the serial library?

Here's a code snippet that shows all combinations:

```
// serialize XML with both an XML declaration and with a DOCTYPE (default)
ostr << MSerial_Xml << obj;

// serialize XML without an XML declaration
ostr << MSerial_Xml(fSerial_Xml_NoXmlDecl) << obj;

// serialize XML without a DOCTYPE
ostr << MSerial_Xml(fSerial_Xml_NoRefDTD) << obj;

// serialize XML without either an XML declaration or a DOCTYPE
ostr << MSerial_Xml(fSerial_Xml_NoXmlDecl | fSerial_Xml_NoRefDTD) << obj;
```

Note: The serial library can read XML whether or not it contains the XML declaration or DOCTYPE without using special flags. For example:

```
istr >> MSerial_Xml >> obj;
```

Compiling

How do I compile for 32-bit on a 64-bit machine in a typical C++ Toolkit app?

Our 64-bit Linux systems only support building 64-bit code; to produce 32-bit binaries, you'll need a 32-bit system.

Which Visual C++ project should I build?

After creating a new project, you may notice quite a few projects appear in the solution, besides your program, and that the **-HIERARCHICAL-VIEW-** project is bold (indicating that it's the startup project). Do not build any of these projects or the solution as a whole. Instead, set your program as the default startup project and build it.

You can build **-CONFIGURE-DIALOG-** if you need to reconfigure your project (see the section on using the configuration GUI), and you will need to build **-CONFIGURE-** if you add libraries (see the question below on adding a library to a Visual C++ project).

What compiler options are required to be compatible with C++ Toolkit libraries?

These compiler options must be properly set under Microsoft Visual C++:

- C++ exceptions
- Runtime library
- Checked iterators

C++ exceptions

NCBI C++ Toolkit libraries use the /EHsc compiler option with Visual C++ to:

- ensure that C++ objects that will go out of scope as a result of the exception are destroyed;
- ensure that only C++ exceptions should be caught; and
- assume that extern C functions never throw a C++ exception.

For more information, see the MSDN page on /EH.

Runtime library

You must specify the appropriate Visual C++ runtime library to link with:

Configuration	**Compiler Option**
DebugDLL	/MDd
DebugMT	/MTd
ReleaseDLL	/MD
ReleaseMT	/MT

For more information, see the MSDN page on runtime library options.

Checked iterators

Note: Parts of this section refer to Visual C++ 2008, which is no longer supported. This content is currently retained for historical reference only, and may be removed in the future.

Microsoft Visual C++ provides the option of using "Checked Iterators" to ensure that you do not overwrite the bounds of your STL containers. Checked iterators have a different internal structure than, and are therefore incompatible with, non-checked iterators. If both are used in the same program, it will probably crash. Checked iterators also have somewhat lower performance than non-checked iterators.

Therefore, when building with Visual C++, you must ensure that the same checked iterators setting is used for all compilation units that use STL iterators. This includes the Visual C++ standard libraries, the NCBI C++ Toolkit libraries, your code, and any other libraries you link with.

To disable checked iterators, set _SECURE_SCL=0; to enable them, set _SECURE_SCL=1.

The Visual C++ defaults for _SECURE_SCL are:

Visual C++ Version	**Debug**	**Release**
2010	1	0
2008	1	1

By default, the compiler options for NCBI C++ Toolkit libraries do not specify the _SECURE_SCL option for debug configurations, and specify _SECURE_SCL=0 for release configurations. Therefore they use checked iterators for debug configurations, but not for release configurations.

Note: Your code may crash if any two libraries you link with don't use the same settings. For example:

- You're building a release configuration using Visual C++ 2008. You build the C++ Toolkit separately and use it as a third party package (in which case it will use _SECURE_SCL=0). Your other code and/or other libraries are compiled with default settings (which for release in VS2008 sets _SECURE_SCL=1).
- You're using a third party library that uses different settings than the C++ Toolkit.

If you need to use a different setting for _SECURE_SCL than the Toolkit uses, you will have to recompile all Toolkit libraries that you want to link with. To change this setting and rebuild the Toolkit:

1. Open src\build-system\Makefile.mk.in.msvc.
2. Edit the PreprocessorDefinitions entry in the [Compiler.*release] section for the desired configuration(s), using _SECURE_SCL=0; or _SECURE_SCL=1;.
3. Build the -CONFIGURE- project in the solution that contains all the Toolkit libraries you want to use. See the section on choosing a build scope for tips on picking the solution. Ignore the reload solution prompts - when the build completes, then close and reopen the solution.
4. Build the -BUILD-ALL- project to rebuild the libraries.

A similar situation exists for the _HAS_ITERATOR_DEBUGGING macro, however the C++ Toolkit does not set this macro for either 2008 or 2010, so you are unlikely to encounter any problems due to this setting. It's possible (however unlikely) that other third party libraries could turn this macro off in debug, in which case you'd have to rebuild so the settings for all libraries match.

By default, _HAS_ITERATOR_DEBUGGING is turned on in debug but can be turned off. However, it cannot be turned on in release.

Finally, the macro _ITERATOR_DEBUG_LEVEL was introduced with Visual C++ 2010 to simplify the use of _SECURE_SCL and _HAS_ITERATOR_DEBUGGING.

If you set _ITERATOR_DEBUG_LEVEL, then _SECURE_SCL and _HAS_ITERATOR_DEBUGGING will be set according to this table:

_ITERATOR_DEBUG_LEVEL	**_SECURE_SCL**	**_HAS_ITERATOR_DEBUGGING**
0	0	0
1	1	0
2	1	1

If you don't set _ITERATOR_DEBUG_LEVEL, it will be set automatically according to the values of _SECURE_SCL and _HAS_ITERATOR_DEBUGGING per the above table. Therefore, you can use either _ITERATOR_DEBUG_LEVEL or _SECURE_SCL and _HAS_ITERATOR_DEBUGGING as you see fit. In most cases, you won't need to set any of them. You just need to know about them in case you link with libraries that use different settings.

For more information, see:

- Checked Iterators
- What's New in Visual C++
- Breaking Changes in Visual C++

Linking

How does one find the libraries to link when the linker complains of undefined symbols?

Two tools are available to resolve the common linking questions:

Question	**Tool**
Which libraries contain a given symbol?	Library search
Which other libraries does a given library depend on?	Library dependencies

For example, suppose the linker complains about the symbol ncbi::CIStreamBuffer::FindChar (char) being undefined. Here is how to use these tools to resolve this problem:

1. To find the library(s) where the unresolved symbol is defined, use the Library search tool:

   Using the example above, enter FindChar as a search term. The library where this symbol is defined is libxutil.a (ncbi_core).

   Now that you have the library that defines the symbol, you can proceed to find the library dependencies it introduces. Note: The simplest way to do this is by just clicking on the library in the search results to show its dependencies. Alternatively, you can proceed to step 2.

2. The Library dependencies tool finds all the other libraries that a given library depends on. This tool can also help you create the LIB and LIBS lines in your makefile. For example, enter your current LIB and LIBS lines plus the new library from step 1, and it will generate optimized LIB and LIBS lines containing the library needed for your

symbol and any other needed libraries.

Continuing with the example above, entering libxutil.a (or just xutil) will create this result:

```
LIB  = xutil xncbi
LIBS = $(ORIG_LIBS)
```

Clicking on any of the links will show the required items for that link plus a dependency graph for the clicked item. The nodes in the diagram are also navigable.

Note: If you are using Visual C++, please also see the question about adding libraries to Visual C++ projects.

To make it easier to work with the NCBI C++ Toolkit's many libraries, we have generated illustrations of their dependency relationships, available for various scopes and in various formats:

NCBI C++ Library Dependency Graphs (including internal libraries)

	GIF	**PNG**	**PDF**	**PostScript**	**Text**
All libraries			PDF	PS	TXT
Just C++ Toolkit libraries			PDF	PS	
Highly connected or otherwise noteworthy public libraries	GIF	PNG	PDF	PS	

NCBI C++ Library Dependency Graphs (public libraries only)

	GIF	**PNG**	**PDF**	**PostScript**	**Text**
All libraries			PDF	PS	TXT
Non-GUI libraries			PDF	PS	
GUI libraries	GIF	PNG	PDF	PS	
Highly connected or otherwise noteworthy public libraries	GIF	PNG	PDF	PS	

In cases where the above methods do not work, you can also search manually using the following steps:

1. Look for the source file that defines the symbol. This can be done by going to the LXR source browser and doing an identifier search on the symbol (e.g., CDate or XmlEncode). Look for a source file where the identifier is defined (e.g. in the "Defined as a class in" section for CDate, or in the "Defined as a function in" section for XmlEncode()). For serializable object classes (such as CDate) look for the base class definition. Follow a link to this source file.
2. Near the top of the LXR page for the source file is a path, and each component of the path links to another LXR page. Click the link to the last directory.
3. The resulting LXR page for the directory should list the makefile for the library of interest (e.g. Makefile.general.lib for CDate or Makefile.corelib.lib for XmlEncode ()). Click on the link to the makefile. You should see the LIB line with the name of the library that contains your symbol.
4. Add the library name to the list of libraries you already have and enter them into the library dependencies tool to create your final LIB and LIBS lines.

In some cases, the library name is a variant on the subdirectory name. These variants are summarized in Table 1.

Most often, difficulties arise when one is linking an application using the numerous "objects/" libraries. To give you some relief, here are some examples involving such libraries. They show the right order of libraries, as well as which libraries you may actually need. Using this as a starting point, it's **much** easier to find the right combination of libraries:

- first, to find and add missing libraries using the generic technique described above
- then, try to throw out libraries which you believe are not actually needed

```
LIB = id1 seqset $(SEQ_LIBS) pub medline biblio general \
 xser xconnect xutil xncbi
LIB = ncbimime cdd cn3d mmdb scoremat seqset $(SEQ_LIBS) \
 pub medline biblio general xser xutil xncbi
```

How do I add a library to a Visual C++ project?

If you are using Visual C++, you should add the appropriate LIB and LIBS lines to the Makefile.<your_project>.app file located in the source directory, then build the **-CONFIGURE-** target, then close and reopen the solution. This process will update the project properties with the proper search directories and required libraries.

Linker complains it "cannot find symbol" in something like: "SunWS_cache/CC_obj_b/bXmZkg3zX5VBJvYgjABX.o"

Go to the relevant build dir, clean and rebuild everything using:

```
cd /home/qqq/c++/WorkShop6-Debug/build/FooBar
make purge_r all_r
```

MAKE complains it does not know "how to make target: /home/qqq/c++/WorkShop6-Debug/lib/.seqset.dep"

This means that the "libseqset.a" library is not built. To build it:

```
cd /home/qqq/c++/WorkShop6-Debug/build/objects/seqset
make
```

Still getting bizarre errors with unresolved symbols, unfound libraries, etc., and nothing seems to help out much

As the last resort, try to CVS update, reconfigure, clean and rebuild everything:

```
cd /home/qqq/c++/
cvs -q upd -d
compilers/WorkShop6.sh 32 ........
make purge_r
make all_r
```

Debugging

Debugger (DBX) warns it "cannot find file /home/coremake/c++/foobar.cpp", then it does not show source code

This happens when you link to the public C++ Toolkit libraries (from "$NCBI/c++/*/lib/"), which are built on other hosts and thus hard-coded with the source paths on these other hosts.

All you have to do is to point DBX to the public sources (at "$NCBI/c++") by just adding to your DBX resource file (~/.dbxrc) the following lines:

```
pathmap /home/coremake/c++ /netopt/ncbi_tools/c++
pathmap /home/coremake/c++2 /netopt/ncbi_tools/c++
pathmap /home/coremake/c++3 /netopt/ncbi_tools/c++
pathmap /j/coremake/c++ /netopt/ncbi_tools/c++
pathmap /j/coremake/c++2 /netopt/ncbi_tools/c++
pathmap /j/coremake/c++3 /netopt/ncbi_tools/c++
```

ASN

Creating an out-of-tree application that uses your own local ASN.1 spec and a pre-built C++ Toolkit

Lets say you have your ASN.1 specification (call it foo.asn) and now you want to build an application (call its source code foo_main.cpp) which performs serialization of objects described in foo.asn. To complicate things, lets also assume that your ASN.1 spec depends on (imports) one of the ASN.1 specs already in the C++ Toolkit, like Date described in the NCBI-General module of general.asn. For example, your foo.asn could look like:

```
NCBI-Foo DEFINITIONS ::=
BEGIN
EXPORTS Foo;
IMPORTS Date FROM NCBI-General;
Foo ::= SEQUENCE {
 str VisibleString,
 date Date
}
END
```

Now, lets assume that the pre-built version of the NCBI C++ Toolkit is available at $NCBI/c++, and that you want to use the Toolkit's pre-built sources and libraries in your application. First, generate (using datatool) the serialization sources, and create the serialization library:

```
## Create new project directory, with a model makefile for your
## local ASN.1 serialization library, and copy "foo.asn"
cd ~/tmp
new_project foo lib/asn
cd foo
cp /bar/bar/bar/foo.asn .

## Using DATATOOL, generate data serialization sources for your
## ASN.1 specs described in "foo.asn":
datatool -oR $NCBI/c++ -m foo.asn \
 -M "objects/general/general.asn" -oA -oc foo -opc . -oph .

## Adjust in the library makefile "Makefile.foo.lib"
SRC = foo__ foo___

## Build the library
make -f Makefile.foo_lib
```

Then, create and build the application:

```
## Create new application project, and copy your app sources.
new_project foo_main app
cd foo_main
cp /bar/bar/bar/foo_main.cpp .

## Adjust the application makefile "Makefile.foo_main.app"
PRE_LIBS = -L.. -lfoo
CPPFLAGS = -I.. $(ORIG_CPPFLAGS)
LIB = general xser xutil xncbi

## Build the application
make -f Makefile.foo_main_app
```

How to add new ASN.1 specification to the C++ Toolkit?

Caution! If you are not in the C++ core developers group, please do not do it yourself! -- instead, just send your request to cpp-core@ncbi.nlm.nih.gov.

Converting ASN.1 object in memory from C to C++ representation (or vice versa)

The C++ Toolkit header ctools/asn_converter.hpp now provides a template class (CAsnConverter<>) for this exact purpose.

Useful Documentation Links

- [Doc] ISO/ANSI C++ Draft Standard Working Papers (Intranet only)
- [Doc] MSDN Online Search
- [Literature] Books and links to C++ and STL manuals
- [Example] NCBI C++ makefile hierarchy for project "corelib/"
- [Chart] NCBI C++ source tree hierarchy
- [Chart] NCBI C++ build tree hierarchy
- [Chart] NCBI C++ Library Dependency graph
- [Doc] NCBI IDX Database Documentation (Intranet only)
- [Doc] Documentation styles

Mailing Lists

- Announcements: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-announce (read-only)
- Everybody: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp
- Core developers: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-core
- Object Manager: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-objmgr
- GUI: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-gui
- SVN and CVS logs: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-cvs (read-only)

Internal mailing lists are also available to those inside NCBI.

Table 1. Examples where the library name is a variant on the sub directory name

Directory	**Library**
corelib/test	test_mt
corelib	xncbi
ctools/asn	xasn
cgi	xcgi or xfcgi
connect	xconnect
connect/test	xconntest
ctools	xctools
html	xhtml
objects/mmdb{1,2,3}	mmdb (consolidated)
objects/seq{,align,block,feat,loc,res}	seq (consolidated) or $(SEQ_LIBS)
objmgr	xobjmgr
objmgr/util	xobjutil
objtools/alnmgr	xalnmgr
serial	xser
util	xutil

Part 7: Library and Applications Configuration

Part 7 discusses configuration parameters for libraries and applications:

31: Library Configuration

Last Update: June 20, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter describes the run-time configuration parameters of the NCBI C++ Toolkit libraries. Such parameters change the default behavior of applications built using the Toolkit.

Configuration parameters can be set by environment variables, entered into a configuration file, defined by code, or any combination of those methods. Note: If a parameter is specified in both a configuration file and the environment, the environment takes precedence. The methods supported by each library and application are described below.

Chapter Outline

The following is an outline of the topics presented in this chapter:

Defining and Using Parameters

The following sections discuss the methods that libraries can use to define configuration parameters, and the corresponding methods that client applications can use to specify values for those parameters.

- CParam
- Registry
- Environment

CParam

Note: The preferred way for libraries to define their configuration parameters is with the macros in the CParam class (e.g. NCBI_PARAM_DECL). More details on the CParam class and its macros are presented in an earlier chapter. Libraries that use CParam can get configuration parameters using either the registry or the environment. Also, the CParam value can be stored and accessed on different levels: globally (application wide) and/or per-thread (TLS-like) and/or locally (cached within a CParam instance). Note that the name of an environment variable linked to a CParam can be customized or follow the default naming convention, so you have to look up the actual name used in the tables below before setting a configuration parameter using the environment.

Registry

If the CParam class cannot be used, the registry (configuration file) may be used to load, access, modify and store the values read from a configuration file. For libraries that use the registry, client applications can set the library configuration parameters using either the registry or the environment. In these cases the environment variable must follow the default naming convention.

These environment variables can be used to specify where to look for the registry.

The registry is case-insensitive for section and entry names. More details on the registry are presented in an earlier chapter.

Environment

For configuration parameters defined by either CParam or the registry, there is an equivalent environment variable having the form NCBI_CONFIG__<section>__<name> (note the double-underscores preceding <section> and <name>). The equivalent form is all uppercase.

Note: Environment variables may not contain dots (a.k.a. period or full stop) on many platforms. However, dots are allowed in registry section and entry names. The equivalent environment variable for parameters containing a dot in the section or entry name is formed by replacing the period with _DOT_. For example, the equivalent environment variable for [FastCGI]
WatchFile.Name is NCBI_CONFIG__FASTCGI__WATCHFILE_DOT_NAME.

Note: Environment variables are case-sensitive on many platforms. Therefore, when setting a configuration parameter via the environment, be sure to use the case shown in the tables below.

Some configuration parameters can only be set with an environment variable - for example, DIAG_SILENT_ABORT. In such cases, there is no corresponding registry entry.

Non-Specific Parameters

The following sections discuss configuration parameters that are not library-specific.

- Logging
- Diagnostic Trace
- Run-Time
- Abnormal Program Termination
- NCBI

Logging

The application log consists of diagnostic messages. Some of them are available only in debug builds. Others - namely, those produced by the ERR_POST or LOG_POST macros - can be redirected into a file. Normally, the name and location of the application log is specified using the logfile command-line argument.

These parameters tune the usage and behavior of the application log file.

Diagnostic Trace

These parameters tune the visibility and contents of diagnostic messages produced by _TRACE or ERR_POST macros.

See Table 3.

Run-Time

Run-time configuration parameters allow specifying memory size limit, CPU time limit, and memory allocation behavior. Note: not all operating systems support these parameters.

Abnormal Program Termination

These parameters specify how to handle abnormal situations when executing a program.

NCBI

These parameters tune generic NCBI C++ Toolkit-wide behavior.

Library-Specific Parameters

The following sections discuss library-specific configuration parameters.

- Connection
- NetCache and NetSchedule
- CGI and FCGI
- Serial
- Objects, Object Manager, Object Tools
- DBAPI
- Eutils

Connection

These parameters affect various aspects of internet connections established by the connection library. See the Networking and IPC chapter for a description of the corresponding network information structure.

CGI and FCGI

These parameters tune the behavior of CGI and FCGI applications and built with the NCBI C ++ Toolkit libraries. See Table 10 for CGI Load balancing configuration parameters.

Serial

These parameters tune the behavior of the Serial library.

Objects, Object Manager, Object Tools

These parameters tune the behavior of the Objects-related libraries, including the Object Manager and loader and reader libraries.

cSRA

sraread library

Note: This section applies only inside NCBI.

The following parameters tune the behavior of the sraread library:

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
If true, will add CIGAR info to Seq-align's returned by cSRA iterators.	[csra] cigar_in_align_ext CSRA_CIGAR_IN_ALIGN_EXT	Boolean	true
If true, will clip the read ranges returned by cSRA short read iterators according to quality.	[csra] clip_by_quality CSRA_CLIP_BY_QUALITY	Boolean	true
If true, will add mate info to Seq-align's returned by cSRA iterators.	[csra] explicit_mate_info CSRA_EXPLICIT_MATE_INFO	Boolean	false
If true, cSRA short read iterators will also include technical reads.	[csra] include_technical_reads CSRA_INCLUDE_TECHNICAL_READS	Boolean	true

ncbi_xloader_csra library

Note: This section applies only inside NCBI.

The following parameters tune the behavior of the ncbi_xloader_csra library:

Purpose	[Registry section] Registry name Environment variable	Valid values	Default
If >= 9, log alignment chunks. If >= 5, log major function calls. If >= 2, log refseq stats. If >= 1, log summary data.	[csra_loader] debug CSRA_LOADER_DEBUG	int	0
The max number of SRR files to keep open.	[csra_loader] gc_size CSRA_LOADER_GC_SIZE	size_t	10
If > 0, defines the max number of separate spot groups.	[csra_loader] max_separate_spot_groups CSRA_LOADER_MAX_SEPARATE_SPOT_GROUPS	int	0
If > 0, defines the minimum quality threshold for loading alignment and pileup chunks.	[csra_loader] pileup_graphs CSRA_LOADER_PILEUP_GRAPHS	int	0
If true, fetch quality graphs along with short reads.	[csra_loader] quality_graphs CSRA_LOADER_QUALITY_GRAPHS	Boolean	false

DBAPI

These parameters tune the behavior of the DBAPI library.

Eutils

These parameters tune the behavior of the Eutils library.

Application-Specific Parameters

The following sections discuss configuration parameters that are specific to selected applications.

- NetCache and NetSchedule
- Seqfetch.cgi

NetCache and NetSchedule

Note: This applies only inside NCBI.

Table 16 describes configuration parameters that are common to both NetCache and NetSchedule client APIs. These parameters are found in the netservice_api registry section.

Table 17 describes configuration parameters for NetCache client applications. These parameters are found in the netcache_api registry section. Note: The netcache_api registry section was formerly called netcache_client.

Table 18 describes configuration parameters for NetSchedule client applications. These parameters are found in the netschedule_api registry section.

See the Distributed Computing chapter for more information on NetCache and NetSchedule.

Seqfetch.cgi

Note: This applies only inside NCBI.

These parameters tune the behavior of the seqfetch.cgi application.

Table 1. Registry configuration parameters

Purpose	**Environment variable**	**Valid values**
If this variable is defined, the value is an extra-high-priority configuration file whose entries override those from other configuration files.	NCBI_CONFIG_OVERRIDES	a valid path
If this variable is defined, use it exclusively as the registry search path.	NCBI_CONFIG_PATH	a valid path
If this variable is **not** defined, append the current directory and home directory to the registry search path (after NCBI_CONFIG_PATH).	NCBI_DONT_USE_LOCAL_CONFIG	anything
If this variable is defined, append the value to the registry search path (after the home directory).	NCBI	a valid path
For Windows: If this variable is defined, append the value to the registry search path (after NCBI). For non-Windows, this variable is not checked and /etc is appended to the registry search path (after NCBI).	SYSTEMROOT	a valid path
If this variable is **not** defined, attempt to load a low-priority system-wide registry (ncbi.ini on Windows; .ncbirc on non-Windows). Note: the system-wide registry will not be loaded if it contains the DONT_USE_NCBIRC entry in the NCBI section.	NCBI_DONT_USE_NCBIRC	anything

Table 2. Log file configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Used by logging framework if the real client IP can not be obtained.	[LOG] Client_Ip NCBI_LOG_CLIENT_IP	a valid IPv4 or IPv6 address	""
Reset the log file to the specified file.	[LOG] File NCBI_CONFIG__LOG__FILE [c]	a valid file name	""
Specify when to use the File, NoCreate, Truncate, and TryRootLogFirst registry parameters shown in this table. Note: those parameters will only be used if the log file has not been set already or if IgnoreEnvArg is set to true.	[LOG] IgnoreEnvArg NCBI_CONFIG__LOG__IGNOREENVARG [c]	Boolean [a]	false
The listed environment variables will be logged as an 'extra' after each 'request-start' message. The extra message starts with a "LogEnvironment=true" pair.	[LOG] LogEnvironment DIAG_LOG_ENVIRONMENT [sic]	space separated list of environment variable names	""
The listed registry entries will be logged as an 'extra' after each 'request-start' message. The extra message starts with a "LogRegistry=true" pair.	[LOG] LogRegistry DIAG_LOG_REGISTRY [sic]	space separated list of registry section:name values	""
Do not create the log file if it does not exist already.	[Log] NoCreate NCBI_CONFIG__LOG__NOCREATE [c]	Boolean [b]	false
Specifies what to do if an invalid page hit ID is encountered. Valid PHIDs match the regex /[A-Za-z0-9:@_-]+(\.[0-9]+)*/.	[Log] On_Bad_Hit_Id LOG_ON_BAD_HIT_ID	"Allow", "AllowAndReport", "Ignore", "IgnoreAndReport", "Throw"	"AllowAndReport"
Specifies what to do if an invalid session ID is encountered. Valid session IDs match the format specified by LOG_SESSION_ID_FORMAT.	[Log] On_Bad_Session_Id LOG_ON_BAD_SESSION_ID	"Allow", "AllowAndReport", "Ignore", "IgnoreAndReport", "Throw"	"AllowAndReport"
Turn performance logging on or off (globally).	[Log] PerfLogging LOG_PerfLogging [c]	Boolean [b]	false
Defines the default session ID, which is used for any request which has no explicit session ID set.	[Log] Session_Id NCBI_LOG_SESSION_ID	any valid session ID string	""

Specifies which format rule to check session IDs against:
for "Ncbi" use ^[0-9]{16}_[0-9]{4,}SID$
for "Standard" use ^[A-Za-z0-9_.:@-]+$
for "Other" use ^.*$ (i.e. anything is valid).
	[Log]
	Session_Id_Format
	
	LOG_SESSION_ID_FORMAT
		"Ncbi", "Standard", "Other"	"Standard"

If set, a "log_site" entry with the given value will be added to request-start entries in log files.	[LOG] Site NCBI_LOG_SITE	a URL-encoded site name	(none)
If this parameter is defined, use the CSysLog facility setting when posting.	[LOG] SysLogFacility NCBI_CONFIG__LOG__SYSLOGFACILITY [c]	any non-empty string	(none)
Truncate the log file – i.e. discard the contents when opening an existing file.	[Log] Truncate LOG_TRUNCATE	Boolean [b]	false
Specify whether to try creating the log file under /log before trying other locations (e.g. a location specified by the registry or by NCBI_CONFIG__LOG__FILE).	[LOG] TryRootLogFirst NCBI_CONFIG__LOG__TRYROOTLOGFIRST [c]	Boolean [a]	false
If true, default to logging warnings when unsafe static array types are copied.	[NCBI] STATIC_ARRAY_COPY_WARNING NCBI_STATIC_ARRAY_COPY_WARNING	Boolean [b]	false
If true, log warnings for unsafe static array types.	[NCBI] STATIC_ARRAY_UNSAFE_TYPE_WARNING NCBI_STATIC_ARRAY_UNSAFE_TYPE_WARNING	Boolean [b]	true

[a] case-insensitive: true, t, yes, y, false, f, no, n

[b] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[c] environment variable name formed from registry section and entry name

Table 3. Diagnostic trace configuration parameters

Purpose	[Registry section] Registry name Environment variable	Valid values	Default
Specify the severity level threshold for posting diagnostic messages – i.e. less severe messages will not be posted. Note: If the parameter is set then the function ncbi::SetDiagPostLevel() is disabled - except for setting the level to eDiag_Trace.	[DEBUG] DIAG_POST_LEVEL DIAG_POST_LEVEL	CI [b]: Info, Warning, Error, Critical, Fatal, Trace	(none)
Diagnostic trace will be enabled if this parameter is given any value.	[DEBUG] DIAG_TRACE DIAG_TRACE or NCBI_CONFIG__DEBUG__DIAG_TRACE [c]	any non-empty string	(none)
Specify a file that stores a mapping of error codes to their descriptions.	[DEBUG] MessageFile NCBI_CONFIG__DEBUG__MessageFile [c]	a valid file name	(none)
Specify the maximum number of messages that can be posted to the AppLog within the AppLog period.	[Diag] AppLog_Rate_Limit DIAG_APPLOG_RATE_LIMIT	unsigned integer	50000
Specify the AppLog period in seconds.	[Diag] AppLog_Rate_Period DIAG_APPLOG_RATE_PERIOD	unsigned integer	10
Specify whether context properties should be automatically printed when set or changed.	[Diag] AutoWrite_Context DIAG_AUTOWRITE_CONTEXT	Boolean [a]	false
Specify the maximum number of diagnostic messages to collect. Messages beyond the limit will result in erasing the oldest message.	[Diag] Collect_Limit DIAG_COLLECT_LIMIT	size_t	1000
Specify the maximum number of messages that can be posted to the ErrLog within the ErrLog period.	[Diag] ErrLog_Rate_Limit DIAG_ERRLOG_RATE_LIMIT	unsigned integer	5000
Specify the ErrLog period in seconds.	[Diag] ErrLog_Rate_Period DIAG_ERRLOG_RATE_PERIOD	unsigned integer	1
Limit the log file size, and rotate the log when it reaches the limit.	[Diag] Log_Size_Limit DIAG_LOG_SIZE_LIMIT	non-negative long integer	0
Use the old output format if the flag is set.	[Diag] Old_Post_Format DIAG_OLD_POST_FORMAT	Boolean [a]	true

Specify a diagnostics post filter string (see an earlier chapter for more detail on filtering).	[DIAG] POST_FILTER NCBI_CONFIG__DIAG__POST_FILTER [c]	see the syntax rules	(none)
Print the system TID rather than CThread::GetSelf().	[Diag] Print_System_TID DIAG_PRINT_SYSTEM_TID	Boolean [a]	false
Defines the maximum number of entries to be listed in a stack trace. All stack trace entries above the specified level are not printed.	[DIAG] Stack_Trace_Max_Depth DEBUG_STACK_TRACE_MAX_DEPTH	a positive integer	200
Specify the minimum severity that will activate Tee_To_Stderr. See the Tee Output to STDERR section.	[Diag] Tee_Min_Severity DIAG_TEE_MIN_SEVERITY	CI [b]: Info, Warning, Error, Critical, Fatal, Trace	Warning (debug); Error (release)
Duplicate messages to stderr. See the Tee Output to STDERR section.	[Diag] Tee_To_Stderr DIAG_TEE_TO_STDERR	Boolean [a]	false
Specify a diagnostics trace filter string (see an earlier chapter for more detail on filtering).	[DIAG] TRACE_FILTER NCBI_CONFIG__DIAG__TRACE_FILTER [c]	see the syntax rules	(none)
Specify the maximum number of messages that can be posted to the TraceLog within the TraceLog period.	[Diag] TraceLog_Rate_Limit DIAG_TRACELOG_RATE_LIMIT	unsigned integer	5000
Specify the TraceLog period in seconds.	[Diag] TraceLog_Rate_Period DIAG_TRACELOG_RATE_PERIOD	unsigned integer	1
If true and AppLog severity is not locked, print the current GMT time in diagnostic messages; otherwise print local time.	[Diag] UTC_Timestamp DIAG_UTC_TIMESTAMP	Boolean [a]	false

[a] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[b] CI = case-insensitive

[c] environment variable name formed from registry section and entry name

Table 4. Run-time configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Set a CPU time limit for the application in seconds.	[NCBI] CpuTimeLimit NCBI_CONFIG__NCBI__CPUTIMELIMIT [b]	non-negative integer	0 (unlimited)
Set a memory size limit for the application in MB or as a percent of total system memory.	[NCBI] MemorySizeLimit NCBI_CONFIG__NCBI__MEMORYSIZELIMIT [b]	non-negative integer or percent, for example "70" (for 70 MB) or "70%" (for 70% of all memory)	0 (unlimited)
Specify the method for filling allocated memory.	[NCBI] MEMORY_FILL NCBI_MEMORY_FILL	CI [a]: none, zero, pattern	pattern

[a] CI = case-insensitive

[b] environment variable name formed from registry section and entry name

Table 5. Abnormal program termination configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
If this parameter is defined, abort the program if a CException is thrown.	[DEBUG] ABORT_ON_THROW NCBI_CONFIG__DEBUG__ABORT_ON_THROW [c]	any non-empty string	(none)
Specify whether the NCBI application framework should catch exceptions that are not otherwise caught.	[Debug] Catch_Unhandled_Exceptions DEBUG_CATCH_UNHANDLED_EXCEPTIONS	Boolean [a]	true
Specify whether ncbi::Abort() will call _ASSERT (false). Note: this only applies to MSVC.	[Diag] Assert_On_Abort DIAG_ASSERT_ON_ABORT	Boolean [a]	false
If this parameter is true, abort the program if a CObjectException is thrown.	[NCBI] ABORT_ON_COBJECT_THROW NCBI_ABORT_ON_COBJECT_THROW	Boolean [a]	false
If this parameter is true, abort the program on an attempt to access or release a NULL pointer stored in a CRef object.	[NCBI] ABORT_ON_NULL NCBI_ABORT_ON_NULL	Boolean [a]	false
Specify what to do when ncbi::Abort() is called. When the variable is set to a "yes" value, Abort() will call exit(255). When the variable is set to a "no" value, Abort() will call abort(). When the variable is not set, Abort() will call exit(255) for release builds and abort() for debug builds - unless compiled with MSVC and the DIAG_ASSERT_ON_ABORT parameter is true, in which case Abort() will call _ASSERT(false).	[N/A] N/A DIAG_SILENT_ABORT	Boolean [b]	(none)

[a] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[b] case-insensitive: y, 1, n, 0

[c] environment variable name formed from registry section and entry name

Table 6. NCBI C++ Toolkit-wide configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Specify whether throwing an exception of at least Critical severity will cause an immediate abort().	[EXCEPTION] Abort_If_Critical EXCEPTION_ABORT_IF_CRITICAL	Boolean [a]	false
Specify the minimum severity that will result in the stack trace being added to exceptions.	[EXCEPTION] Stack_Trace_Level EXCEPTION_STACK_TRACE_LEVEL	CI [b]: Trace, Info, Warning, Error, Critical, Fatal	Critical
A single path to check for common data files via g_FindDataFile(). Takes a lower precedence than paths in NCBI_DATA_PATH.	[NCBI] Data NCBI_CONFIG__NCBI__DATA [c]	a valid path	""
A list of paths (delimited in the style of the OS) to check for common data files via g_FindDataFile().	[NCBI] DataPath NCBI_DATA_PATH	a delimited list of valid paths	""
Specify how read-only files are treated on Windows during a remove request.	[NCBI] DeleteReadOnlyFiles NCBI_CONFIG__DELETEREADONLYFILES	Boolean [a]	false
Specify whether the API classes should have logging turned on.	[NCBI] FileAPILogging NCBI_CONFIG__FILEAPILOGGING	Boolean [a]	DEFAULT_LOGGING_VALUE
Declare how umask settings on Unix affect creating files/ directories in the File API.	[NCBI] FileAPIHonorUmask NCBI_CONFIG__FileAPIHonorUmask	Boolean [a]	false
Specify whether to load plugins from DLLs.	[NCBI] Load_Plugins_From_DLLs NCBI_LOAD_PLUGINS_FROM_DLLS	Boolean [a]	LOAD_PLUGINS_FROM_DLLS_BY_DEFAULT
Specify the directory to use for temporary files.	[NCBI] TmpDir NCBI_CONFIG__NCBI__TMPDIR [c]	a valid path	""
Specify the file name of a Unicode-to-ASCII translation table.	[NCBI] UnicodeToAscii NCBI_CONFIG__NCBI__UNICODETOASCII [c]	a valid path	""

[a] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[b] CI = case-insensitive

[c] environment variable name formed from registry section and entry name

Table 7. Connection library configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable (See *Note 2*)**	**Valid values**	**Default**
Service-specific parameters follow this form. **(See *Note 1*)**	**[<service>]** **CONN_<param_name>** **<service>_CONN_<param_name>**		
Global parameters follow this form. **(See *Note 1*)**	**[CONN]** **<param_name>** **CONN_<param_name>**		
Specify arguments for the given service. (See *Note 1*)	[<service>] CONN_ARGS <service>_CONN_ARGS	(service-dependent)	""
Specify how much debug information will be output. (See *Note 1*)	[<service>] CONN_DEBUG_PRINTOUT <service>_CONN_DEBUG_PRINTOUT	CI [a]: *to get some*: 1, on, yes, true, some *to get all*: data, all *to get none*: anything else	""
If this parameter is true, the network dispatcher will be disabled. (See *Note 1*)	[<service>] CONN_DISPD_DISABLE <service>_CONN_DISPD_DISABLE	Boolean [c]	true
If this parameter is true, the Firewall mode will be enabled. (See *Note 1*)	[<service>] CONN_FIREWALL <service>_CONN_FIREWALL	Boolean [c]	not set
Set the dispatcher host name. (See *Note 1*)	[<service>] CONN_HOST <service>_CONN_HOST	a valid host name	www.ncbi.nlm.nih.g
Set the HTTP proxy server. (See *Note 1*)	[<service>] CONN_HTTP_PROXY_HOST <service>_CONN_HTTP_PROXY_HOST	a valid proxy host	""
Set the HTTP proxy server port number. This will be set to zero if <service>_CONN_HTTP_PROXY_HOST is not set. (See *Note 1*)	[<service>] CONN_HTTP_PROXY_PORT <service>_CONN_HTTP_PROXY_PORT	unsigned short	0
Set a custom user header. This is rarely used, and then typically for debugging purposes. (See *Note 1*)	[<service>] CONN_HTTP_USER_HEADER <service>_CONN_HTTP_USER_HEADER	a valid HTTP header	""
Prohibit the use of a local load balancer. Note: This parameter is discouraged for performance reasons - please use <service>_CONN_LBSMD_DISABLE instead. (See *Note 1*)	[<service>] CONN_LB_DISABLE <service>_CONN_LB_DISABLE	Boolean [c]	false

Prohibit the use of a local load balancer. This should be used instead of <service>_CONN_LB_DISABLE. (See *Note 1*)	[<service>] CONN_LBSMD_DISABLE <service>_CONN_LBSMD_DISABLE	Boolean [c]	false
Enable the use of locally configured services. See <service>_CONN_LOCAL_SERVER_<n>. (See *Note 1*)	[<service>] CONN_LOCAL_ENABLE <service>_CONN_LOCAL_ENABLE	Boolean [c]	false
Create a service entry for service, where n is a number from 0 to 100 (not necessarily sequential). The value must be a valid server descriptor, as it would be configured for the load balancing daemon (LBSMD). This is a quick way of configuring locally used services (usually, for the sole purposes of debugging / development) without the need to edit the actual LBSMD tables (which become visible for the whole NCBI). See <service>_CONN_LOCAL_ENABLE. Not e: This parameter has no corresponding global parameter. (See *Note 1*)	[<service>] CONN_LOCAL_SERVER_<n> <service>_CONN_LOCAL_SERVER_<n>	any non-empty string	not set
Maximum number of attempts to establish connection. Zero means use the default. (See *Note 1*)	[<service>] CONN_MAX_TRY <service>_CONN_MAX_TRY	unsigned short	3
Specify a password for the connection (only used with <service>_CONN_USER). (See *Note 1*)	[<service>] CONN_PASS <service>_CONN_PASS	the user's password	""
Set the path to the service. (See *Note 1*)	[<service>] CONN_PATH <service>_CONN_PATH	a valid service path	/Service/dispd.cgi
Set the dispatcher port number. (See *Note 1*)	[<service>] CONN_PORT <service>_CONN_PORT	unsigned short	0
Set a non-transparent CERN-like firewall proxy server. (See *Note 1*)	[<service>] CONN_PROXY_HOST <service>_CONN_PROXY_HOST	a valid proxy host	""
Set the HTTP request method. (See *Note 1*)	[<service>] CONN_REQ_METHOD <service>_CONN_REQ_METHOD	CI [a]: any, get, post	ANY
Redirect connections to <service> to the specified alternative service. See Service Redirection. (See *Note 1*)	[<service>] CONN_SERVICE_NAME <service>_CONN_SERVICE_NAME	a replacement for the service name	(none)
Set to true if the client is stateless. (See *Note 1*)	[<service>] CONN_STATELESS <service>_CONN_STATELESS	Boolean [c]	false

Zero means no waiting but polling (may not work well with all connections); "infinite" means no timeout (i.e. to wait for I/O indefinitely); other values are the maximum number of seconds to wait before failing. (See *Note 1*.)	[<service>] CONN_TIMEOUT <service>_CONN_TIMEOUT	floating point >= 0.0 (1 microsecond precision) [f] or "infinite"	30.0
Specify a username for the connection (see <service>_CONN_PASS). Only necessary for connections requiring authentication. (See *Note 1*)	[<service>] CONN_USER <service>_CONN_USER	a username with access rights for the connection	(none)
Set the level of logging detail that GNUTLS should produce about secure transactions. Log levels greater than 7 also dump scrambled data from GNUTLS.	[CONN] GNUTLS_LOGLEVEL CONN_GNUTLS_LOGLEVEL	0 to 10	0
A true value enables HTTP connections to dump headers of error server responses only (successful responses do not get logged).	[CONN] HTTP_ERROR_HEADER_ONLY CONN_HTTP_ERROR_HEADER_ONLY	Boolean [c]	false
A true value enables HTTP connections to follow https to http transitions (http to https transitions are secure and therefore don't need to be enabled).	[CONN] HTTP_INSECURE_REDIRECT CONN_HTTP_INSECURE_REDIRECT	Boolean [c]	false
Set a default referer (applies to all HTTP connections).	[CONN] HTTP_REFERER CONN_HTTP_REFERER	a valid referer	(none)
A list of identifiers to be treated as local services defined in the registry / environment. This parameter is optional and is used only for reverse address-to-name lookups.	[CONN] LOCAL_SERVICES CONN_LOCAL_SERVICES	whitespace-delimited [d] list of identifiers	(none)
Set the mail gateway host.	[CONN] MX_HOST CONN_MX_HOST	a valid host name	localhost on UNIX platforms except Cygwin; mailgw on other platforms
Set the mail gateway port.	[CONN] MX_PORT CONN_MX_PORT	1 to 65535	25 (SMTP)
Set the mail gateway communication timeout in seconds.	[CONN] MX_TIMEOUT CONN_MX_TIMEOUT	floating point >= 0.0 (zero means default)	120
Enable CServer to catch exceptions.	[server] Catch_Unhandled_Exceptions CSERVER_CATCH_UNHANDLED_EXCEPTIONS	Boolean [b]	true
Deprecated.	[server] allow_implicit_job_return NCBI_CONFIG__SERVER__ALLOW_IMPLICIT_JOB_RETURN [e]	Boolean [b]	false
Maximum time worker nodes are allowed to live without a single NetSchedule server.	[server] max_wait_for_servers NCBI_CONFIG__SERVER__MAX_WAIT_FOR_SERVERS [e]	unsigned int	24 * 60 * 60 second

Causes the worker node to shut down if any jobs fail.	[server] stop_on_job_errors NCBI_CONFIG__SERVER__STOP_ON_JOB_ERRORS [e]	Boolean [b]	true
Enable CThreadInPool_ForServer to catch exceptions.	[ThreadPool] Catch_Unhandled_Exceptions NCBI_CONFIG__THREADPOOL__CATCH_UNHANDLED_EXCEPTIONS [e]	Boolean [b]	true

[a] CI = case-insensitive

[b] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[c] case-insensitive: true values are { 1, on, yes, true }; false is anything else

[d] whitespace can be any number of spaces and/or tabs

[e] environment variable name formed from registry section and entry name

[f] although very precise values may be specified, practical host limitations my result in less precise effective values

Note 1: All service-specific parameters shown in Table 7 (except one) have corresponding global parameters - i.e. parameters that apply to all services. For these global parameters, the registry section name is CONN; the registry entry name doesn't have the CONN_ prefix; and the environment variable doesn't have the <service>_ prefix. For example, the service-specific parameter specified by the CONN_ARGS entry in a given [<service>] section of the registry (or by the <service>_CONN_ARGS environment variable) corresponds to the global parameter specified by the ARGS entry in the [CONN] section of the registry (or by the CONN_ARGS environment variable). When both a service-specific parameter and its corresponding global parameter are set, the service-specific parameter takes precedence.

Note 2: Environment variable names for service-specific parameters are formed by capitalizing the service name.

Table 8. CGI-related configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Set to the user agent string you would like to be used by the web server.	[N/A] N/A HTTP_USER_AGENT	A valid user agent string.	(none)
Add to the user agent list of bot names. This parameter affect only CCgiUserAgent::IsBot().	[CGI] Bots NCBI_CONFIG__CGI__BOTS[f]	Delimited list [b] of bot names, e.g. "Googlebot Scooter WebCrawler Slurp".	(none)
According to RFC-2109, cookies should not be encoded. Instead, they should be just quoted. However, for backward compatibility with code that decodes incoming cookies, both quoted cookies and encoded cookies can be parsed. This setting controls which method of encoding/decoding is used.	[CGI] Cookie_Encoding CGI_COOKIE_ENCODING	"Url", "Quote"	"Url"
Severity level for cookie-related error messages.	[CGI] Cookie_Error_Severity CGI_Cookie_Error_Severity	CI [e]: Info, Warning, Error, Critical, Fatal, Trace	Error
Defines which characters cannot be used in cookie names.	[CGI] Cookie_Name_Banned_Symbols CGI_Cookie_Name_Banned_Symbols	A string of banned characters.	" ,;="
Set to true to make the application count the amount of data read/ sent. The numbers are then printed in request stop log messages.	[CGI] Count_Transfered CGI_COUNT_TRANSFERED	Boolean [c]	true
Set the name of an environment variable, which in turn specifies a prefix that will be added to all diagnostic messages issued during HTTP request processing.	[CGI] DiagPrefixEnv NCBI_CONFIG__CGI__DIAGPREFIXENV[f]	a valid environment variable name	(none)
Set to true to disable the creation of a tracking cookie during session initialization.	[CGI] DisableTrackingCookie NCBI_CONFIG__CGI__DISABLETRACKINGCOOKIE[f]	Boolean [c]	false

Set to true to enable logging.	[CGI]
	Log
	
	NCBI_CONFIG__CGI__LOG[f]
		CI [e]:
		On => enabled;
		True => enabled;
		OnError => enabled for errors;
		OnDebug => enabled (debug builds only)
			disabled

An ampersand-delimited string of GET and/or POST arguments to exclude from the log (helps limit the size of the log file)	[CGI] LOG_EXCLUDE_ARGS CGI_LOG_EXCLUDE_ARGS	valid format: arg1&arg2...	(none)

Allows specifying limits for multiple GET and/or POST arguments in one parameter string.	[CGI] LOG_LIMIT_ARGS CGI_LOG_LIMIT_ARGS	valid format: arg1:size1&arg2:size2...&*:size special argument: * means all unspecified arguments; special limits: -2 means exclude; -1 means no limit	*:1000000
Enable logging of CGI request parameters. Only the specified parameters will be logged.	[CGI] LogArgs NCBI_CONFIG__CGI__LOGARGS[f]	Delimited list [b] of environment variables (optionally aliased on output for shortening logs, e.g. envvar=1).	(none)
Set to true to merge log lines.	[CGI] Merge_Log_Lines CGI_MERGE_LOG_LINES	Boolean [c]	true
Specify additional mobile device names. This parameter affect only CCgiUserAgent::IsMobileDevice ().	[CGI] MobileDevices NCBI_CONFIG__CGI__MobileDevices[f]	Delimited list [b] of additional device names.	(none)
Add to the user agent list of names that aren't bots. This parameter affect only CCgiUserAgent::IsBot().	[CGI] NotBots NCBI_CONFIG__CGI__NotBots[f]	Delimited list [b] of names that aren't bots.	(none)
Add to the user agent list of names that aren't mobile devices. This parameter affect only CCgiUserAgent::IsMobileDevice ().	[CGI] NotMobileDevices NCBI_CONFIG__CGI__NotMobileDevices[f]	Delimited list [b] of names that aren't mobile devices.	(none)
Add to the user agent list of names that aren't phone devices. This parameter affect only CCgiUserAgent::IsPhoneDevice ().	[CGI] NotPhoneDevices NCBI_CONFIG__CGI__NotPhoneDevices[f]	Delimited list [b] of names that aren't phone devices.	(none)
Add to the user agent list of names that aren't tablet devices. This parameter affect only CCgiUserAgent::IsTabletDevice ().	[CGI] NotTabletDevices NCBI_CONFIG__CGI__NotTabletDevices[f]	Delimited list [b] of names that aren't tablet devices.	(none)
Control error handling of incoming cookies (doesn't affect outgoing cookies set by application).	[CGI] On_Bad_Cookie CGI_ON_BAD_COOKIE	CI [e]: Throw, SkipAndError, Skip, StoreAndError, Store	Store
Specify additional phone device names. This parameter affect only CCgiUserAgent::IsPhoneDevice ().	[CGI] PhoneDevices NCBI_CONFIG__CGI__PhoneDevices[f]	Delimited list [b] of additional device names.	(none)
Specifies whether to print the referer during LogRequest().	[CGI] Print_Http_Referer CGI_PRINT_HTTP_REFERER	Boolean [c]	true
Specifies whether to print the URL during LogRequest().	[CGI] Print_Self_Url CGI_PRINT_SELF_URL	Boolean [c]	true

Specifies whether to print the user agent during LogRequest().	[CGI] Print_User_Agent CGI_PRINT_USER_AGENT	Boolean [c]	true
Set the size of CGI request buffer that is printed when the request cannot be parsed.	[CGI] RequestErrBufSize NCBI_CONFIG__CGI__REQUESTERRBUFSIZE[f]	buffer size in bytes	256
Specify the registry section name for the result cache.	[CGI] ResultCacheSectionName NCBI_CONFIG__CGI__RESULTCACHESECTIONNAME[f]	valid section name	result_cache
Enable statistics logging.	[CGI] StatLog NCBI_CONFIG__CGI__STATLOG[f]	Boolean [d]	false
Specify additional tablet device names. This parameter affect only CCgiUserAgent::IsTabletDevice().	[CGI] TabletDevices NCBI_CONFIG__CGI__TabletDevices[f]	Delimited list [b] of additional device names.	(none)
Controls whether the output stream will throw for bad states.	[CGI] ThrowOnBadOutput NCBI_CONFIG__CGI__THROWONBADOUTPUT[f]	Boolean [c]	true
Log start time, end time, and elapsed time.	[CGI] TimeStamp NCBI_CONFIG__CGI__TIMESTAMP[f]	Boolean [d]	false
Disable statistics logging if the CGI request took less than the specified number of seconds.	[CGI] TimeStatCutOff NCBI_CONFIG__CGI__TIMESTATCUTOFF[f]	non-negative integer (zero enables logging)	0
Specify the domain for the tracking cookie.	[CGI] TrackingCookieDomain NCBI_CONFIG__CGI__TRACKINGCOOKIEDOMAIN[f]	valid domain	.nih.gov
Specify the tracking cookie name.	[CGI] TrackingCookieName NCBI_CONFIG__CGI__TRACKINGCOOKIENAME[f]	valid cookie name	ncbi_sid
Specify the path for the tracking cookie.	[CGI] TrackingCookiePath NCBI_CONFIG__CGI__TRACKINGCOOKIEPATH[f]	valid path	/
Defines the **name** of the NCBI tracking cookie (session ID cookie).	[CGI] TrackingTagName CGI_TrackingTagName	Any valid cookie name.	"NCBI-SID"

[a] List may be delimited by semicolon, space, tab, or comma.

[b] List may be delimited by semicolon, space, tab, vertical bar, or tilde.

[c] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[d] case-insensitive: true, t, yes, y, false, f, no, n

[e] CI = case-insensitive

[f] environment variable name formed from registry section and entry name

Table 9. FCGI-related configuration parameters

Purpose	[Registry section] Registry name Environment variable	Valid values	Default
A true value enables logging of current iteration, max iterations, and process ID during the FastCGI run.	[FastCGI] Debug NCBI_CONFIG__FASTCGI__DEBUG [b]	Boolean [a]	false
A true value enables termination of a FastCGI application by the presence of the request entry "exitfastcgi".	[FastCGI] HonorExitRequest NCBI_CONFIG__FASTCGI__HONOREXITREQUEST [b]	Boolean [a]	false
Specify the number of requests that the FCGI application will process before exiting.	[FastCGI] Iterations NCBI_CONFIG__FASTCGI__ITERATIONS [b]	positive integer	10
Make the FastCGI application run as a stand-alone server on a local port. The value is a UNIX domain socket or a MS Windows named pipe, or a colon followed by a port number	[FastCGI] StandaloneServer FCGI_STANDALONE_SERVER	valid local port or named socket	(none)
Make the FastCGI application stop if an error is encountered.	[FastCGI] StopIfFailed NCBI_CONFIG__FASTCGI__STOPIFFAILED [b]	Boolean [a]	false
Make the FastCGI application exit if the named file changes.	[FastCGI] WatchFile.Name NCBI_CONFIG__FASTCGI__WATCHFILE_DOT_NAME [b]	valid file name	(none)
The number of bytes to read from the watch file to see if it has changed.	[FastCGI] WatchFile.Limit NCBI_CONFIG__FASTCGI__WATCHFILE_DOT_LIMIT [b]	positive integer (non-positives trigger default)	1024
The period in seconds between checking the watch file for changes.	[FastCGI] WatchFile.Timeout NCBI_CONFIG__FASTCGI__WATCHFILE_DOT_TIMEOUT [b]	positive integer (non-positives trigger default, which is to disable the watch file checking)	0

[a] case-insensitive: true, t, yes, y, false, f, no, n

[b] environment variable name formed from registry section and entry name

Table 10. CGI Load balancing configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Specify the internet domain.	[CGI-LB] Domain NCBI_CONFIG__CGI-LB__DOMAIN [b]	a valid domain	.ncbi.nlm.nih.gov
Specify the host IP address.	[CGI-LB] Host NCBI_CONFIG__CGI-LB__HOST [b]	a valid host IP	(none)
Specify the cookie expiration period in seconds.	[CGI-LB] LifeSpan NCBI_CONFIG__CGI-LB__LIFESPAN [b]	integer	0
Specify the name of the load balancing cookie in the HTTP response.	[CGI-LB] Name NCBI_CONFIG__CGI-LB__NAME [b]	a valid cookie name	(none)
Specify the cookie path.	[CGI-LB] Path NCBI_CONFIG__CGI-LB__PATH [b]	a valid path	(none)
Specify the cookie security mode.	[CGI-LB] Secure NCBI_CONFIG__CGI-LB__SECURE [b]	Boolean [a]	false

[a] case-insensitive: true, t, yes, y, false, f, no, n

[b] environment variable name formed from registry section and entry name

Table 11. Serial library configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Skip unknown data members in the input stream, or throw an exception.	[N/A] N/A SERIAL_SKIP_UNKNOWN_MEMBERS	CI [a]: yes, no, never, always	no (throw)
If true, causes CObjectOStream::WriteDouble () to use fast conversion.	[SERIAL] FastWriteDouble NCBI_CONFIG__SERIAL__FastWriteDouble [b]	Boolean [c]	true
While reading binary ASN.1 data allow VisibleString tag where UTF-8 string tag is expected by specification.	[SERIAL] READ_ANY_UTF8STRING_TAG SERIAL_READ_ANY_UTF8STRING_TAG	Boolean [c]	true
While reading binary ASN.1 data allow UTF-8 string tag where VisibleString tag is expected by specification.	[SERIAL] READ_ANY_VISIBLESTRING_TAG SERIAL_READ_ANY_VISIBLESTRING_TAG	0 (disallow, throws an exception); 1 (allow, but warn once); 2 (allow without warning)	1
Specify how to handle unknown variants when reading Object streams.	[SERIAL] SKIP_UNKNOWN_MEMBERS NCBI_CONFIG__SERIAL__SKIP_UNKNOWN_MEMBERS [b]	CI [a]: no (throw an exception), never (even if set to skip later), yes (skip), always (even if set to not skip later)	no
Specify how to handle unknown variants when reading Object streams.	[SERIAL] SKIP_UNKNOWN_VARIANTS NCBI_CONFIG__SERIAL__SKIP_UNKNOWN_VARIANTS [b]	CI [a]: no (throw an exception), never (even if set to skip later), yes (skip), always (even if set to not skip later)	no
Throw an exception on an attempt to access an uninitialized data member.	[SERIAL] VERIFY_DATA_GET SERIAL_VERIFY_DATA_GET	CI [a]: yes, no, never, always, defvalue, defvalue_always	yes
Throw an exception if a mandatory data member is missing in the input stream.	[SERIAL] VERIFY_DATA_READ SERIAL_VERIFY_DATA_READ	CI [a]: yes, no, never, always, defvalue, defvalue_always	yes
Throw an exception on an attempt to write an uninitialized data member.	[SERIAL] VERIFY_DATA_WRITE SERIAL_VERIFY_DATA_WRITE	CI [a]: yes, no, never, always, defvalue, defvalue_always	yes
While writing binary ASN.1 data issue UTF8 string tag as determined by specification, otherwise issue plain string tag.	[SERIAL] WRITE_UTF8STRING_TAG SERIAL_WRITE_UTF8STRING_TAG	Boolean [c]	false
Specifies what to do if an invalid character is read.	[SERIAL] WRONG_CHARS_READ NCBI_CONFIG__SERIAL__WRONG_CHARS_READ [b]	"ALLOW", "REPLACE", "REPLACE_AND_WARN", "THROW", "ABORT"	"REPLACE_AND_WARN"

Specifies what to do if an invalid character is written.	[SERIAL] WRONG_CHARS_WRITE NCBI_CONFIG__SERIAL__WRONG_CHARS_WRITE [b]	"ALLOW", "REPLACE", "REPLACE_AND_WARN", "THROW", "ABORT"	"REPLACE_AND_WARN"

[a] CI = case-insensitive

[b] environment variable name formed from registry section and entry name

[c] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

Table 13. Objects-related configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
The Object Manager will attach WGS master descriptors to Bioseq data by default. Setting this parameter to false will disable this behavior.	[GENBANK] ADD_WGS_MASTER GENBANK_ADD_WGS_MASTER	Boolean [a]	true
A non-zero value turns on debugging messages about GenBank loader's interaction with cache.	[GENBANK] CACHE_DEBUG GENBANK_CACHE_DEBUG	>=0, currently only zero and non-zero are distinguished	0
Specify whether an attempt should be made to recompress the cache.	[GENBANK] CACHE_RECOMPRESS GENBANK_CACHE_RECOMPRESS	Boolean [a]	true
A non-zero value turns on debugging messages about opening/closing connections to ID1/ID2 services.	[GENBANK] CONN_DEBUG GENBANK_CONN_DEBUG	>=0, currently only zero and non-zero are distinguished	0
Disable attaching WGS master descriptors when retrieving ASN.1 blobs using the CPubseqReader and CPubseq2Reader classes.	[GENBANK/PUBSEQOS] or [GENBANK/PUBSEQOS2] EXCLUDE_WGS_MASTER NCBI_CONFIG__GENBANK_PUBSEQOS__EXCLUDE_WGS_MASTER or NCBI_CONFIG__GENBANK_PUBSEQOS2__EXCLUDE_WGS_MASTER	Boolean [b]	false
Set the severity level for ID1 debug tracing.	[GENBANK] ID1_DEBUG GENBANK_ID1_DEBUG	int: 0 = none, 1 = error, 2 = open, 4 = conn, 5 = asn, 8 = asn data	0
Specify the ID1 reader service name. Not e: The services can be redirected using generic Service Redirection technique.	In priority order: [GENBANK] ID1_SERVICE_NAME, [NCBI] SERVICE_NAME_ID1 In priority order: GENBANK_ID1_SERVICE_NAME, GENBANK_SERVICE_NAME_ID1	a valid reader service name	ID1 (see API)
Specify the ID2 reader service name. Not e: The services can be redirected using generic Service Redirection technique.	In priority order: [GENBANK] ID2_CGI_NAME, [GENBANK] ID2_SERVICE_NAME, [NCBI] SERVICE_NAME_ID2 In priority order: GENBANK_ID2_CGI_NAME, GENBANK_ID2_SERVICE_NAME, GENBANK_SERVICE_NAME_ID2	a valid reader service name	ID2 (see API)

Set the severity level for ID2 debug tracing.	[GENBANK] ID2_DEBUG GENBANK_ID2_DEBUG	int: 0 = none, 1 = error, 2 = open, 4 = conn, 5 = asn, 8 = blob, 9 = blob data	debug: none release: error (see API)
Number of chunks allowed in a single request.	[GENBANK] ID2_MAX_CHUNKS_REQUEST_SIZE GENBANK_ID2_MAX_CHUNKS_REQUEST_SIZE	int: 0 = unlimited request size; 1 = do not use packets or get-chunks requests	100
Maximum number of requests packed in a single ID2 packet.	[GENBANK] ID2_MAX_IDS_REQUEST_SIZE GENBANK_ID2_MAX_IDS_REQUEST_SIZE	>=0	100
The maximum number of connections the reader can establish to the data source. This is run-time limited to 1 for single threaded clients and for all clients using the cache or gi reader, and to 5 for multi-threaded clients using the id1, id2, pubseqos, and pubseqos2 readers.	[GENBANK] MAX_NUMBER_OF_CONNECTIONS	int	3 for id1 and id2; 2 for pubseqos and pubseqos2
See MAX_NUMBER_OF_CONNECTIONS	[GENBANK] NO_CONN		
See OPEN_TIMEOUT_INCREMENT	[GENBANK] OPEN_INCREMENT		
See OPEN_TIMEOUT_MAX	[GENBANK] OPEN_MAX		
See OPEN_TIMEOUT_MULTIPLIER	[GENBANK] OPEN_MULTIPLIER		
The OPEN_TIMEOUT* parameters describe the timeout for opening a GenBank connection. The timeout allows the server a reasonable time to respond while providing a means to quickly abandon unresponsive servers.	[GENBANK] OPEN_TIMEOUT NCBI_CONFIG__GENBANK__OPEN_TIMEOUT [c]	any floating point value >= 0.0	5 seconds
OPEN_TIMEOUT_MULTIPLIER and OPEN_TIMEOUT_INCREMENT specify the way the open timeout is increased if no response is received (next_open_timeout = prev_open_timeout * multiplier + increment).	[GENBANK] OPEN_TIMEOUT_INCREMENT NCBI_CONFIG__GENBANK__OPEN_TIMEOUT_INCREMENT [c]	any floating point value >= 0.0	0 seconds
The limit of increasing the open timeout using OPEN_TIMEOUT_MULTIPLIER and OPEN_TIMEOUT_INCREMENT.	[GENBANK] OPEN_TIMEOUT_MAX NCBI_CONFIG__GENBANK__OPEN_TIMEOUT_MAX [c]	floating point >= 0.0	30 seconds
See OPEN_TIMEOUT_INCREMENT	[GENBANK] OPEN_TIMEOUT_MULTIPLIER NCBI_CONFIG__GENBANK__OPEN_TIMEOUT_MULTIPLIER [c]	floating point >= 0.0	1.5

Turns on different levels of debug messages in PubSeqOS reader. A value >=2 means debug opening connections while >=5 means debug results of Seq-id resolution requests. Note: only applies to debug builds.	[GENBANK] PUBSEQOS_DEBUG GENBANK_PUBSEQOS_DEBUG	int	0
Whether to open first connection immediately or not.	[GENBANK] preopen NCBI_CONFIG__GENBANK__PREOPEN [c]	Boolean [b]	true
Specify the level of reader statistics to collect.	[GENBANK] READER_STATS GENBANK_READER_STATS	int: 0 = none, 1 = verbose	0
Prioritized list of drivers to try for the reader.	Sources searched for list: [GENBANK] ReaderName, [GENBANK] LOADER_METHOD, default Sources searched for list: GENBANK_LOADER_METHOD, default	list items are semicolon-delimited; each item is a colon-delimited list of drivers. valid drivers: id1, id2, cache, pubseqos	"ID2:PUBSEQOS:ID1", or "ID2:ID1" (see API)
Specify whether the reader manager should automatically register ID1, ID2, and cache.	[GENBANK] REGISTER_READERS GENBANK_REGISTER_READERS	Boolean [a]	true
Specify whether the blob stream processor should try to use string packing.	[N/A] N/A NCBI_SERIAL_PACK_STRINGS	Boolean [d]	true
On some platforms, equal strings can share their character data, reducing the required memory. Set this parameter to true to have the GenBank loader try to use this feature if it is available.	[GENBANK] SNP_PACK_STRINGS GENBANK_SNP_PACK_STRINGS	Boolean [a]	true
In ID1/PubSeqOS readers present SNP data as ID2-split entries to reduce memory usage.	[GENBANK] SNP_SPLIT GENBANK_SNP_SPLIT	Boolean [a]	true
Storing all the SNPs as plain ASN.1 objects would require a huge amount of memory. The SNP table is a compact way of storing SNPs to reduce memory consumption. Set this parameter to true to have the object manager try to use the SNP table.	[GENBANK] SNP_TABLE GENBANK_SNP_TABLE	Boolean [a]	true
Set to a positive integer to enable dumping (to stderr in text ASN.1 form) all the SNPs that don't fit into the SNP table. Note: this is only available in debug mode.	[GENBANK] SNP_TABLE_DUMP GENBANK_SNP_TABLE_DUMP	Boolean [a]	false
Set this parameter to true to dump (to stdout) some statistics on the process of storing SNPs into the SNP table. This option may help determine why not all the SNPs could fit in the table.	[GENBANK] SNP_TABLE_STAT GENBANK_SNP_TABLE_STAT	Boolean [a]	false

Specify whether to use a memory pool.	[GENBANK] USE_MEMORY_POOL GENBANK_USE_MEMORY_POOL	Boolean [a]	true
The WAIT_TIME* parameters describe the wait time before opening new GenBank connections in case of communication errors. The wait time is necessary to allow network and/or GenBank servers to recover. WAIT_TIME is the initial wait after the first error. See also: GenBank reader configuration.	[GENBANK] WAIT_TIME NCBI_CONFIG__GENBANK__WAIT_TIME [c]	floating point >= 0.0	1 second
Specifies for how many sequential communication errors the response should be to use wait time, before trying to open a new connection instead.	[GENBANK] WAIT_TIME_ERRORS NCBI_CONFIG__GENBANK__WAIT_TIME_ERRORS [c]	int	2 errors
WAIT_TIME_MULTIPLIER and WAIT_TIME_INCREMENT specify the way wait time is increased if errors continue to happen (next_wait_time = prev_wait_time * multiplier + increment).	[GENBANK] WAIT_TIME_INCREMENT NCBI_CONFIG__GENBANK__WAIT_TIME_INCREMENT [c]	any floating point value >= 0.0	1 second
The limit of increasing wait time using WAIT_TIME_MULTIPLIER and WAIT_TIME_INCREMENT.	[GENBANK] WAIT_TIME_MAX NCBI_CONFIG__GENBANK__WAIT_TIME_MAX [c]	floating point >= 0.0	30 seconds
See WAIT_TIME_INCREMENT	[GENBANK] WAIT_TIME_MULTIPLIER NCBI_CONFIG__GENBANK__WAIT_TIME_MULTIPLIER [c]	any floating point value >= 0.0	1.5
Prioritized list of drivers to try for the writer.	Sources searched for list: [GENBANK] WriterName, [GENBANK] LOADER_METHOD, default Sources searched for list: GENBANK_LOADER_METHOD, default	list items are semicolon-delimited; each item is a colon-delimited list of drivers. valid drivers: id1, id2, cache, pubseqos	"ID2:PUBSEQOS:ID1", or "ID2:ID1" (see API)
If non-zero, reserve Dense-seg vectors using predefined pre-read hook.	[OBJECTS] DENSE_SEG_RESERVE OBJECTS_DENSE_SEG_RESERVE	Boolean [a]	true
If non-zero, reserve Seq-graph vectors using predefined pre-read hook.	[OBJECTS] SEQ_GRAPH_RESERVE OBJECTS_SEQ_GRAPH_RESERVE	Boolean [a]	true
If non-zero, reserve Seq-table vectors using predefined pre-read hook.	[OBJECTS] SEQ_TABLE_RESERVE OBJECTS_SEQ_TABLE_RESERVE	Boolean [a]	true
Specify whether Seq-id general trees are packed.	[OBJECTS] PACK_GENERAL OBJECTS_PACK_GENERAL	Boolean [a]	true

Specify whether Seq-id text-seq trees are packed.	[OBJECTS] PACK_TEXTID OBJECTS_PACK_TEXTID	Boolean [a]	true
Specify whether empty Seq-descr's will be allowed (or throw if not).	[OBJECTS] SEQ_DESCR_ALLOW_EMPTY OBJECTS_SEQ_DESCR_ALLOW_EMPTY	Boolean [a]	false
Sets the maximum number of master TSE blobs that will be cached.	[OBJMGR] BLOB_CACHE OBJMGR_BLOB_CACHE	unsigned int	10
Specify whether the scope can be auto-released.	[OBJMGR] SCOPE_AUTORELEASE OBJMGR_SCOPE_AUTORELEASE	Boolean [a]	true
Specify the size of the scope auto-release.	[OBJMGR] SCOPE_AUTORELEASE_SIZE OBJMGR_SCOPE_AUTORELEASE_SIZE	unsigned int	10
Specify whether the new FASTA implementation will be used.	[READ_FASTA] USE_NEW_IMPLEMENTATION NCBI_CONFIG__READ_FASTA__USE_NEW_IMPLEMENTATION [c]	Boolean [a]	true

[a] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[b] case-insensitive: true, t, yes, y, false, f, no, n

[c] environment variable name formed from registry section and entry name

[d] case-insensitive: true values are { yes | 1 }; anything else is false

Table 14. DBAPI configuration parameters

Purpose	[Registry section] Registry name Environment variable	Valid values	Default
If RESET_SYBASE is true, the Sybase client path will be set to the value in the SYBASE variable.	[N/A] N/A RESET_SYBASE	Boolean [a]	(none)
If RESET_SYBASE is true, the Sybase client path will be set to the value in the SYBASE variable.	[N/A] N/A SYBASE	a path containing a Sybase client	(none)
The version of the TDS protocol to use with the CTLIB driver.	[CTLIB] TDS_VERSION CTLIB_TDS_VERSION	an installed TDS version	125 (see AP I)
The version of the TDS protocol to use with the FTDS driver.	[FTDS] TDS_VERSION FTDS_TDS_VERSION	0 (auto-detect), 50 (Sybase or Open Server), 70 (SQL Server)	0
Whether connecting with Kerberos authentication is supported. If true, and the username and password are empty strings, then DBAPI will attempt to use Kerberos to connect to the database. The user must ensure that the database will allow them to connect via Kerberos and that their Kerberos ticket is not expired.	[dbapi] can_use_kerberos NCBI_CONFIG__DBAPI__CAN_USE_KERBEROS [c]	Boolean [b]	false
Whether to encrypt login data.	[dbapi] conn_use_encrypt_data NCBI_CONFIG__DBAPI__CONN_USE_ENCRYPT_DATA [c]	Boolean [b]	false
The maximum number of simultaneously open connections to database servers.	[dbapi] max_connection NCBI_CONFIG__DBAPI__MAX_CONNECTION [c]	unsigned int	100

The maximum number of connection attempts that will be made for any server.	[DB_CONNECTION_FACTORY] MAX_CONN_ATTEMPTS NCBI_CONFIG__DB_CONNECTION_FACTORY__MAX_CONN_ATTEMPTS [c]	unsigned int	1
The maximum number of validation attempts that will be made for each connection.	[DB_CONNECTION_FACTORY] MAX_VALIDATION_ATTEMPTS NCBI_CONFIG__DB_CONNECTION_FACTORY__MAX_VALIDATION_ATTEMPTS [c]	unsigned int	1
The maximum number of servers to try to connect to for each service name (this is only meaningful if the number of servers running this service exceeds this value).	[DB_CONNECTION_FACTORY] MAX_SERVER_ALTERNATIVES NCBI_CONFIG__DB_CONNECTION_FACTORY__MAX_SERVER_ALTERNATIVES [c]	unsigned int	32
The maximum number of connections to be made to one particular server (when several connections to the same service name are requested) before an attempt to connect to another server will be made. A value of 0 means connect to the same server indefinitely.	[DB_CONNECTION_FACTORY] MAX_DISPATCHES NCBI_CONFIG__DB_CONNECTION_FACTORY__MAX_DISPATCHES [c]	unsigned int	0
The timeout, in seconds, to be used for all connection attempts (0 means to use either the default value or a value set specifically for the driver context).	[DB_CONNECTION_FACTORY] CONNECTION_TIMEOUT NCBI_CONFIG__DB_CONNECTION_FACTORY__CONNECTION_TIMEOUT [c]	unsigned int	30
The timeout, in seconds, to be used while logging into the server for all connection attempts (0 means to use either the default value or a value set specifically for the driver context).	[DB_CONNECTION_FACTORY] LOGIN_TIMEOUT NCBI_CONFIG__DB_CONNECTION_FACTORY__LOGIN_TIMEOUT [c]	unsigned int	30

If DBAPI resolved the passed name as a service name and then couldn't connect to any server associated with that service name, then this parameter determines whether DBAPI should also try to resolve the passed name as a server name (a database alias from "interfaces" file or a DNS name). See also: database load balancing.	[DB_CONNECTION_FACTORY] TRY_SERVER_AFTER_SERVICE NCBI_CONFIG__DB_CONNECTION_FACTORY__TRY_SERVER_AFTER_SERVICE [c]	Boolean [a]	false
See 'PRAGMA cache_size' in the SQLite documentation.	[LDS2] SQLiteCacheSize LDS2_SQLITE_CACHE_SIZE	any valid cache size for an SQLite database	2000

[a] case-insensitive: true, t, yes, y, false, f, no, n

[b] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[c] environment variable name formed from registry section and entry name

Table 15. eutils library configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Specify the base URL for Eutils requests.	[Eutils] Base_URL EUTILS_BASE_URL	a valid URL	http://eutils.ncbi.nlm.nih.gov/entrez/eutils/ (see API)

Table 16. Common NetCache and NetSchedule client API configuration parameters (netservice_api)

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid va**
Fail the request if the network I/O is inactive (blocked waiting for the communication channel to become readable or writable) for more than the specified timeout in seconds. Applies to all socket operations after the initial connection is established (see NCBI_CONFIG__NETSERVICE_API__CONNECTION_TIMEOUT). Can be overridden by NCBI_CONFIG__NETCACHE_API__COMMUNICATION_TIMEOUT or NCBI_CONFIG__NETSCHEDULE_API__COMMUNICATION_TIMEOUT.	[netservice_api] communication_timeout NCBI_CONFIG__NETSERVICE_API__COMMUNICATION_TIMEOUT [a]	floating point >= (zero me default)
The maximum number of times the API will retry a communication command on a socket. Setting connection_max_retries to zero will prevent NetCache API from retrying the connection and command execution	[netservice_api] connection_max_retries NCBI_CONFIG__NETSERVICE_API__CONNECTION_MAX_RETRIES [a]	unsignec
The timeout in seconds for establishing a **new** connection to a server. Can be overridden by NCBI_CONFIG__NETCACHE_API__CONNECTION_TIMEOUT or NCBI_CONFIG__NETSCHEDULE_API__CONNECTION_TIMEOUT.	[netservice_api] connection_timeout N/A	floating point > (millisec precision minimum 0.001 (1 millisec
The number of connections to keep in the local connection pool. If zero, the server will grow the connection pool as necessary to accomodate new connections. Otherwise, when all connections in the pool are used, new connections will be created and destroyed.	[netservice_api] max_connection_pool_size NCBI_CONFIG__NETSERVICE_API__MAX_CONNECTION_POOL_SIZE [a]	non- negative
The maximum number of attempts to resolve the LBSMD service name. If not resolved within this limit an exception is thrown.	[netservice_api] max_find_lbname_retries NCBI_CONFIG__NETSERVICE_API__MAX_FIND_LBNAME_RETRIES [a]	positive
The delay in seconds between retrying a command; the total time should not exceed NCBI_CONFIG__NETCACHE_API__MAX_CONNECTION_TIME.	[netservice_api] retry_delay NCBI_CONFIG__NETSERVICE_API__RETRY_DELAY [a]	floating point >=
Close connections with zero timeout to prevent sockets in TIME_WAIT on the client side. By default, the Linux kernel delays releasing ports for a certain period after close() because there might be a delayed arrival of packets. Setting this parameter to true disables that behavior and therefore allows faster recycling of ports. This is important when the server is handling a large number of connections due to the limited number of ports available.	[netservice_api] use_linger2 NCBI_CONFIG__NETSERVICE_API__USE_LINGER2 [a]	Boolean

[a] environment variable name formed from registry section and entry name

[b] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

Table 17. NetCache client API configuration parameters (netcache_api)

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**
Enable input caching (provides for slow blob retrieval).	[netcache_api] cache_input N/A	Boolean [b]
Only applies when using CNetICacheClient. Provides a "namespace" for blobs. Thus, blobs are uniquely identified by the { key, version, subkey, cache_name } combination.	[netcache_api] cache_name N/A	up to 36 characters (cas
Enable output caching (provides for saving a blob with pauses more than "communication_timeout").	[netcache_api] cache_output N/A	Boolean [b]
The name of your application, as identified to NetCache.	[netcache_api] client N/A	your application's name
Synonym for [netcache_api]/client, which is preferred.	[netcache_api] client_name N/A	
Can be used to override NCBI_CONFIG__NETSERVICE_API__COMMUNICATION_TIMEOUT. Please see that entry for details.	[netcache_api] communication_timeout N/A	floating point >= 0.0 (z NCBI_CONFIG__NET
Can be used to override [netservice_api]/connection_timeout. Please see that entry for details.	[netcache_api] connection_timeout N/A	floating point >= 0.0, m [netservice_api]/connec
Depending on the value, enables mirroring: if true, mirroring is unconditionally enabled, if false, it is disabled completely. The special value "if_key_mirrored" is used to enable mirroring for the blobs that already have mirroring extensions in their keys.	[netcache_api] enable_mirroring N/A	Boolean , or "if_key_m
The host:port address for the NetCache server that will be used for blob creation if none of the servers configured via LBSM were able to create the blob. This is only for new blob requests.	[netcache_api] fallback_server NCBI_CONFIG__NETCACHE_API__FALLBACK_SERVER [a]	a valid server
In conjunction with [netcache_api]/port, a synonym for [netcache_api]/service_name, which is preferred.	[netcache_api] host N/A	
Max total time for each NetCache transaction.	[netcache_api] max_connection_time N/A	floating point >= 0.0 (z
In conjunction with [netcache_api]/host, a synonym for [netcache_api]/service_name, which is preferred.	[netcache_api] port N/A	

A trigger for LBSM query (query LBSM once per the specified number of NetCache operations).	[netcache_api] rebalance_requests N/A	integer >= 0 (zero mear
Another trigger for LBSM query (query LBSM at least once per the specified number of seconds)	[netcache_api] rebalance_time N/A	floating point >= 0.0 (z
Synonym for [netcache_api]/host, which is preferred.	[netcache_api] server N/A	
Synonym for [netcache_api]/service_name.	[netcache_api] service N/A	
The LBSM name that specifies which servers to use. The service name is only used when creating blobs.	[netcache_api] service_name N/A	any registered LBSM se
This is one condition that will trigger server throttling and is defined as a string having the form "A / B" where A and B are integers. Throttling will be triggered if there are A failures in the last B operations.	[netcache_api] throttle_by_connection_error_rate N/A	a string having the form
This is another condition that will trigger server throttling and is defined as follows. Server throttling will be triggered if this number of consecutive connection failures happens.	[netcache_api] throttle_by_consecutive_connection_failures N/A	integer
Do not release server throttling until the server appears in LBSMD.	[netcache_api] throttle_hold_until_active_in_lb N/A	Boolean [c]
Indicates when server throttling will be released.	[netcache_api] throttle_relaxation_period N/A	integer time period in se
Where to save blob caches.	[netcache_api] tmp_dir N/A	a valid directory
Synonym for [netcache_api]/tmp_dir.	[netcache_api] tmp_path N/A	
A true value enables an alternative method for checking if a blob exists. Note: This option is available only for backward compatibility and should not be used.	[netcache_api] use_hasb_fallback NCBI_CONFIG__NETCACHE_API__USE_HASB_FALLBACK [a]	Boolean [b]
Defines LBSM affinity name to use for floor assignment, etc.	[netcache_api] use_lbsm_affinity N/A	a valid affinity

[a] environment variable name formed from registry section and entry name

[b] case-insensitive: true, t, yes, y, 1, false, f, no, n, 0

[c] case-insensitive: true, t, yes, y, false, f, no, n

Table 18. NetSchedule client API configuration parameters (netschedule_api)

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**
Name of the queue (DO NOT use default queue for your application).	[netschedule_api] queue_name N/A	your application's queue name
The name of your application, as identified to NetSchedule.	[netschedule_api] client_name N/A	your application's name
Can be used to override NCBI_CONFIG__NETSERVICE_API__COMMUNICATION_TIMEOUT. Please see that entry for details.	[netschedule_api] communication_timeout N/A	floating point >= 0.0 (zero means use the default from NCBI_CONFIG__NETSERVICE_API__COMMUNICATION_TI
Can be used to override [netservice_api]/connection_timeout. Please see that entry for details.	[netschedule_api] connection_timeout N/A	floating point >= 0.0 (zero means use the default from [netservice_a connection_timeout)

Table 19. seqfetch.cgi application configuration parameters

Purpose	**[Registry section]** **Registry name** **Environment variable**	**Valid values**	**Default**
Point to the current script.	[SeqFetch] Viewer_fcgi_path SEQFETCH_VIEWER_FCGI_PATH	a valid path	/sviewer/viewer.fcgi
Name the current load-balanced proxy.	[SeqFetch] Viewer_fcgi_proxy SEQFETCH_VIEWER_FCGI_PROXY	a valid proxy name	sviewer_lb

The NCBI C++ Toolkit

Release Notes

This appendix is a compilation of all of the release notes in reverse chronological order. So the latest release notes are listed first.

These notes give a somewhat superficial and incomplete (albeit still useful) description of the latest NCBI C++ Toolkit changes, fixes and additions. Some important topics (especially numerous bug fixes and feature improvements, but possibly a bigger fish) are just out of scope of these notes. Feel free to write to the mailing group http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp with any questions or reports.

The NCBI C++ Toolkit

Release Notes (Version 12, May 2013)

Created: June 18, 2013.
Last Update: June 20, 2013.

Download

Download the source code archives at: ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/ARCHIVE/12_0_0/

- ncbi_cxx--12_0_0.tar.gz — for UNIX'es (see the list of UNIX flavors below) and MacOSX
- ncbi_cxx--12_0_0.exe — for MS-Windows (32- and 64-bit) / MSVC++ 10.0 — self-extracting
- ncbi_cxx--12_0_0.zip — for MS-Windows (32- and 64-bit) / MSVC++ 10.0

The sources correspond to the NCBI production tree sources, which are originally based on the development tree source snapshot from March 11, 2013 but also include many hundreds of important and safe code updates made since then and through May 17, 2013 (and then some).

Third Party Packages

Some parts of the C++ Toolkit just cannot be built without 3rd party libraries, and other parts of the Toolkit will work more efficiently or provide more functionality if some 3rd-party packages (such as BerkeleyDB which is used for local data cache and for local data storage) are available.

For more information, see the FTP README.

Table 1. Currently Supported/Tested Versions of Third Party Packages

Package	Versions expected to work (obtained by build-environment inspection in some cases)	Versions known to work (used in-house on any platform)
BerkeleyDB	4.3.0 or newer	4.5.20, 4.6.21.1, 4.7.25, 4.6.21.NC
Boost Test	1.35.0 or newer	1.40.0.1, 1.42.0, 1.45.0
FastCGI	All versions	2.1, 2.4.0
libbzip2	All versions	1.0.2, 1.0.5
libjpeg	All versions	6b, 8.0
libpng	All versions	1.2.26, 1.2.7, 1.5.13
libtiff	All versions	3.6.1, 3.9.2, 4.0.0
libungif	All versions	4.1.3 (libungif), 4.1.6 (giflib)
libxml2	All versions	2 2.7.3, 2.7.6, 2.7.8,
libxslt	1.1.14	1.1.24, 1.1.26
LZO	2.x	2.05
PCRE	All versions	7.8, 7.9, 8.32,
SQLite3	3.6.6 or newer	3.6.12, 3.6.14.2, 3.6.22, 3.7.13
Sybase	All versions	12.5
zlib	All versions	1.2.3, 1.2.3.3

For Mac OS X and UNIX OS's, the user is expected to download and build the 3rd party packages themselves. The release's package list includes links to download sites. However, the user still needs a list of the 3rd party packages and which versions of them are compatible with the release.

To facilitate the building of these 3rd-party libraries on Windows, there is an archive that bundles together source code of the 3rd-party packages, plus MSVC "solutions" to build all (or any combination) of them.

Table 2. Versions of Third Party Packages Included in the FTP Archive

Package	**Depends On**	**Included Version** [a]
BerkeleyDB		4.6.21.NC
Boost Test		1.42.0
libbzip2		1.0.2
libjpeg		6b
libpng	zlib 1.2.3	1.2.7
libtiff	libjpeg 6b, zlib 1.2.3	3.6.1
libungif		4.1.3
LZO		2.05
PCRE		7.9
SQLite3		3.6.14.2
zlib		1.2.3

[a] Applies to MSVC 9, MSVC 10

Build

For guidelines to configure, build and install the Toolkit see here.

New Developments

HIGHLIGHTS

Major advances, additions to the BAM, SRA, cSRA, WGS, VDB data loaders of the Bio-Sequence Object Manager

FreeTDS driver -- Support Kerberos authentication.

Redesigned Unicode support (stage 1) - added new CUtf8 class which will handle UTF8 conversions and replace CStringUTF8, prohibited implicit single byte character string conversions.
Significant additions and improvements in the XML and JSON serialization APIs.

Cleaned up the code (again) from non-MT-safe static objects.

CORELIB

New functionality:

- Added possibility of having several argument description (CArgDescription) objects in a program; proper description is chosen based on the value of the very first command line argument - "command", the rest of the arguments is then parsed according to the chosen description. Such command descriptions can be combined into command groups.

- Added platform-independent error reporting mechanism, similar to errno or SetLastError, - CNcbiError. When a Toolkit core API function fails, it reports aditional information there.
- Redesigned Unicode support - added new CUtf8 class which will handle UTF8 conversions and replace CStringUTF8, prohibited implicit single byte character string conversions.
- Added CException manipulators for severity and console output.
- NStr:: -- improved errno handling, dropped support for fIgnoreErrno flag.
- NStr:: -- addednew methods CommonPrefixSize(), CommonSuffixSize(), CommonOverlapSize().
- NStr::StringToNumeric() -- renamed to StringToNonNegativeInt().
- Nstr::ParseEscapes() -- added options to parse out-of-range escape sequences.
- NStr::CEncode() -- rewrite to produce use double-quoted strings by default, and added counterpart method CParse() to decode a "C" strings.
- CTime -- added GetCurrentTimeT() to get current GMT time with nanoseconds.
- CSignal -- added method ClearSignals().
- CDirEntry -- add permission/mode <-> string conversion methods.
- CDirEntry -- added methods: GetUmask(), SetUmask, ModeFromModeT().
- SetCpuTimeLimit() -- added new declaration and deprecated old one, re-enabled user print handler for SIGXCPU.
- SetMemoryLimit[Soft|Hard]() -- new methods to allow separately specify soft and hard memory limits for application.
- Added string literals as well as directory pathes to CExprParser
- CNCBIRegistry (and other registries) is able to work with configuration data not belonging to any section, when created with fSectionlessEntries
- CExprParser is able to accept logical literals starting with a number in fLogicalOnly mode

Improvements:

- In-heap CObject detection via TLS variable.
- Inline internal method CObject::InitCounter() for speed.
- CTempStringEx -- Optionally own data.
- NStr -- Split, Tokenize, etc. now accept flags, controlling not only delimiter merging but also whether to treat multi-character delimiters as patterns (generalizing TokenizePattern) and to treat any subset of \'" as special.

DATA SERIALIZATION

New functionality:

- Added support for mandatory elements with default in XML serialization.
- Added possibility of using NCBI_PARAM mechanism for data verification and
- skipping unknown members settings.

- Added possibility of skipping unknown data in JSON input; added JSONP output mode.

Improvements:

- Optimization of deserialization methods (mostly binary ASN.1).

Bug fixes:

- Avoid double closing tag when skipping enums in XML.
- Store serialization format flags for correct delayed parsing.

XMLWrapp:

- Safe dereferencing node iterators
- xml::nodes_view is not supported anymore
- A few memory leaks are fixed
- exslt auto registration if available
- XSLT extension functions support added
- XSLT extension elements support added
- run_xpath_expression(…) to handle boolean, number and string types as return values

DATATOOL

- Enhanced SOAP client code generation to support WSDL specification which contains several XML schemas - to handle elements with identical names in different namespaces.
- Added possibility of converting data to and from JSON format.

CGI

- CCgiUserAgent -- separate CCgiUserAgent into user_agent.[h|c]pp.
- CCgiUserAgent -- added methods: GetDeviceType(), IsPhoneDevice(), IsTabletDevice().
- Added flags to allow use external pattern lists on the parsing step to identify bots, phones, tablets and mobile devices. Return iPad back to the list of mobile devices. Interpret all Android based devices as mobile devices.
- CCgiUserAgent -- update list of browsers and mobile devices.

UTILITES

- Compression API -- allow concatenated files for eGZipFile mode by default.
- Compression API -- added support for "empty input data" compression via fAllowEmptyData flag, It will allow to compress zero-length input data and provide proper format header/footer in the output, if applicable. By default the Compression API not provide any output for zero-length input data.
- CRegexp -- changed GetSub()/GetMatch() methods to return CTempString instead of string.
- include/util/diff/dipp.hpp -- new DIFF API (CDiff, CDiffText).
- Added possibility to convert differently typed static array.
- Added limited_size_map<> for caching and garbage collection.

- Mask matching is rewritten with CTempString for efficiency.
- ILineReader -- Clarify API, introducing ReadLine and GetCurrentLine as synonyms of operator++ and operator* respectively.
- CTextJoiner -- New template for collecting and joining strings with a minimum of heap churn.

DBAPI

- Support Sybase ASE 15.5 servers.
- Python bindings -- Optionally release Python's global lock around blocking DBAPI operations; rework exception translation to follow PEP 249 (*).
- Added support for Kerberos authentication (copied from FreeTDS 0.91).

BIO-OBJECTS

New functionality:

- CDeflineGenerator -- Generally streamline; make expensive operations (currently just consulting related sequences' annotations) optional.
- CFastaOStream -- Add a gap-mode parameter, making it possible to represent gaps by runs of inline dashes or special >?N lines; optionally (but by default) check for duplicate sequence IDs; support processing an entire raw Seq-entry without even a temporary scope.
- CFastaReader -- Add two flags that can increase performance: fLeaveAsText skips reencoding in a (more compact) binary format, and fQuickIDCheck directs local ID validation to consider just the first character.
- CSeq_id -- Accept parse flags when parsing a single ID from a string; recognize WGS scaffolds, additional prefixes (F???, G???, HY, HZ, J??, JV-JZ, KA-KF, and WP_), 10-digit refseq_wgs_nuc accessions (notably for spruce), and more TPE protein accessions (still interspersed with EMBL's own accessions).
- Added CGeneFinder class for finding the genes of a feature using the flatfile generator's logic
- Seq_entry_CI can now optionally include the top seq-entry
- objects::CGC_Replicon now has accessors to return molecule type (‘Chromosome’, ‘Plasmid’, etc.) and location (‘Nuclear’, ‘Mitochondrion’, ‘Chloroplast’, etc.). You can also retrieve a label (GetMoleculeLabel()) which summarizes molecule type and location in one string.

BIO-TOOLS

New Development:

- Validator:
  - Added functions for validating and autocorrecting lat-lon, collection-date, and country BioSource SubSource modifiers. Synchronized validation with C Toolkit.
- Flat-file generator:
  - can now be set to show only certain blocks
  - optionally set callback for each item or bisoeq that's written which allows changing the text and specifying to skip that item or even halt flatfile generation.

  - support the /pseudogene qualifier
  - allow complex locations in transl_excepts. (a.k.a. code-breaks)
  - support "pcr" linkage-evidence
  - support for /altitude qualifier
  - Support "Assembly" in DBLINK
  - API for conversion between source-qualifier and feature-qualifier enums and strings
  - support assembly gap feature quals (e.g. /gap_type, /linkage_evidence, etc.)
- ASN.1 Cleanup:
  - Set pseudo to true if pseudogene is set
  - More places where it sorts and removes redundancies (example: sort and unique organism synonyms)
  - Remove duplicate pcr-primers
  - clean up altitude
  - fixing some genbank quals into real quals (example: gene-synonym)
- CFastaOstream:
  - Can optionally show [key=val] style mods in deflines
- CFeature_table_reader:
  - now supports more quals (example: centromere)
- CFastaReader:
  - optionally accumulate warnings in a vector instead of printing them to allow more flexible handling and more info to caller.
- AGP:
  - created CAgpToSeqEntry for converting an AGP file into a Seq-entry.

COBALT

Bug fixes:

- Incorrect alignments with sequence clustering

BIO-OBJECT MANAGER

New functionality:

- Added fast CScope methods for getting some sequence information without loading the whole entry - length, type, taxonomy id, GI, accession, label.
- Added processing of Seq-table column "disabled".
- Added FeatId manipulation methods.
- Added feature::ReassignFeatureIds().
- Added CSeq_table_CI with location mapping.
- Added CSeqVector_CI::GetGapSeq_literal().
- Added recursive mode and seq-entry type filtering to CSeq_entry_CI.

Improvements:

- Allow non-scope bioseq lookup in CSeq_Map (for segset entries).

- Allow post-load modification of sequences.
- Optimization of ContainsBioseq() for split entries.
- Added CTSE_Info::GetDescription() for better diagnostics.
- More detailed error message in annots.
- Allow iteration over non-set entries in CSeq_entry_CI - treat them as empty sets.

Bug fixes:

- Fixed generation of Seq-table features.
- Fixed loading of various Seq-id info from multiple data loaders.
- Made bulk and single requests to return the same results.
- Fixed unexpected CBlobStateException for non-existent sequences.
- Avoid deadlock when updating split annot index.
- Fixed recursive iteration in CSeq_entry_CI if sub-entry
- doesn't have matching entries.
- Fixed mixup of feature ids and xrefs.
- Fixed fetching by feat id/xref from split entries.
- Fixed in-TSE sequence lookup via matching Seq-id.
- Fixed matching Seq-id lookup with multiple candidates.
- CSeqMap_CI::GetRefData() should work for gaps too.
- Exclude removed features from un-indexed search.

OBJECT LIBRARIES

New functionality:

- Implemeted multi-id Seq-loc comparison.

GENBANK DATA LOADER

Bug fixes:

- Allow withdrawn/suppressed entries with non-default credentials.
- Preserve blob state if Seq-entry skeleton is attached to split info.
- Remember blob state from get-blob-ids reply too.
- Detect non-existent Seq-id when loading blob-ids.
- Release connection as soon as possible to avoid deadlock.
- Lock split TSE only after receiving split info.

BAM DATA LOADER

New functionality:

- Implemented pileup graphs for BAM loader.

Improvements:

- Generate simple ID2 split info to postpone record loading.

SRA DATA LOADER

New functionality:

- Added option to clip SRA sequences.

cSRA DATA LOADER

New functionality:

- Implemented CCSraShortReadIterator.
- Added short read info into Seq-align.ext.
- Added pileup graph param setter and getter.
- Added support for SECONDARY_ALIGNMENT.
- Use gnl|SRA|<acc>.<spot>.<read> for short read ids.
- Added lookup for short reads by SPOT_ID and READ_ID.
- Allow optional VDB columns.
- Added clippig by quality.
- Added option to exclude cSRA file path from short read ids.

Improvements:

- Allow cSRA reader to open old SRA tables.
- Reduced number of TSE chunks.
- Removed obsolete config parameters: INT_LOCAL_IDS, SEPARATE_LOCAL_IDS.
- Removed empty VDB table, cursor, and column constructors.
- Generate simple split info to postpone cSRA record loading.
- Exclude technical reads.
- Check VDB column data type to detect incompatible VDB files.
- Place short reads in a separate blob.
- Added lookup from short read to refseq.
- Added mapping align on short read.
- Added secondary alignment indicator.
- Added centralized MT-safe VDB cursor cache.
- Allow ERR accessions in cSRA loader.
- Switched to new SRA SDK accession resolution scheme.
- Use SRA SDK configuration mechanism.
- Added SRA file cache garbage collector.
- Accept multiple ids in reference sequences.
- Reduce number of reads per blob to 1 for speed.
- Allow cSRA data to have no REFERENCE table.
- Increased limit on allowed number of short reads per spot.
- Increased flexibility on existing VDB columns.
- Try to resolve remote VDB files too.
- Use GC for loaded entries.

- Indicate that cSRA loader can load data by blob id.
- Set max value of quality graph properly.

Bug fixes:

- Fixed MISMATCH generation for I segments.
- Added missing RegisterInObjectManager().

WGS DATA LOADER

New functionality:

- Implemented VDB WGS reader and data loader.

VDB DATA LOADER

New functionality:

- Implemented VDB graph reader and data loader.

BLAST

New functionality:

- Added new API to return blast preliminary stage result as a list of CStd_seg
- Added new tabular features for blast which includes taxonomy information, strand sign and query coverage
- Added new features for blastdbcmd batch sequence retrieval which allow user to specify strand sign and sequence range
- Added new functionality in makeprofiledb to produce database that supports composition based statistics
- For more details, see BLAST+ 2.2.27 and 2.2.28 release notes (http://www.ncbi.nlm.nih.gov/books/NBK131777/)

Bug fix

- Fix ASN 1 input for makeblastdb

APPLICATIONS

- convert_seq -- Allow for more efficient operation in some cases, mostly by bypassing object manager overhead; implement a new "IDs" input format; have non-zero inflags for ASN.1 or XML request sequence data repacking.
- multireader -- Added AGP.
- blastn's -- Changed default value - use_index to false
- vecscreen -- Added command line application
- rmblastn -- Added command line application
- asn2asn -- added ability to read and write Seq-submits

BUILD FRAMEWORK (UNIX)

- configure and frontends (compilers/unix/*.sh) -- Don't override explicitly specified optimization flags with default FAST settings (but do still apply custom FAST settings if also specified).

- compilers/unix/Clang.sh, .../LLVM-GCC.sh -- New frontends for configure to simplify compiler selection.
- new_project.sh -- Improve support for projects involving libraries.

CHANGES TO COMPILER SUPPORT

Linux ICC support extends up to version 13.

Mac OS X support extends to version 10.8.x, with Clang, FSF GCC, or LLVM GCC (also via Xcode).

Solaris support extends to version 11, with GCC or WorkShop (as with older OS versions).

Documentation

Location

The documentation is available online as a searchable book "The NCBI C++ Toolkit": http://www.ncbi.nlm.nih.gov/toolkit/doc/book/.

The C++ Toolkit book also provides PDF version of the chapters. The PDF version can be accessed by a link that appears on each page.

Content

Documentation has been grouped into chapters and sections that provide a more logical coherence and flow. New sections and paragraphs continue to be added to update and clarify the older documentation or provide new documentation. The chapter titled "Introduction to the C++ Toolkit" gives an overview of the C++ Toolkit. This chapter contains links to other chapters containing more details on a specific topic and is a good starting point for the newcomer.

A C/C++ Symbol Search query appears on each page of the online Toolkit documentation. You can use this to perform a symbol search on the up-to-date public or in-house versions using source browsers LXR, Doxygen and Library - or do an overall search.

Public assess to our SVN trunk:

- For browsing: http://www.ncbi.nlm.nih.gov/viewvc/v1/trunk/c++
- For retrieval: http://anonsvn.ncbi.nlm.nih.gov/repos/v1/trunk/c++ (NOTE: Some WebDAV clients may require dav:// instead of http://)

Supported Platforms (OS's and Compilers)

- UNIX
- MS Windows
- Mac OS X
- Added
- Discontinued

This release was successfully tested on at least the following platforms (but may also work on other platforms). Since the previous release, some platforms were dropped from this list and some were added. Also, it can happen that some projects would not work (or even compile) in the absence of 3rd-party packages, or with older or newer versions of such packages. In these cases, just skipping such projects (e.g. using flag "-k" for make on UNIX), can get you through.

In cases where multiple versions of a compiler are supported, the mainstream version is shown in **bold**.

UNIX

Table 3. UNIX OS's and Supported Compilers

Operating System	**Architecture**	**Compilers**
CentOS 5.x (LIBC 2.5)	x86-64	**GCC 4.4.2,** 4.0.1[a], 4.1.2[a], 4.3.3[a], 4.6.0[a], 4.6.3[a] ·GCC 4.7.2 [a]
CentOS 5.x (LIBC 2.5)	x86-32	**GCC** 4.4.5 [a]**, 4.6.0**
CentOS 6.x (LIBC 2.12)	x86-64	**GCC 4.4.2**, 4.6.3 [a], 4.7.2 [a], 4.8.0 [a]
Ubuntu 9.04 ("jaunty") (LIBC 2.9)	x86-32 x86-64	**GCC 4.3.3**
Solaris 10, 11[a]	SPARC	GCC 4.1.1[b], 4.5.3[b] **Sun Studio 12 (C++ 5.9)**, Sun Studio 12 Update 1 (C++ 5.10)[a] Oracle Studio 12.2 (C++ 5.11)[a]
Solaris 10, 11[a]	x86-32	GCC 4.2.3 **Sun Studio 12 (C++ 5.9)**, Sun Studio 12 Update 1 (C++ 5.10)[a] Oracle Studio 12.2 (C++ 5.11)[a]
Solaris 10, 11[a]	x86-64	**Sun Studio 12 (C++ 5.9)**, Sun Studio 12 Update 1 (C++ 5.10)[a] Oracle Studio 12.2 (C++ 5.11)[a]
FreeBSD-8.3	x86-32	GCC 4.2.2

[a] some support

[b] 32-bit only

MS Windows

Table 4. MS Windows and Supported Compilers

Operating System	**Architecture**	**Compilers**
MS Windows	x86-32	MS Visual C++ 2010 (C++ 10.0) NOTE: We also ship an easily buildable archive of 3rd-party packages for this platform.
MS Windows	x86-64	MS Visual C++ 2010 (C++ 10.0) NOTE: We also ship an easily buildable archive of 3rd-party packages for this platform
Cygwin 1.7.9	x86-32	GCC 4.5.3- nominal support only.

Mac OS X

Table 5. Mac OS and Supported Compilers

Operating System	**Architecture**	**Compilers**
Mac OS X 10.6 Mac OS X 10.8	Native (PowerPC or x86-32 or x86-64)	Xcode 3.0 - 3.2.6
Darwin 10.x	Native (PowerPC or x86-32 or x86-64), Universal (PowerPC and x86-32)	GCC 4.0.1 GCC 4.2.1 (only available under Darwin 10.x) LLVM Clang 3.0

NOTE: the correspondence between Darwin kernel versions and Mac OS versions:

Darwin 10.x = Mac OS 10.6.x

Darwin 12.x = Mac OS 10.8.x

Added Platforms

Table 6. Added Platforms

Operating System	**Architecture**	**Compilers**
CentOS 5.x (LIBC 2.5)	x86-32	GCC 4.4.5 [a], 4.6.0
CentOS 5.x	x86-64	GCC 4.7.2 [a]
CentOS 6.x (LIBC 2.12)	x86-64	GCC 4.4.2 , 4.6.3 [a], 4.7.2 [a], 4.8.0 [a]
Mac OS X 10.5, MacOS x 10.6,	Native (PowerPC or x86-32 or x86-64)	Xcode 3.2.3 - 3.2.6 LLVM Clang 3.0

[a] some support

Discontinued Platforms

Table 7. Discontinued Platforms

Operating System	**Architecture**	**Compilers**
MS Windows	x86-32, 64	MS Visual C++ 2008 (C++ 9.0)
Mac OS X 10.4.x(Darwin 8.x), Mac OS X 10.5.x(Darwin 9.x)	Native (PowerPC or x86-32 or x86-64), Universal (PowerPC and x86-32)	GCC 4.0.1, Clang 3.0
FreeBSD-6.1	x86-32	GCC 3.4.6
All	All	All GCC 4.0.1 and below

Last Updated

This section last updated on July 1, 2013.

Appendix - Books and Styles

Books and links to C++ and STL manuals

Books

- *On To C++, by Patrick Henry Winston*. If you are looking for a short and concise tutorial, this is as close as you can get. It doesn't cover all of C++, but many of the essential features (except the STL). A decent first book to buy.
- *The C++ Primer, Third Edition*, by Stanley Lippman and Josee Lajoie. A decent book, much expanded from previous editions. Gets carried away with very long examples, which makes it harder to use as a reference. Full coverage of ANSI/ISO C++.
- *The C++ Programming Language, Third Edition* by Bjarne Stroustrup. Often called the best book for C++ written in Danish. Written by the designer of C++, this is a difficult read unless you already know C++. Full coverage of ANSI/ISO C ++.
- *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers. . A must-have that describes lots of tips, tricks, and pitfalls of C++ programming.
- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, by Scott Meyers.. Same as above. For example, how is the new operator different from operator new? Operator new is called by the new operator to allocate memory for the object being created. This is how you hook your own malloc into C++.