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:

- Writing a Simple Application
  - NCBI C++ Toolkit Application Framework Classes
    - CNcbiApplication
    - CNcbiArguments
    - CNcbiEnvironment
    - CNcbiRegistry
    - CNcbiDiag
  - Creating a Simple Application
    - Unix-like Systems
    - MS Windows
    - Discussion of the Sample Application
  - Inside the NCBI Application Class
- Processing Command-Line Arguments
  - Capabilities of the Command-Line API
  - The Relationships between the CArgDescriptions, CArgs, and CArgValue Classes

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

- Command-Line Syntax
- The CArgDescriptions (*) Class
  - 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 CArgs (*) Class: A Container Class for CArgValue (*) Objects
- CArgValue (*) Class: The Internal Representation of Argument Values
- Supporting Command-Based Command Lines
- Code Examples

- Namespace, Name Concatenation, and Compiler-specific Macros
  - NCBI Namespace
  - Other Namespace Macros
  - Name Concatenation
  - Compiler-specific Macros
- Configuration Parameters
  - General Usage Information
  - Macros for Creating Parameters
  - Methods for Using Parameters
  - Supporting Classes
- Using the CNcbiRegistry Class
  - 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
- Portable Stream Wrappers
- Working with Diagnostic Streams (*)
  - Where Diagnostic Messages Go
  - Setting Diagnostic Severity Levels
  - Diagnostic Messages Filtering
  - Log File Format
    - The Old Post Format
    - The New Post Format
    - Controlling Appearance of Diagnostic Message Using Post Flags

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

- Defining the Output Stream
- Tee Output to STDERR
- The Message Buffer
- Request Exit Status Codes
  - Standard (HTTP-like) status codes
  - NCBI-specific status codes
- Error Codes and Their Descriptions
  - Preparing an Error Message File
- Defining Custom Handlers Using CDiagHandler
- The ERR_POST and LOG_POST Macros
- The _TRACE macro
- Stack Traces
  - Printing a Stack Trace
  - Obtaining a Stack Trace for Exceptions

- Debug Macros
- Handling Exceptions
- Defining the Standard NCBI C++ Types and Their Limits
  - Headers Files Containing Portability Definitions
  - Built-in Integral Types
  - Auxiliary Types
  - Fixed-Size Integer Types
  - The "Ncbi_BigScalar" Type
  - Encouraged and Discouraged Types
- Understanding Smart Pointers: the CObject and CRef Classes
  - STL auto_ptrs
  - The CRef (*) Class
  - The CObject (*) Class
  - The CObjectFor (*) Class: Using Smart Pointers for Standard Types
  - When to Use CRefs and auto_ptrs
  - CRef Pitfalls
    - Inadvertent Object Destruction
- Atomic Counters
- Portable Mechanisms for Loading DLLs
  - CDll Constructor
  - CDll Basename
  - Other CDll Methods
- Executing Commands and Spawing Processes Using the CExec Class
  - Executing a System Command Using the System() Method
  - Defining Spawned Process Modes (EMode Type)

The NCBI C+ Toolkit Book
The NCBI C+ Toolkit Book

- Spawning a Process Using SpawnX() Methods
- Waiting for a Process to Terminate Using the Wait() Method

• Implementing Parallelism Using Threads and Synchronization Mechanisms
  - Using Threads
  - CThread (*) Class Public Methods
  - CThread (*) Class Protected Methods
  - Thread Life Cycle
  - Referencing Thread Objects
  - Thread Local Storage (CTls<> class [*])
  - Mutexes
    - ♦ CMutex
    - ♦ CFastMutex
    - ♦ SSystemMutex and SSystemFastMutex
    - ♦ CMutexGuard and CFastMutexGuard
    - ♦ Lock Classes
      - CRWLock
      - CAutoRW
      - CReadLockGuard
      - CWriteLockGuard
      - CInternalRWLock
      - CSemaphore

• Working with File and Directories Using CFile and CDir
  - CDirEntry Class
  - CFile Class
  - CDir Class
  - CMemoryFile Class

• String APIs
  - String Constants
  - NStr Class
  - UTF-8 Strings
  - PCase and PNocase

• Portable Time Class
  - CTime Class Constructors
  - Other CTime Methods

• Template Utilities
  - Function Objects
  - Template Functions

• Miscellaneous Types and Macros
  - Miscellaneous Enumeration Types

- AutoPtr Class
- ITERATE Macros
- Sequence Position Types
- Containers
  - template<typename Coordinate> class CRange
    - Typedefs
    - Methods
  - template<typename Object, typename Coordinate = int> class CRangeMap
  - template<typename Object, typename Coordinate = int> class CRangeMultiMap
  - class CIntervalTree
- Thread Pools
  - class CThreadPool
  - class CThreadPool_Task
  - class CThreadPool_Thread
  - class CThreadPool_Controller
  - class CThreadPool_Controller_PID
- Miscellaneous Classes
  - class CTempString
  - class CChecksum
- Input/Output Utility Classes
  - class CIStreamBuffer
  - class COStreamBuffer
  - class CByteSource
  - class CStreamByteSource
  - class CFStreamByteSource
  - class CFileByteSource
  - class CMemoryByteSource
  - class CByteSourceReader
  - class CSubSourceCollector
- Using the C++ Toolkit from a Third Party Application Framework

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 Relationships between the CArgDescriptions, CArgs, and CArgValue Classes
- Command-Line Syntax
- The CArgDescriptions Class
- The CArgs Class: A Container Class for CArgValue Objects
- CArgValue Class: The Internal Representation of Argument Values
- Supporting Command-Based Command Lines
- Code Examples

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.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

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)

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

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.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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:

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
[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,
```

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book


```
 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
- Setting Diagnostic Severity Levels
- Diagnostic Messages Filtering
- Log File Format
  - The Old Post Format
  - The New Post Format
  - Controlling the Appearance of Diagnostic Messages using Post Flags
- Defining the Output Stream
- Tee Output to STDERR
- The Message Buffer
- Request Exit Status Codes
  - Standard (HTTP-like) status codes
  - NCBI-specific status codes
- Error codes and their Descriptions
- Defining Custom Handlers using CDiagHandler
- The ERR_POST and LOG_POST Macros
- The _TRACE macro
- Stack Traces
  - Printing a Stack Trace
  - Obtaining a Stack Trace for Exceptions

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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:

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book


```
<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.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

- 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.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

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
- The CRef Class
- The CObject Class
- The CObjectFor class: using smart pointers for standard types
- When to use CRefs and auto_ptrs
- CRef Pitfalls

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);
```


The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

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 public methods
- CThread class protected methods
- Thread Life Cycle
- Referencing thread objects
- Synchronization
- Thread local storage (CTls<> class [*])

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.

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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.

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

- 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,

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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)
 {
```

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book


```
 // 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.

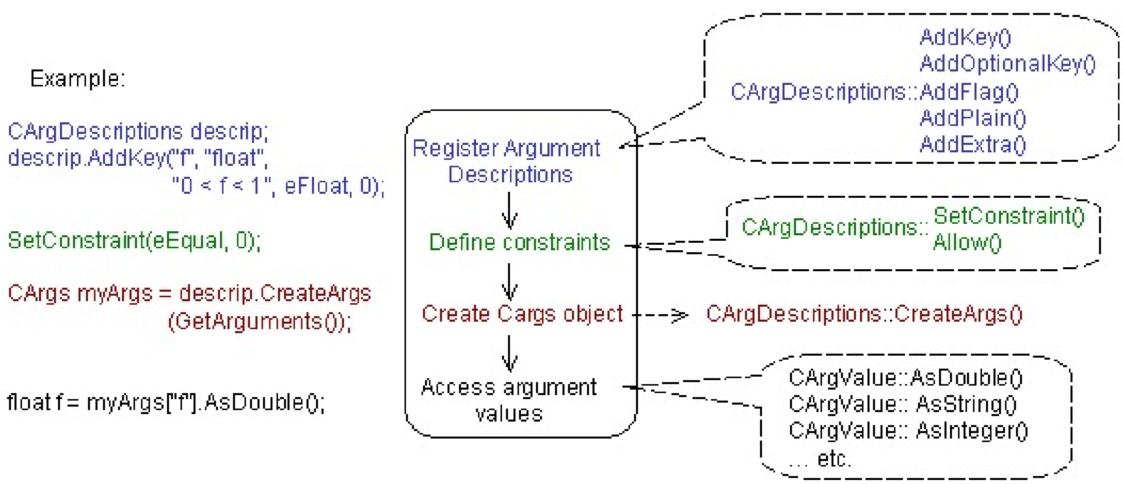

Figure 1. Argument processing class relations.

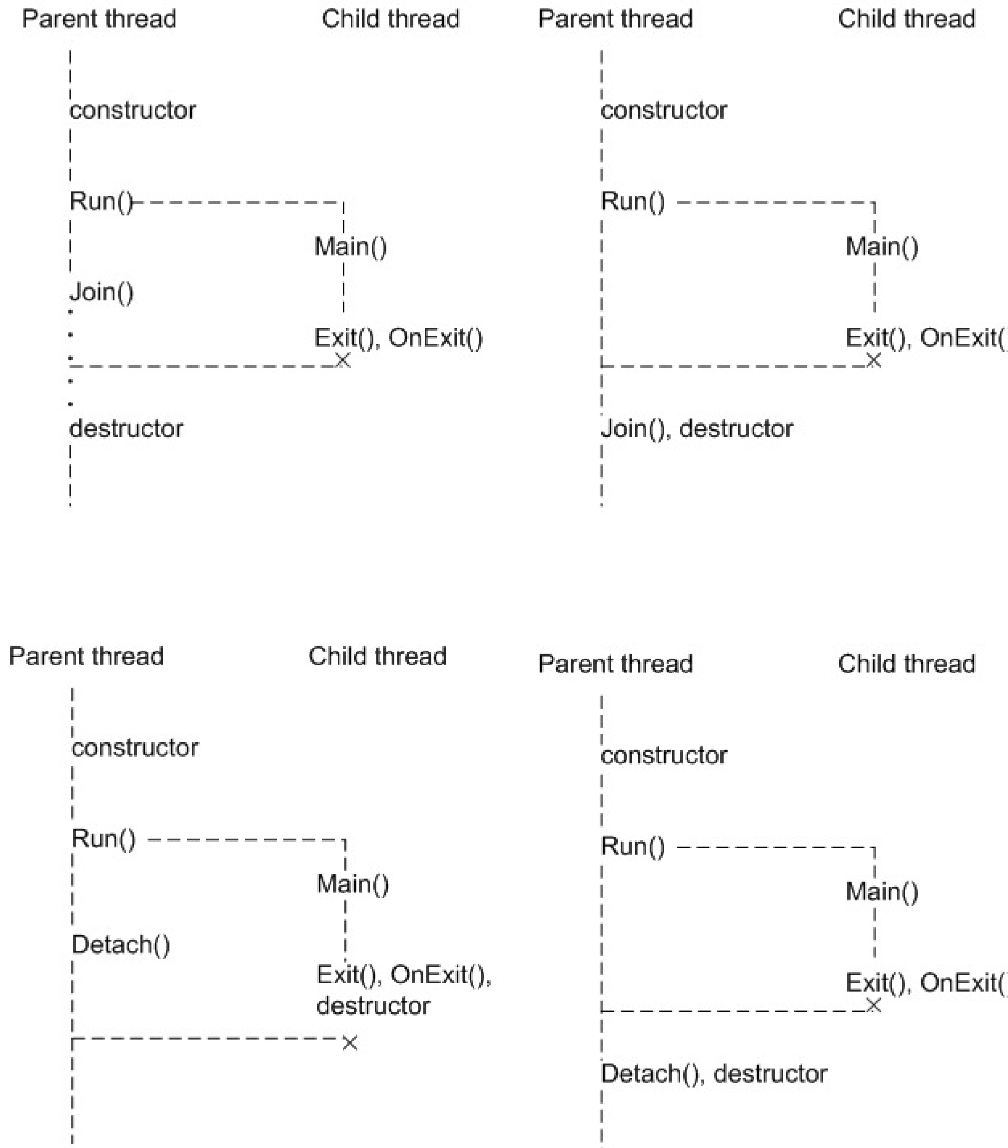

Figure 2. Thread Life Cycle

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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.

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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)

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

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

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

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