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

The NCBI C++ Toolkit

20: Using the Boost Unit Test Framework

Last Update: January 20, 2011.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discusses the Boost Unit Test Framework and how to use it within NCBI. The NCBI C++ Toolkit has incorporated and extended the open source Boost.Test Library, and provides a simplified way for the developers to create Boost-based C++ unit tests.

The NCBI extensions add the ability to:

- execute the code in a standard (*CNcbiApplication* -like) environment;
- disable test cases or suites, using one of several methods;
- establish dependencies between test cases and suites;
- use NCBI command-line argument processing;
- add initialization and finalization functions; and
- use convenience macros for combining NO_THROW with other test tools.

While the framework may be of interest to outside organizations, this chapter is intended for NCBI C++ developers. See also the Doxygen documentation for tests.

Chapter Outline

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

Why Use the Boost Unit Test Framework?

"*...I would like to see a practical plan for every group in Internal Services to move toward standardized testing. Then, in addition to setting an example for the other coding groups, I hope that you will have guidance for them as well about how best to move ahead in this direction. Once you have that, and are adhering to it yourselves, I will start pushing the other coding groups in that direction*."

- Jim Ostell, April 21, 2008

The value of unit testing is clearly recognized at the highest levels of management at NCBI. Here are some of the ways that using the Boost Unit Test Framework will directly benefit the developer:

- The framework provides a uniform (and well-supported) testing and reporting environment.
- Using the framework simplifies the process of creating and maintaining unit tests:
  - The framework helps keep tests well-structured, straightforward, and easily expandable.
  - You can concentrate on the testing of your functionality, while the framework takes care of all the testing infrastructure.
- The framework fits into the NCBI nightly build system:
  - All tests are run nightly on many platforms.
  - All results are archived and available through a web interface.

How to Use the Boost Unit Test Framework

This chapter assumes you are starting from a working Toolkit source tree. If not, please refer to the chapters on obtaining the source code, and configuring and building the Toolkit.

Creating a New Unit Test

On UNIX or MS Windows, use the new_project script to create a new unit test project:

```
new_project <name> app/unit_test
```

For example, to create a project named foo, type this in a command shell:

```
new_project foo app/unit_test
```

This creates a directory named foo and then creates two projects within the new directory. One project will be the one named on the command-line (e.g. foo) and will contain a sample unit test using all the basic features of the Boost library. The other project will be named unit_test_alt_sample and will contain samples of advanced techniques not required in most unit tests.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

You can build and run these projects immediately to see how they work:

```
cd foo
make
make check
```

Once your unit test is created, you must customize it to meet your testing requirements. This involves editing these files:

File	Purpose
Makefile	Main makefile for this directory - builds both the foo and unit_test_alt_sample unit tests.
Makefile.builddir	Contains the path to a pre-built C++ Toolkit.
Makefile.foo_app	Makefile for the foo unit test.
Makefile.in	
Makefile.unit_test_alt_sample_app	Makefile for the unit_test_alt_sample unit test.
foo.cpp	Source code for the foo unit test.
unit_test_alt_sample.cpp	Source code for the unit_test_alt_sample unit test.
unit_test_alt_sample.ini	Configuration file for the unit_test_alt_sample unit test.

Customizing an Existing Unit Test

This section contains the following topics:

Modifying the Makefile

The new_project script generates a new unit test project that includes everything needed to use the Boost Unit Test Framework, but it won't include anything specifically needed to build the library or application you are testing.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Therefore, edit the unit test makefile (e.g. Makefile.foo_app) and add the appropriate paths and libraries needed by your library or application. Note that while the new_project script creates five makefiles, you will generally need to edit only one.

By default (for users at NCBI) your unit test will be incorporated into the nightly test system. If you don't want it included, comment out the CHECK_CMD line in Makefile.foo_app. It is possible to put multiple CHECK_CMD lines in a single makefile - for example, if your unit test program needs to be run multiple times with different command-line arguments.

If you'd like to get nightly test results automatically emailed to you, add your email address to the WATCHERS macro in the makefile. Note that the WATCHERS macro has replaced the CHECK_AUTHORS macro which had a similar purpose.

If your project has a configuration file and/or any data files, add a CHECK_COPY line (see Makefile.unit_test_alt_sample_app for an example).

A unit test timeout can also be specified by adding a CHECK_TIMEOUT line to the makefile.

Note: Since the unit tests are based on Boost Unit Test Framework, the makefiles must specify:

```
REQUIRES = Boost.Test.Included
```

If you are using the new_project script (recommended), this setting is included automatically. Otherwise, make sure that Boost.Test.Included is listed in REQUIRES.

For more information about including your application in the nightly test suite, see the "Definition and running tests" section in the "Project Creation and Management" chapter.

Modifying the Source File

A unit test is simply a test of a unit of code, such as a class. Because each unit has many requirements, each unit test has many test cases. Your unit test code should therefore consist of a test case for each testable requirement. Each test case should be as small and independent of other test cases as possible. For information on how to handle dependencies between test cases, see the section on managing dependencies.

Starting with an existing unit test source file, simply add, change, or remove test cases as appropriate for your unit test. Test cases are defined by the BOOST_AUTO_TEST_CASE macro, which looks similar to a function. The macro has a single argument (the test case name) and a block of code that implements the test. Test case names must be unique at each level of the test suite hierarchy (see managing dependencies). Test cases should contain code that will succeed if the requirement under test is correctly implemented, and fail otherwise. Determination of success is made using Boost testing tools such as BOOST_REQUIRE and BOOST_CHECK.

The following sections discuss modifying the source file in more detail:

- Using Testing Tools
- Adding Initialization and/or Finalization
- Handling Timeouts
- Handling Command-Line Arguments in Test Cases
- Creating Test Suites
- Managing Dependencies

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

- Unit Tests with Multiple Files

Using Testing Tools

Testing tools are macros that are used to detect errors and determine whether a given test case passes or fails.

While at a basic level test cases can pass or fail, it is useful to distinguish between those failures that make subsequent testing pointless or impossible and those that don't. Therefore, there are two levels of testing: CHECK (which upon failure generates an error but allows subsequent testing to continue), and REQUIRE (which upon failure generates a fatal error and aborts the current test case). In addition, there is a warning level, WARN, that can report something of interest without generating an error, although by default you will have to set a command-line argument to see warning messages.

If the failure of one test case should result in skipping another then you should add a dependency between them.

Many Boost testing tools have variants for each error level. The most common Boost testing tools are:

Testing Tool	**Purpose**
BOOST_<level>(predicate)	Fails if the Boolean predicate (any logical expression) is false.
BOOST_<level>_EQUAL(left, right)	Fails if the two values are not equal.
BOOST_<level>_THROW(expression, exception)	Fails if execution of the expression doesn't throw an exception of the given type (or one derived from it).
BOOST_<level>_NO_THROW(expression)	Fails if execution of the expression throws any exception.

Note that BOOST_<level>_EQUAL(var1,var2) is equivalent to BOOST_<level>(var1==var2), but in the case of failure it prints the value of each variable, which can be helpful. Also, it is not a good idea to compare floating point values directly - instead, use BOOST_<level>_CLOSE(var1,var2,tolerance).

See the Boost testing tools reference page for documentation on these and other testing tools.

The NCBI extensions to the Boost library add a number of convenience testing tools that enclose the similarly-named Boost testing tools in a NO_THROW test:

Boost Testing Tool	**NCBI "NO_THROW " Extension**
BOOST_<level>(predicate)	NCBITEST_<level>(predicate)
BOOST_<level>_EQUAL(left, right)	NCBITEST_<level>_EQUAL(left, right)
BOOST_<level>_NE(left, right)	NCBITEST_<level>_NE(left, right)
BOOST_<level>_MESSAGE(pred, msg)	NCBITEST_<level>_MESSAGE(pred, msg)

Adding Initialization and/or Finalization

If your unit test requires initialization prior to executing test cases, or if finalization / clean-up is necessary, use these functions:

```
NCBITEST_AUTO_INIT()
{
 // Your initialization code here...
}

NCBITEST_AUTO_FINI()
{
 // Your finalization code here...
}
```

Handling Timeouts

If exceeding a maximum execution time constitutes a failure for your test case, use this:

```
// change the second parameter to the duration of your timeout in seconds
BOOST_AUTO_TEST_CASE_TIMEOUT(TestTimeout, 3);
BOOST_AUTO_TEST_CASE(TestTimeout)
{
 // Your test code here...
}
```

Handling Command-Line Arguments in Test Cases

It is possible to retrieve command-line arguments from your test cases using the standard C++ Toolkit argument handling API. The first step is to initialize the unit test to expect the arguments. Add code like the following to your source file:

```
NCBITEST_INIT_CMDLINE(descrs)
{
 // Add calls like this for each command-line argument to be used.
 descrs->AddOptionalPositional("some_arg",
 "Sample command-line argument.",
 CArgDescriptions::eString);
}
```

For more examples of argument processing, see test_ncbiargs_sample.cpp.

Next, add code like the following to access the argument from within a test case:

```
BOOST_AUTO_TEST_CASE(TestCaseName)
{
 const CArgs& args = CNcbiApplication::Instance()->GetArgs();
 string arg_value = args["some_arg"].AsString();
 // do something with arg_value ...
}
```

Adding your own command-line arguments will not affect the application's ability to process other command-line arguments such as -help or -dryrun.

Creating Test Suites

Test suites are simply groups of test cases. The test cases included in a test suite are those that appear between the beginning and ending test suite declarations:

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
BOOST_AUTO_TEST_SUITE(TestSuiteName)

BOOST_AUTO_TEST_CASE(TestCase1)
{
 //...
}

BOOST_AUTO_TEST_CASE(TestCase2)
{
 //...
}

BOOST_AUTO_TEST_SUITE_END();
```

Note that the beginning test suite declaration defines the test suite name and does not include a semicolon.

Managing Dependencies

Test cases and suites can be dependent on other test cases or suites. This is useful when it doesn't make sense to run a test after some other test fails:

```
NCBITEST_INIT_TREE()
{
 // define individual dependencies
 NCBITEST_DEPENDS_ON(test_case_dep, test_case_indep);
 NCBITEST_DEPENDS_ON(test_case_dep, test_suite_indep);
 NCBITEST_DEPENDS_ON(test_suite_dep, test_case_indep);
 NCBITEST_DEPENDS_ON(test_suite_dep, test_suite_indep);

 // define multiple dependencies
 NCBITEST_DEPENDS_ON_N(item_dep, 2, (item_indep1, item_indep2));
}
```

When an independent test item (case or suite) fails, all of the test items that depend on it will be skipped.

Unit Tests with Multiple Files

The new_project script is designed to create single-file unit tests by default, but you can add as many files as necessary to implement your unit test. Use of the BOOST_AUTO_TEST_MAIN macro is now deprecated.

Disabling Tests

The Boost Unit Test Framework was extended by NCBI to provide several ways to disable test cases and suites. Test cases and suites are disabled based on logical expressions in the application configuration file or, less commonly, by explicitly disabling or skipping them. The logical expressions are based on unit test variables which are defined either by the library or by the user. All such variables are essentially Boolean in that they are either defined (true) or not defined (false). Note: these methods of disabling tests don't apply if specific tests are run from the command-line.

- Disabling Tests with Configuration File Entries

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

- Library-Defined Variables
- User-Defined Variables
- Disabling or Skipping Tests Explicitly in Code

Disabling Tests with Configuration File Entries

The [UNITTESTS_DISABLE] section of the application configuration file can be customized to disable test cases or suites. Entries in this section should specify a test case or suite name and a logical expression for disabling it (expressions that evaluate to true disable the test). The logical expression can be formed from the logical constants true and false, numeric constants, library-defined or user-defined unit test variables, logical operators ('!', '&&', and '||'), and parentheses.

To disable specific tests, use commands like:

```
[UNITTESTS_DISABLE]
SomeTestCaseName = OS_Windows && PLATFORM_BigEndian
SomeTestSuiteName = (OS_Linux || OS_Solaris) && COMPILER_GCC
```

There is a special entry GLOBAL that can be used to disable all tests. For example, to disable all tests under Cygwin, use:

```
[UNITTESTS_DISABLE]
GLOBAL = OS_Cygwin
```

Note: If the configuration file contains either a test name or a variable name that has not been defined (e.g. due to a typo) then the test program will exit immediately with an error, without executing any tests.

Library-Defined Variables

When the NCBI-extended Boost Test library is built, it defines a set of unit test variables based on the build, compiler, operating system, and platform. See Table 1 for a list of related variables (test_boost.cpp has the latest list of variables).

At run-time, the library also checks the FEATURES environment variable and creates unit test variables based on the current set of features. See Table 2 for a list of feature, package, and project related variables (test_boost.cpp has the latest list of features).

The automated nightly test suite defines the FEATURES environment variable before launching the unit test applications. In this way, unit test applications can also use run-time detected features to exclude specific tests from the test suite.

Note: The names of the features are modified slightly when creating unit test variables from names in the FEATURES environment variable. Specifically, each feature is prefixed by FEATURE_ and all non-alphanumeric characters are changed to underscores. For example, to require the feature in-house-resources for a test (i.e. to disable the test if the feature is not present), use:

```
[UNITTESTS_DISABLE]
SomeTestCaseName = !FEATURE_in_house_resources
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

User-Defined Variables

You can define your own variables to provide finer control on disabling tests. First, define a variable in your source file:

```
NCBITEST_INIT_VARIABLES(parser)
{
 parser->AddSymbol("my_ini_var", <some bool expression goes here>);
}
```

Then add a line to the configuration file to disable a test based on the value of the new variable:

```
[UNITTESTS_DISABLE]
MyTestName = my_ini_var
```

User-defined variables can be used in conjunction with command-line arguments:

```
NCBITEST_INIT_VARIABLES(parser)
{
 const CArgs& args = CNcbiApplication::Instance()->GetArgs();
 parser->AddSymbol("my_ini_var", args["my_arg"].HasValue());
}
```

Then, passing the argument on the command-line controls the disabling of the test case:

```
./foo my_arg # test is disabled
./foo # test is not disabled (at least via command-line / config file)
```

Disabling or Skipping Tests Explicitly in Code

The NCBI extensions include a macro, NCBITEST_DISABLE, to unconditionally disable a test case or suite. This macro must be placed in the NCBITEST_INIT_TREE function:

```
NCBITEST_INIT_TREE()
{
 NCBITEST_DISABLE(test_case_name);
 NCBITEST_DISABLE(test_suite_name);
}
```

The extensions also include two functions for globally disabling or skipping all tests. These functions should be called only from within the NCBITEST_AUTO_INIT or NCBITEST_INIT_TREE functions:

```
NCBITEST_INIT_TREE()
{
 NcbiTestSetGlobalDisabled(); // A given unit test might include one
 NcbiTestSetGlobalSkipped(); // or the other of these, not both.
 // Most unit tests won't use either.
}
```

The difference between these functions is that globally disabled unit tests will report the status DIS to check scripts while skipped tests will report the status SKP.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Viewing Unit Tests Results from the Nightly Build

The Boost Unit Test Framework provides more than just command-line testing. Each unit test built with the framework becomes incorporated into nightly testing and is tested on multiple platforms and under numerous configurations. All such results are archived in the database and available through a web interface.

The main page (see Figure 1) provides many ways to narrow down the vast quantity of statistics available. The top part of the page allows you to select test date, test result, build configuration (branch, compiler, operating system, etc), debug/release, and more. The page also has a column for selecting tests, and a column for configurations. For best results, refine the selection as much as possible, and then click on the “See test statistics” button.

The “See test statistics” button retrieves the desired statistics in a second page (see Figure 2). The results are presented in tables: one for each selected date, with unit tests down the left side and configurations across the top. Further refinements of the displayed results can be made by removing rows, columns, or dates; and by selecting whether all columns, all cells, or only selected cells are displayed.

Each cell in the results tables represents a specific unit test performed on a specific date under a specific configuration. Clicking on a cell retrieves a third page (see Figure 3) that shows information about that test and its output.

Running Unit Tests from a Command-Line

To run one or more selected test cases from a command-line, use this:

```
./foo --run_test=TestCaseName1,TestCaseName2
```

Multiple test cases can be selected by using a comma-separated list of names.

To see all test cases in a unit test, use this:

```
./foo -dryrun
```

To see exactly which test cases passed and failed, use this:

```
./foo --report_level=detailed
```

To see warning messages, use this:

```
./foo --log_level=warning
```

Additional runtime parameters can be set. For a complete list, see the online documentation.

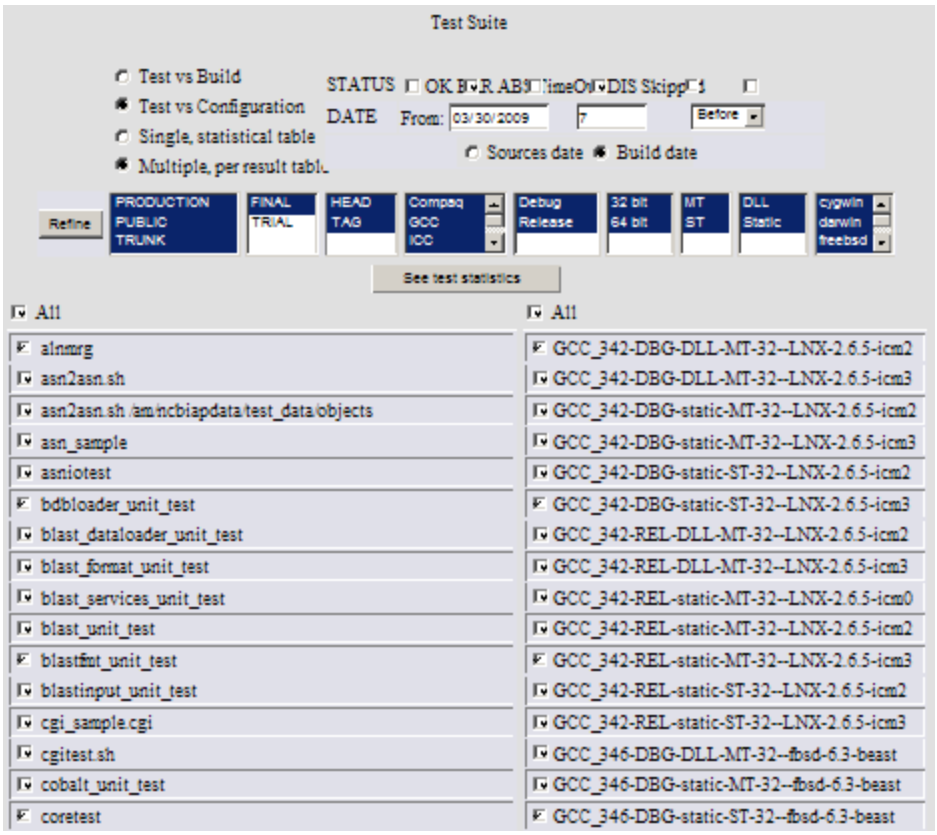
Limitations of the Boost Unit Test Framework

The currently known limitations are:

- It is not suitable for most multi-threaded tests.
- It is not suitable for "one-piece" applications (such as server or CGI). Such applications should be tested via their clients (which would preferably be unit test based).

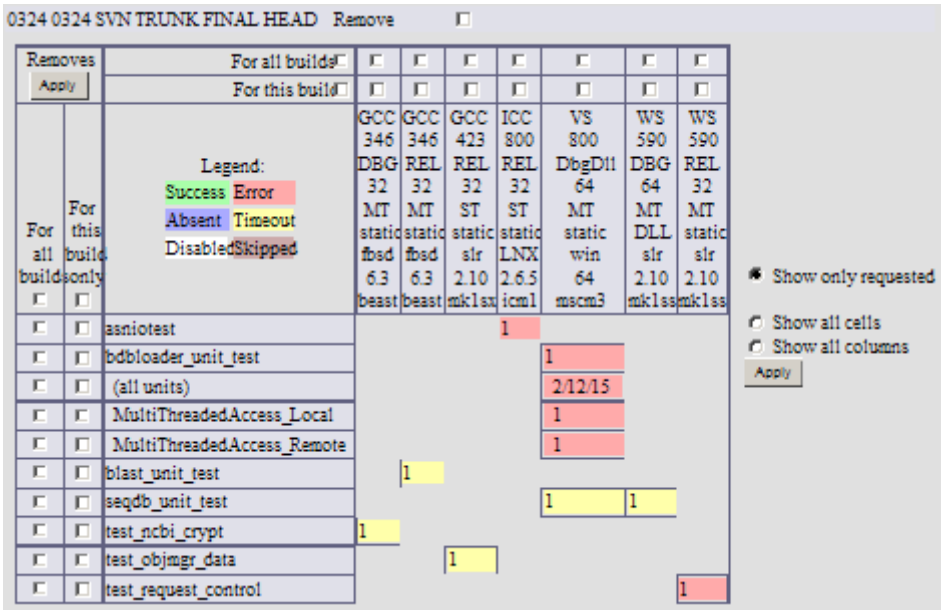
The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book



1. .

Figure 1. Test Interface



2. .

Figure 2. Test Matrix

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Test	algo/blast/api/unit_test/blast_unit_test
Build	2009-03-24 000301 TRUNK FINAL HEAD https://svn.ncbi.nlm.nih.gov/repos/toolkit/trunk/internal/c+
Configuration	GCC_346-Release-static-MT-32--freebsd-6.3-beastie
Run at	03/24/2009 152902
Result / Exit code	TO
Test timeout	750

Output:

```
--------------------------------------------------------------------------
blast_unit_test
--------------------------------------------------------------------------

terminate called after throwing an instance of 'ncbi::CSeqDBException'
  what():  NCBI C++ Exception:
    "/netopt/ncbi_tools/c++.by-date/20090324/GCC-ReleaseMT/../src/objtools/blast/seqdb_reader/seqdbalias.cpp", line 1


Maximum execution time of 750 seconds is exceeded

real 750.57
user 0.00
sys 0.01
Start time   : 03/24/2009 15:29:02
Stop time    : 03/24/2009 15:41:33

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$ EXIT CODE: 143
```

3. .

Figure 3. Test Result

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

Table 1. Build Generated Predefined Variables

Builds	**Compilers**	**Operating Systems**	**Platforms**
BUILD_Debug	COMPILER_Compaq	OS_AIX	PLATFORM_BigEndian
BUILD_Dll	COMPILER_GCC	OS_BSD	PLATFORM_Bits32
BUILD_Release	COMPILER_ICC	OS_Cygwin	PLATFORM_Bits64
BUILD_Static	COMPILER_KCC	OS_Irix	PLATFORM_LittleEndian
	COMPILER_MipsPro	OS_Linux	
	COMPILER_MSVC	OS_MacOS	
	COMPILER_VisualAge	OS_MacOSX	
	COMPILER_WorkShop	OS_Solaris	
		OS_Tru64	
		OS_Unix	
		OS_Windows	

Table 2. Check Script Generated Predefined Variables

Features	**Packages**	**Projects**
AIX	BerkeleyDB	algo
BSD	BerkeleyDB__ (use for BerkeleyDB++)	app
CompaqCompiler	Boost_Regex	bdb
Cygwin	Boost_Spirit	cgi
CygwinMT	Boost_Test	connext
DLL	Boost_Test_Included	ctools
DLL_BUILD	Boost_Threads	dbapi
Darwin	BZ2	gbench
GCC	C_ncbi	gui
ICC	C_Toolkit	local_bsm
in_house_resources	CPPUNIT	ncbi_crypt
IRIX	DBLib	objects
KCC	EXPAT	serial
Linux	Fast_CGI	
MIPSpro	FLTK	
MSVC	FreeTDS	
MSWin	FreeType	
MT	FUSE	
MacOS	GIF	
Ncbi_JNI	GLUT	
OSF	GNUTLS	
PubSeqOS	HDF5	
SRAT_internal	ICU	
Solaris	JPEG	
unix	LIBXML	
VisualAge	LIBXSLT	
WinMain	LocalBZ2	
WorkShop	LocalMSGMAIL2	
XCODE	LocalNCBILS	
	LocalPCRE	
	LocalSSS	
	LocalZ	
	LZO	

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

	MAGIC	
	MESA	
	MUPARSER	
	MySQL	
	NCBILS2	
	ODBC	
	OECHEM	
	OpenGL	
	OPENSSL	
	ORBacus	
	PCRE	
	PNG	
	PYTHON	
	PYTHON23	
	PYTHON24	
	PYTHON25	
	SABLOT	
	SGE	
	SP	
	SQLITE	
	SQLITE3	
	SQLITE3ASYNC	
	SSSDB	
	SSSUTILS	
	Sybase	
	SybaseCTLIB	
	SybaseDBLIB	
	TIFF	
	UNGIF	
	UUID	
	Xalan	
	Xerces	
	XPM	
	Z	
	wx2_8	

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

	wxWidgets	
	wxWindows	