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

The NCBI C++ Toolkit

30: FAQs, Useful Documentation Links, and Mailing Lists

Last Update: April 19, 2012.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter contains frequently asked questions and useful links.

Chapter Outline

FAQs

General

How do I prepare my development environment for using the C++ Toolkit?

That depends on your development environment and whether you are inside or outside of NCBI:

- Unix or Mac OS X inside NCBI
- Unix or Mac OX X outside NCBI
- Windows inside NCBI
- Windows outside NCBI

Unix or Mac OS X inside NCBI

All developer Unix accounts should be automatically prepared for using the C++ Toolkit. You should have a ~/.ncbi_hints file with a non-trivial facilities line that will be sourced when logging in. If everything is set up properly, the following commands should provide meaningful output:

```
svn --version
new_project
echo $NCBI
```

Unix or Mac OX X outside NCBI

After downloading the Toolkit source, set environment variable NCBI to <toolkit_root> (where <toolkit_root> is the top-level directory containing configure) and add $NCBI/scripts/common to your PATH.

Once the Toolkit is configured and built, then you can use it.

Windows inside NCBI

A supported version of MSVC must be installed.

A Subversion client must be installed. For help on that, please see \\snowman\win-coremake\App\ThirdParty\Subversion. To make sure subversion is working, enter svn --version in your cmd.exe shell.

Your PATH should include \\snowman\win-coremake\Scripts\bin.

If you want to step into the source for the C++ Toolkit libraries while debugging, then drive S: must be mapped to \\snowman\win-coremake\Lib. You can map it or let the new_project script map it for you.

Windows outside NCBI

A supported version of MSVC must be installed.

Download the Toolkit source.

Once the Toolkit is configured and built, then you can use it.

The GetTypeInfo() method is not declared or defined in any of the objects for which it is part of the interface

The macro DECLARE_INTERNAL_TYPE_INFO() is used in the *.hpp files to declare the GetTypeInfo(). There are several macros that are used to implement GetTypeInfo() methods in *.cpp files. These macros are generally named and used as follows:

```
BEGIN_*_INFO(...)
{
 ADD_*(...)
 ...
}
```

See User-defined Type Information in the Programming Manual for more information.

*Which include file should be used in *.cpp files, class.hpp or class_.hpp?*

Include the **class**.hpp (file without underscore). Never instantiate or use a class of the form C*_Base directly. Instead use the C* form which inherits from the C*_Base class (e.g., don't use CSeq_id_Base directly -- use CSeq_id instead).

How can I disable the XML declaration or DOCTYPE with the serial library?

Here's a code snippet that shows all combinations:

```
// serialize XML with both an XML declaration and with a DOCTYPE (default)
ostr << MSerial_Xml << obj;

// serialize XML without an XML declaration
ostr << MSerial_Xml(fSerial_Xml_NoXmlDecl) << obj;

// serialize XML without a DOCTYPE
ostr << MSerial_Xml(fSerial_Xml_NoRefDTD) << obj;

// serialize XML without either an XML declaration or a DOCTYPE
ostr << MSerial_Xml(fSerial_Xml_NoXmlDecl | fSerial_Xml_NoRefDTD) << obj;
```

Note: The serial library can read XML whether or not it contains the XML declaration or DOCTYPE without using special flags. For example:

```
istr >> MSerial_Xml >> obj;
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Compiling

How do I compile for 32-bit on a 64-bit machine in a typical C++ Toolkit app?

Our 64-bit Linux systems only support building 64-bit code; to produce 32-bit binaries, you'll need a 32-bit system.

Which Visual C++ project should I build?

After creating a new project, you may notice quite a few projects appear in the solution, besides your program, and that the **-HIERARCHICAL-VIEW-** project is bold (indicating that it's the startup project). Do not build any of these projects or the solution as a whole. Instead, set your program as the default startup project and build it.

You can build **-CONFIGURE-DIALOG-** if you need to reconfigure your project (see the section on using the configuration GUI), and you will need to build **-CONFIGURE-** if you add libraries (see the question below on adding a library to a Visual C++ project).

What compiler options are required to be compatible with C++ Toolkit libraries?

These compiler options must be properly set under Microsoft Visual C++:

- Checked iterators
- C++ exceptions
- Runtime library

Checked iterators

Microsoft Visual C++ provides the option of using "Checked Iterators" to ensure that you do not overwrite the bounds of your STL containers. Checked iterators have a different internal structure than, and are therefore incompatible with, non-checked iterators. If both are used in the same program, it will probably crash. Checked iterators also have somewhat lower performance than non-checked iterators.

Therefore, when building with Visual C++, you must ensure that the same checked iterators setting is used for all compilation units that use STL iterators. This includes the Visual C++ standard libraries, the NCBI C++ Toolkit libraries, your code, and any other libraries you link with.

To disable checked iterators, set _SECURE_SCL=0; to enable them, set _SECURE_SCL=1.

The Visual C++ defaults for _SECURE_SCL are:

Visual C++ Version	Debug	Release
2010	1	0
2008	1	1

By default, the compiler options for NCBI C++ Toolkit libraries do not specify the _SECURE_SCL option for debug configurations, and specify _SECURE_SCL=0 for release configurations. Therefore they use checked iterators for debug configurations, but not for release configurations.

Note: Your code may crash if any two libraries you link with don't use the same settings. For example:

- You're building a release configuration using Visual C++ 2008. You build the C++ Toolkit separately and use it as a third party package (in which case it will use _SECURE_SCL=0). Your other code and/or other libraries are compiled with default settings (which for release in VS2008 sets _SECURE_SCL=1).
- You're using a third party library that uses different settings than the C++ Toolkit.

If you need to use a different setting for _SECURE_SCL than the Toolkit uses, you will have to recompile all Toolkit libraries that you want to link with. To change this setting and rebuild the Toolkit:

1. Open src\build-system\Makefile.mk.in.msvc.
2. Edit the PreprocessorDefinitions entry in the [Compiler.*release] section for the desired configuration(s), using _SECURE_SCL=0; or _SECURE_SCL=1;.
3. Build the -CONFIGURE- project in the solution that contains all the Toolkit libraries you want to use. See the section on choosing a build scope for tips on picking the solution. Ignore the reload solution prompts - when the build completes, then close and reopen the solution.
4. Build the -BUILD-ALL- project to rebuild the libraries.

A similar situation exists for the _HAS_ITERATOR_DEBUGGING macro, however the C++ Toolkit does not set this macro for either 2008 or 2010, so you are unlikely to encounter any problems due to this setting. It's possible (however unlikely) that other third party libraries could turn this macro off in debug, in which case you'd have to rebuild so the settings for all libraries match.

By default, _HAS_ITERATOR_DEBUGGING is turned on in debug but can be turned off. However, it cannot be turned on in release.

Finally, the macro _ITERATOR_DEBUG_LEVEL was introduced with Visual C++ 2010 to simplify the use of _SECURE_SCL and _HAS_ITERATOR_DEBUGGING.

If you set _ITERATOR_DEBUG_LEVEL, then _SECURE_SCL and _HAS_ITERATOR_DEBUGGING will be set according to this table:

_ITERATOR_DEBUG_LEVEL	_SECURE_SCL	_HAS_ITERATOR_DEBUGGING
0	0	0
1	1	0
2	1	1

If you don't set _ITERATOR_DEBUG_LEVEL, it will be set automatically according to the values of _SECURE_SCL and _HAS_ITERATOR_DEBUGGING per the above table. Therefore, you can use either _ITERATOR_DEBUG_LEVEL or _SECURE_SCL and _HAS_ITERATOR_DEBUGGING as you see fit. In most cases, you won't need to set any of them. You just need to know about them in case you link with libraries that use different settings.

For more information, see:

- Checked Iterators
- What's New in Visual C++
- Breaking Changes in Visual C++

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

C++ exceptions

NCBI C++ Toolkit libraries use the /EHsc compiler option with Visual C++ to:

- ensure that C++ objects that will go out of scope as a result of the exception are destroyed;
- ensure that only C++ exceptions should be caught; and
- assume that extern C functions never throw a C++ exception.

For more information, see the MSDN page on /EH.

Runtime library

You must specify the appropriate Visual C++ runtime library to link with:

Configuration	Compiler Option
DebugDLL	/MDd
DebugMT	/MTd
ReleaseDLL	/MD
ReleaseMT	/MT

For more information, see the MSDN page on runtime library options.

Linking

How does one find the libraries to link when the linker complains of undefined symbols?

Two tools are available to resolve the common linking questions:

Question	Tool
Which libraries contain a given symbol?	Library search
Which other libraries does a given library depend on?	Library dependencies

For example, suppose the linker complains about the symbol ncbi::CIStreamBuffer::FindChar (char) being undefined. Here is how to use these tools to resolve this problem:

1. To find the library(s) where the unresolved symbol is defined, use the Library search tool:

   Using the example above, enter FindChar as a search term. The library where this symbol is defined is libxutil.a (ncbi_core).

   Now that you have the library that defines the symbol, you can proceed to find the library dependencies it introduces. Note: The simplest way to do this is by just clicking on the library in the search results to show its dependencies. Alternatively, you can proceed to step 2.

2. The Library dependencies tool finds all the other libraries that a given library depends on. This tool can also help you create the LIB and LIBS lines in your makefile. For example, enter your current LIB and LIBS lines plus the new library from step 1, and it will generate optimized LIB and LIBS lines containing the library needed for your symbol and any other needed libraries.

Continuing with the example above, entering libxutil.a (or just xutil) will create this result:

```
LIB  = xutil xncbi
LIBS = $(ORIG_LIBS)
```

Clicking on any of the links will show the required items for that link plus a dependency graph for the clicked item. The nodes in the diagram are also navigable.

Note: If you are using Visual C++, please also see the question about adding libraries to Visual C++ projects.

To make it easier to work with the NCBI C++ Toolkit's many libraries, we have generated illustrations of their dependency relationships, available for various scopes and in various formats:

NCBI C++ Library Dependency Graphs (including internal libraries)

	GIF	PNG	PDF	PostScript	Text
All libraries			PDF	PS	TXT
Just C++ Toolkit libraries			PDF	PS	
Highly connected or otherwise noteworthy public libraries	GIF	PNG	PDF	PS	

NCBI C++ Library Dependency Graphs (public libraries only)

	GIF	PNG	PDF	PostScript	Text
All libraries			PDF	PS	TXT
Non-GUI libraries			PDF	PS	
GUI libraries	GIF	PNG	PDF	PS	
Highly connected or otherwise noteworthy public libraries	GIF	PNG	PDF	PS	

In cases where the above methods do not work, you can also search manually using the following steps:

1. Look for the source file that defines the symbol. This can be done by going to the LXR source browser and doing an identifier search on the symbol (e.g., CDate or XmlEncode). Look for a source file where the identifier is defined (e.g. in the "Defined as a class in" section for CDate, or in the "Defined as a function in" section for XmlEncode()). For serializable object classes (such as CDate) look for the base class definition. Follow a link to this source file.
2. Near the top of the LXR page for the source file is a path, and each component of the path links to another LXR page. Click the link to the last directory.
3. The resulting LXR page for the directory should list the makefile for the library of interest (e.g. Makefile.general.lib for CDate or Makefile.corelib.lib for XmlEncode ()). Click on the link to the makefile. You should see the LIB line with the name of the library that contains your symbol.
4. Add the library name to the list of libraries you already have and enter them into the library dependencies tool to create your final LIB and LIBS lines.

In some cases, the library name is a variant on the subdirectory name. These variants are summarized in Table 1.

Most often, difficulties arise when one is linking an application using the numerous "objects/" libraries. To give you some relief, here are some examples involving such libraries. They show the right order of libraries, as well as which libraries you may actually need. Using this as a starting point, it's **much** easier to find the right combination of libraries:

- first, to find and add missing libraries using the generic technique described above
- then, try to throw out libraries which you believe are not actually needed

```
LIB = id1 seqset $(SEQ_LIBS) pub medline biblio general \
 xser xconnect xutil xncbi
LIB = ncbimime cdd cn3d mmdb scoremat seqset $(SEQ_LIBS) \
 pub medline biblio general xser xutil xncbi
```

How do I add a library to a Visual C++ project?

If you are using Visual C++, you should add the appropriate LIB and LIBS lines to the Makefile.<your_project>.app file located in the source directory, then build the **-CONFIGURE-** target, then close and reopen the solution. This process will update the project properties with the proper search directories and required libraries.

Linker complains it "cannot find symbol" in something like: "SunWS_cache/CC_obj_b/bXmZkg3zX5VBJvYgjABX.o"

Go to the relevant build dir, clean and rebuild everything using:

```
cd /home/qqq/c++/WorkShop6-Debug/build/FooBar
make purge_r all_r
```

MAKE complains it does not know "how to make target: /home/qqq/c++/WorkShop6-Debug/lib/.seqset.dep"

This means that the "libseqset.a" library is not built. To build it:

```
cd /home/qqq/c++/WorkShop6-Debug/build/objects/seqset
make
```

Still getting bizarre errors with unresolved symbols, unfound libraries, etc., and nothing seems to help out much

As the last resort, try to CVS update, reconfigure, clean and rebuild everything:

```
cd /home/qqq/c++/
cvs -q upd -d
compilers/WorkShop6.sh 32 ........
make purge_r
make all_r
```

Debugging

Debugger (DBX) warns it "cannot find file /home/coremake/c++/foobar.cpp", then it does not show source code

This happens when you link to the public C++ Toolkit libraries (from "$NCBI/c++/*/lib/"), which are built on other hosts and thus hard-coded with the source paths on these other hosts. All you have to do is to point DBX to the public sources (at "$NCBI/c++") by just adding to your DBX resource file (~/.dbxrc) the following lines:

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
pathmap /home/coremake/c++ /netopt/ncbi_tools/c++
pathmap /home/coremake/c++2 /netopt/ncbi_tools/c++
pathmap /home/coremake/c++3 /netopt/ncbi_tools/c++
pathmap /j/coremake/c++ /netopt/ncbi_tools/c++
pathmap /j/coremake/c++2 /netopt/ncbi_tools/c++
pathmap /j/coremake/c++3 /netopt/ncbi_tools/c++
```

ASN

Creating an out-of-tree application that uses your own local ASN.1 spec and a pre-built C++ Toolkit

Lets say you have your ASN.1 specification (call it foo.asn) and now you want to build an application (call its source code foo_main.cpp) which performs serialization of objects described in foo.asn. To complicate things, lets also assume that your ASN.1 spec depends on (imports) one of the ASN.1 specs already in the C++ Toolkit, like Date described in the NCBI-General module of general.asn. For example, your foo.asn could look like:

```
NCBI-Foo DEFINITIONS ::=
BEGIN
EXPORTS Foo;
IMPORTS Date FROM NCBI-General;
Foo ::= SEQUENCE {
 str VisibleString,
 date Date
}
END
```

Now, lets assume that the pre-built version of the NCBI C++ Toolkit is available at $NCBI/c++, and that you want to use the Toolkit's pre-built sources and libraries in your application. First, generate (using datatool) the serialization sources, and create the serialization library:

```
## Create new project directory, with a model makefile for your
## local ASN.1 serialization library, and copy "foo.asn"
cd ~/tmp
new_project foo lib/asn
cd foo
cp /bar/bar/bar/foo.asn .

## Using DATATOOL, generate data serialization sources for your
## ASN.1 specs described in "foo.asn":
datatool -oR $NCBI/c++ -m foo.asn \
 -M "objects/general/general.asn" -oA -oc foo -opc . -oph .

## Adjust in the library makefile "Makefile.foo.lib"
SRC = foo__ foo___

## Build the library
make -f Makefile.foo_lib
```

Then, create and build the application:

```
## Create new application project, and copy your app sources.
new_project foo_main app
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
cd foo_main
cp /bar/bar/bar/foo_main.cpp .

## Adjust the application makefile "Makefile.foo_main.app"
PRE_LIBS = -L.. -lfoo
CPPFLAGS = -I.. $(ORIG_CPPFLAGS)
LIB = general xser xutil xncbi

## Build the application
make -f Makefile.foo_main_app
```

How to add new ASN.1 specification to the C++ Toolkit?

Caution! If you are not in the C++ core developers group, please do not do it yourself! -- instead, just send your request to cpp-core@ncbi.nlm.nih.gov.

Converting ASN.1 object in memory from C to C++ representation (or vice versa)

The C++ Toolkit header ctools/asn_converter.hpp now provides a template class (CAsnConverter<>) for this exact purpose.

Useful Documentation Links

- [Doc] ISO/ANSI C++ Draft Standard Working Papers (Intranet only)
- [Doc] MSDN Online Search
- [Literature] Books and links to C++ and STL manuals
- [Example] NCBI C++ makefile hierarchy for project "corelib/"
- [Chart] NCBI C++ source tree hierarchy
- [Chart] NCBI C++ build tree hierarchy
- [Chart] NCBI C++ Library Dependency graph
- [Doc] NCBI IDX Database Documentation (Intranet only)
- [Doc] Documentation styles

Mailing Lists

- Announcements: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-announce (read-only)
- Everybody: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp
- Core developers: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-core
- Object Manager: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-objmgr
- GUI: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-gui
- SVN and CVS logs: http://www.ncbi.nlm.nih.gov/mailman/listinfo/cpp-cvs (read-only)

Internal mailing lists are also available to those inside NCBI.

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

Table 1. Examples where the library name is a variant on the sub directory name

Directory	**Library**
corelib/test	test_mt
corelib	xncbi
ctools/asn	xasn
cgi	xcgi or xfcgi
connect	xconnect
connect/test	xconntest
ctools	xctools
html	xhtml
objects/mmdb{1,2,3}	mmdb (consolidated)
objects/seq{,align,block,feat,loc,res}	seq (consolidated) or $(SEQ_LIBS)
objmgr	xobjmgr
objmgr/util	xobjutil
objtools/alnmgr	xalnmgr
serial	xser
util	xutil