The NCBI C++ Toolkit

6: Project Creation and Management

Last Update: July 10, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

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

Chapter Outline

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

Starting New Projects

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

The following topics are discussed in this section:

- New Projects: Location and File Structure
    - new_project: Starting a New Project outside the C++ Toolkit Tree
    - Creating a New Project Inside the C++ Toolkit Tree
- Projects and the Toolkit's SVN Tree Structure
- Creating source and include SVN dirs for a new C++ project
- Starting New Modules
- Meta-makefiles (to provide multiple and/or recursive builds)
- Project makefiles
    - Example 1: Customized makefile to build a library
    - Example 2: Customized makefile to build an application
    - Example 3: User-defined makefile to build... whatever
- An example of the NCBI C++ makefile hierarchy ("corelib/")

New Projects: Location and File Structure

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

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

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

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

The following topics are discussed in this section:

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

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

Script usage:

```
new_project <name> <type>[/<subtype>] [builddir]
```

NOTE: in NCBI, you can (and should) invoke common scripts simply by name - i.e. without path or extension. The proper script located in the pre-built NCBI C++ toolkit directory will be invoked.

This script will create a startup makefile for a new project which uses the NCBI C++ Toolkit (and possibly the C Toolkit as well). Replace <type> with lib for libraries or app for applications.

Sample code will be included in the project directory for new applications. Different samples are available for type=app[/basic] (a command-line argument demo application based on the corelib library), type=app/cgi (for a CGI or Fast-CGI application), type=app/objmgr (for an application using the Object Manager), type=app/objects (for an application using ASN.1 objects), and many others.

You will need to slightly edit the resultant makefile to:

- specify the name of your library (or application)
- specify the list of source files going to it
- modify some preprocessor, compiler, etc. flags, if needed
- modify the set of additional libraries to link to it (if it's an application), if needed

For example:

```
new_project foo app/basic
```

creates a model makefile Makefile.foo_app to build an application using tools and flags hard-coded in $NCBI/c++/Debug/build/Makefile.mk, and headers from $NCBI/c++/include/. The file /tmp/foo/foo.cpp is also created; you can either replace this with your own foo.cpp or modify its sample code as required.

Now, after specifying the application name, list of source files, etc., you can just go to the created working directory foo/ and build your application using:

```
make -f Makefile.foo_app
```

You can easily change the active version of NCBI C++ Toolkit by manually setting variable $(builddir) in the file Makefile.foo_app to the desired Toolkit path, e.g.,

```
builddir = $(NCBI)/c++/GCC-Release/build
```

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

Note for users inside NCBI: To be able to view debug information in the Toolkit libraries for Windows builds, you will need to have the S: drive mapped to \\snowman\win-coremake\Lib. By default, new_project will make this mapping for you if it's not already done.

Creating a New Project Inside the C++ Toolkit Tree

To create your new project (e.g., "bar_proj") directories in the NCBI C++ Toolkit source tree (assuming that the entire NCBI C++ Toolkit has been checked out to directory foo/c++/):

```
cd foo/c++/include && mkdir bar_proj && svn add bar_proj
cd foo/c++/src && mkdir bar_proj && svn add bar_proj
```

From there, you can now add and edit your project C++ files.

NOTE: remember to add this new project directory to the $(SUB_PROJ) list of the upper level meta-makefile configurable template (e.g., for this particular case, to foo/c++/src/Makefile.in).

Projects and the Toolkit's SVN Tree Structure

(For the overall NCBI C++ SVN tree structure see SVN details.)

Even if you work outside of the C++ tree, it is necessary to understand how the Toolkit uses makefiles, meta-makefiles, and makefile templates, and the SVN tree structure.

The standard SVN location for NCBI C++/STL projects is $SVNROOT/internal/c++/. Public header files (*.hpp, *.inl) of all projects are located below the $SVNROOT/internal/c++/include/ directory. $SVNROOT/internal/c++/src/ directory has just the same hierarchy of subdirectories as .../include/, and its very top level contains:

- Templates of generic makefiles (Makefile.*.in):
    - Makefile.in -- makefile to perform a recursive build in all project subdirectories
    - Makefile.meta.in -- included by all makefiles that provide both local and recursive builds
    - Makefile.lib.in -- included by all makefiles that perform a "standard" library build, when building only static libraries.

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

• The contents of each project are detailed above. If your project is to become part of the Toolkit tree, you need to ensure that all makefiles and Makefile*.in templates are available so the master makefiles can properly configure and build it (see "Meta-Makefiles" and "Project Makefiles" below). You will also need to prepare SVN directories to hold the new source and header files.

Creating source and include SVN dirs for a new C++ project

To create your new project (e.g., "bar_proj") directories in the NCBI C++ SVN tree to directory foo/c++/):

```
cd foo/c++/include && mkdir bar_proj && SVN add -m "Project Bar" bar_proj
cd foo/c++/src && mkdir bar_proj && SVN add -m "Project Bar" bar_proj
```

Now you can add and edit your project C++ files in there.

NOTE: remember to add this new project directory to the $(SUB_PROJ) list of the upper level meta-makefile configurable template (e.g., for this particular case, to foo/c++/src/Makefile.in).

Starting New Modules

Projects in the NCBI C++ Toolkit consist of “modules”, which are most often a pair of source (*.cpp) and header (*.hpp) files. To help create new modules, template source and header files may be used, or you may modify the sample code generated by the script new_project. The template source and header files are .../doc/public/framewrk.cpp and .../doc/public/framewrk.hpp. The template files contain a standard startup framework so that you can just cut-and-paste them to start a new module (just don't forget to replace the "framewrk" stubs by your new module name).

• Header file (*.hpp) -- API for the external users. Ideally, this file contains only (well-commented) declarations and inline function implementations for the public interface. No less, and no more.

• Source file (*.cpp) -- Definitions of non-inline functions and internally used things that should not be included by other modules.

On occasion, a second private header file is required for good encapsulation. Such second headers should be placed in the same directory as the module source file.

Each and every source file **must** include the NCBI disclaimer and (preferably) Subversion keywords (e.g. $Id$). Then, the header file must be protected from double-inclusion, and it must define any inlined functions.

Meta-makefiles (to provide multiple and/or recursive builds)

All projects from the NCBI C++ hierarchy are tied together by a set of meta-makefiles which are present in all project source directories and provide a uniform and easy way to perform both local and recursive builds. See more detail on the Working with Makefiles page. A typical meta-makefile template (e.g. Makefile.in in your foo/c++/src/bar_proj/ dir) looks like that:

```
# Makefile.bar_u1, Makefile.bar_u2 ...
USR_PROJ = bar_u1 bar_u2 ...
# Makefile.bar_l1.lib, Makefile.bar_l2.lib ...
LIB_PROJ = bar_l1 bar_l2 ...
# Makefile.bar_a1.app, Makefile.bar_a2.app ...
APP_PROJ = bar_a1 bar_l2 ...
SUB_PROJ = app sub_proj1 sub_proj2
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

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

Project makefiles

Just like the configurable template Makefile.meta.in is used to ease and standardize the writing of meta-makefiles, so there are templates to help in the creation of "regular" project makefiles to build a library or an application. These auxiliary template makefiles are described on the "Working with Makefiles" page and listed above. The configure'd versions of these templates get put at the very top of a build tree.

In addition to the meta-makefile that must be defined for each project, a customized makefile Makefile.<project_name>.[app|lib] must also be provided. The following three sections give examples of customized makefiles for a library and an application, along with a case where a user-defined makefile is required.

You have great latitude in specifying optional packages, features and projects in makefiles. The macro REQUIRES in the examples is one way to allows you access them. See the "Working with Makefiles" page for a complete list; the configuration page gives the corresponding configure options.

The following examples are discussed in this section:

- Example 1: Customized makefile to build a library
- Example 2: Customized makefile to build an application
- Example 3: User-defined makefile to build... whatever

Example 1: Customized makefile to build a library

Here is an example of a customized makefile to build library libxmylib.a from two source files xmy_src1.cpp and xmy_src2.c, and one pre-compiled object file some_obj1.o. To make the example even more realistic, we assume that the said source files include headers from the NCBI C Toolkit.

```
LIB = xmylib
SRC = xmy_src1 xmy_src2
OBJ = some_obj1
REQUIRES = xrequirement
CFLAGS = $(ORIG_CFLAGS) -abc -DFOOBAR_NOT_CPLUSPLUS
CXXFLAGS = $(FAST_CXXFLAGS) -xyz
CPPFLAGS = $(ORIG_CPPFLAGS) -UFOO -DP1_PROJECT -I$(NCBI_C_INCLUDE)
```

- Skip building this library if xrequirement (an optional package or project) is disabled or unavailable.
- Compile xmy_src1.cpp using the C++ compiler $(CXX) with the flags $(FAST_CXXFLAGS) -xyz $(CPPFLAGS), which are the C++ flags for faster code, some additional flags specified by the user, and the original preprocessor flags.
- Compile xmy_src2.c using the C compiler $(CC) with the flags $(ORIG_CFLAGS) -abc -DFOOBAR_NOT_CPLUSPLUS $(CPPFLAGS), which are the original C flags, some additional flags specified by the user, and the original preprocessor flags.
- Using $(AR) and $(RANLIB) [$(LINK_DLL) if building a shared library], compose the library libxmylib.a [libxmylib.so] from the resultant object files, plus the pre-compiled object file some_obj1.o.
- Copy libxmylib.* to the top-level lib/ directory of the build tree (for the later use by other projects).

This customized makefile should be referred to as xmylib in the LIB_PROJ macro of the relevant meta-makefile. As usual, Makefile.mk will be implicitly included.

This customized makefile can be used to build both static and dynamic (DLL) versions of the library. To encourage its build as a DLL on the capable platforms, you can explicitly specify:

```
LIB_OR_DLL = dll
```

or

```
LIB_OR_DLL = both
```

Conversely, if you want the library be always built as static, specify:

```
LIB_OR_DLL = lib
```

Example 2: Customized makefile to build an application

Here is an example of a customized makefile to build the application my_exe from three source files, my_main.cpp, my_src1.cpp, and my_src2.c. To make the example even more realistic, we assume that the said source files include headers from the NCBI SSS DB packages, and the target executable uses the NCBI C++ libraries libxmylib.* and libxncbi.*, plus NCBI SSS DB, SYBASE, and system network libraries. We assume further that the user would prefer to link statically against libxmylib if building the toolkit as both shared and static libraries (configure --with-dll --with-static ...), but is fine with a shared libxncbi.

```
APP = my_exe
SRC = my_main my_src1 my_src2
OBJ = some_obj
LIB = xmylib$(STATIC) xncbi
```

```
REQUIRES = xrequirement
CPPFLAGS = $(ORIG_CPPFLAGS) $(NCBI_SSSDB_INCLUDE)
LIBS = $(NCBI_SSSDB_LIBS) $(SYBASE_LIBS) $(NETWORK_LIBS) $(ORIG_LIBS)
```

- Skip building this library if xrequirement (an optional package or project) is disabled or unavailable.
- Compile my_main.cpp and my_src1.cpp using the C++ compiler $(CXX) with the flags $(CXXFLAGS) (see Note below).
- Compile my_src2.c using the C compiler $(CC) with the flags $(CFLAGS) (see Note below).
- Using $(CXX) as a linker, build an executable my_exe from the object files my_main.o, my_src1.o, my_src2.o, the precompiled object file some_obj.o, NCBI C++ Toolkit libraries libxmylib.a and libxncbi.*, and NCBI SSS DB, SYBASE, and system network libraries (see Note below).
- Copy the application to the top-level bin/ directory of the build tree (for later use by other projects).

Note: Since we did not redefine CFLAGS, CXXFLAGS, or LDFLAGS, their default values ORIG_*FLAGS (obtained during the build tree configuration) will be used.

This customized makefile should be referred to as my_exe in the APP_PROJ macro of the relevant meta-makefile. Note also, that the Makefile.mk will be implicitly included.

Example 3: User-defined makefile to build... whatever

In some cases, we may need more functionality than the customized makefiles (designed to build libraries and applications) can provide.

So, if you have a "regular" non-customized user makefile, and you want to make from it, then you must enlist this user makefile in the USR_PROJ macro of the project's meta-makefile.

Now, during the project build (and before any customized makefiles are processed), your makefile will be called with one of the standard make targets from the project's build directory. Additionally, the builddir and srcdir macros will be passed to your makefile (via the make command line).

In most cases, it is necessary to know your "working environment"; i.e., tools, flags and paths (those that you use in your customized makefiles). This can be easily done by including Makefile.mk in your makefile.

Shown below is a real-life example of a user makefile:

- build an auxiliary application using the customized makefile Makefile.hc_gen_obj.app (this part is a tricky one...)
- use the resultant application $(bindir)/hc_gen_obj to generate the source and header files humchrom_dat.[ch] from the data file humchrom.dat
- use the script $(top_srcdir)/scripts/if_diff.sh to replace the previous copies (if any) of humchrom_dat.[ch] with the newly generated versions if and only if the new versions are different (or there were no old versions).

And, of course, it provides build rules for all the standard make targets.

```
File $(top_srcdir)/src/internal/humchrom/Makefile.hc_gen_obj:
# Build a code generator for hard-coding the chrom data into
# an obj file
# Generate header and source "humchrom_dat.[ch]" from data
# file "humchrom.dat"
# Deploy the header to the compiler-specific include dir
# Compile source code
#################################
include $(builddir)/Makefile.mk
BUILD__HC_GEN_OBJ = $(MAKE) -f "$(builddir)/Makefile.app.tmpl" \
srcdir="$(srcdir)" TMPL="hc_gen_obj" $(MFLAGS)
all_r: all
all: build_hc_gen_obj humchrom_dat.dep
purge_r: purge
purge: x_clean
 $(BUILD__HC_GEN_OBJ) purge
clean_r: clean
clean: x_clean
 $(BUILD__HC_GEN_OBJ) clean
x_clean:
 -rm -f humchrom_dat.h
 -rm -f humchrom_dat.c
build_hc_gen_obj:
 $(BUILD__HC_GEN_OBJ) all
humchrom_dat.dep: $(srcdir)/data/humchrom.dat $(bindir)/hc_gen_obj
 -cp -p humchrom_dat.c humchrom_dat.save.c
 $(bindir)/hc_gen_obj -d $(srcdir)/data/humchrom.dat
 -f humchrom_dat
 $(top_srcdir)/scripts/if_diff.sh "mv" humchrom_dat.h
 $(incdir)/humchrom_dat.h
 -rm humchrom_dat.h
 $(top_srcdir)/scripts/if_diff.sh "mv" humchrom_dat.c
 humchrom_dat.save.c
 mv humchrom_dat.save.c humchrom_dat.c
 touch humchrom_dat.dep
```

An example of the NCBI C++ makefile hierarchy ("corelib/")

See also the source and build hierarchy charts.

c++/src/Makefile.in:

```
SUB_PROJ = corelib cgi html @serial@ @internal@
include @builddir@/Makefile.meta
```

c++/src/corelib/Makefile.in:

```
LIB_PROJ = corelib
SUB_PROJ = test
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

c++/src/corelib/Makefile.corelib.lib:

```
SRC = ncbidiag ncbiexpt ncbistre ncbiapp ncbireg ncbienv ncbistd
LIB = xncbi
```

c++/src/corelib/test/Makefile.in:

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

Managing the Work Environment

The following topics are discussed in this section:

Obtaining the Very Latest Builds

Each new nightly build is available in the $NCBI/c++.by-date/{date} subdirectory. This is done regardless of whether the build succeeds or not.

There are defined symlinks into this directory tree. They include:

- $NCBI/c++ - Symbolic link to $NCBI/c++.production.

- $NCBI/c++.potluck - The most recent nightly build. It contains whatever libraries and executables have managed to build, and it can miss some of the libraries and/or executables. Use it if you desperately need yesterday's bug fix and do not care of the libraries which are missing.
- $NCBI/c++.metastable - The most recent nightly build for which the compilation (but not necessarily the test suite) succeeded in all configurations on the given platform. Please note that some projects, including the entire "gui" tree, are considered expendable due to their relative instability and therefore not guaranteed to be present.
- $NCBI/c++.current - Symbolic link to $NCBI/c++.metastable.
- $NCBI/c++.stable - The most recent nightly build for which the nightly build (INCLUDING the gui projects) succeeded AND the test suite passed all critical tests on the given platform. This would be the preferred build most of the time for the developers whose projects make use of the actively developed C++ Toolkit libraries. It is usually relatively recent (usually no more than 1 or 2 weeks behind), and at the same time quite stable.
- $NCBI/c++.frozen - A "production candidate" build made out of the production codebase. There are usually two such builds made for each version of production codebase -- one is for the original production build, and another (usually made in about 2 months after the original production build) is the follow-up bugfix build.
- $NCBI/c++.production - The most recent production snapshot. This is determined based on general stability of the Toolkit and it is usually derived off the codebase of one of the prior "c++.stable" builds. Its codebase is the same for all platforms and configurations. It is installed only on the major NCBI development platforms (Linux, MS-Windows, and MacOS). It is the safest bet for long-term development. It changes rarely, once in 1 to 3 months. Also, unlike all other builds mentioned here it is guaranteed to be accessible for at least a year (or more), and its DLLs are installed on all (including production) Linux hosts.
- $NCBI/c++.prod-head - This build is for NCBI developers to quickly check their planned stable component commits using import_project. It is based on the repository path toolkit/production/candidates/production.HEAD – which is the HEAD SVN revision of the C++ Stable Components on which the latest c++.production build was based. It is available on 64-bit Linux.
- $NCBI/c++.trial - This build is for NCBI developers to quickly check their planned stable component commits using import_project. It is based on the repository path toolkit/production/candidates/trial – which is usually a codebase for the upcoming production build. It is available on 64-bit Linux.

Working in a separate directory

The following topics are discussed in this section:

- Setting up Directory Location
- The Project's Makefile
- Testing your setup

Setting up Directory Location

There are two topics relevant to writing an application using the NCBI C++ Toolkit:

- Where to place the source and header files for the project
- How to create a makefile which can link to the correct C++ libraries

What you put in your makefile will depend on where you define your working directory. In this discussion, we assume you will be working **outside** the NCBI C++ tree, say in a directory called newproj. This is where you will write both your source and header files. The first step then, is to create the new working directory and use the new_project script to install a makefile there:

```
mkdir newproj
new_project newproj app $NCBI/c++/GCC-Debug/build
 Created a model makefile "/home/user/newproj/Makefile.newproj_app".
```

The syntax of the script command is:

```
new_project <project_name> <app | lib> [builddir]
```

where: - project_name is the name of the directory you will be working in - app (lib) is used to indicate whether you will be building an application or a library - builddir (optional) specifies what version of the pre-built NCBI C++ Toolkit libraries to link to

Several build environments have been pre-configured and are available for developing on various platforms using different compilers, in either **debug** or **release** mode. These environments include custom-made configuration files, makefile templates, and links to the appropriate pre-built C++ Toolkit libraries. To see a list of the available environments for the platform you are working on, use: ls -d $NCBI/c++/*/build. For example, on Solaris, the build directories currently available are shown in Table 1.

In the example above, we specified the GNU compiler debug environment: $NCBI/c++/GCC-Debug/build. For a list of currently supported compilers, see the release notes. Running the new_project script will generate a ready-to-use makefile in the directory you just created. For a more detailed description of this and other scripts to assist you in the set-up of your working environment, see Starting a new C++ project.

The Project's Makefile

The file you just created with the above script will be called Makefile.newproj_app. In addition to other things, you will see definitions for: - $(builddir) - a path to the build directory specified in the last argument to the above script - $(srcdir) - the path to your current working directory (".") - $(APP) - the application name - $(OBJ) - the names of the object modules to build and link to the application - $(LIB) - specific libraries to link to in the NCBI C++ Toolkit - $(LIBS) - all other libraries to link to (outside the C++ Toolkit)

$(builddir)/lib specifies the library path (-L), which in this case points to the GNU debug versions of the NCBI C++ Toolkit libraries. $(LIB) lists the individual libraries in this path that you will be linking to. Minimally, this should include xncbi - the library which implements the foundational classes for the C++ tools. Additional library names (e.g. xhtml, xcgi, etc.) can be added here.

Since the shell script assumes you will be building a single executable with the same name as your working directory, the application is defined simply as newproj. Additional targets to build can be added in the area indicated towards the end of the file. The list of objects (OBJ) should include the names (without extensions) of all source files for the application (APP). Again, the script makes the simplest assumption, i.e. that there is a single source file named newproj.cpp. Additional source names can be added here.

Testing your setup

For a very simple application, this makefile is ready to be run. Try it out now, by creating the file newproj.cpp:

```
// File name: newproj.cpp
#include <iostream>
using namespace std;
int main() {
cout << "Hello again, world" << endl;
}
```

and running:

```
make -f Makefile.newproj_app
```

Of course, it wasn't necessary to set up the directories and makefiles to accomplish this much, as this example does not use any of the C++ classes or resources defined in the NCBI C++ Toolkit. But having accomplished this, you are now prepared to write an actual application, such as described in Writing a simple application project

Most real applications will at a minimum, require that you #include ncbistd.hpp in your header file. In addition to defining some basic NCBI C++ Toolkit objects and templates, this header file in turn includes other header files that define the C Toolkit data types, NCBI namespaces, debugging macros, and exception classes. A set of template files are also provided for your use in developing new applications.

Working Independently In a C++ Subtree

An alternative to developing a new project from scratch is to work within a subtree of the main NCBI C++ source tree so as to utilize the header, source, and make files defined for that subtree. One way to do this would be to check out the entire source tree and then do all your work within the selected subtree(s) only. A better solution is to create a new working directory and check out only the relevant subtrees into that directory. This is somewhat complicated by the distributed organization of the C++ SVN tree: header files are (recursively) contained in an include subtree, while source files are (recursively) contained in a src subtree. Thus, multiple checkouts may be required to set things up properly, and the customized makefiles (Makefile.*.app) will need to be modified. The shell script import_project will do all of this for you. The syntax is:

```
import_project subtree_name [builddir]
```

where:

- subtree_name is the path to a selected directory inside [internal/]c++/src/
- builddir (optional) specifies what version of the pre-built NCBI C++ Toolkit libraries to link to.

As a result of executing this shell script, you will have a new directory created with the pathname ./[internal/]c++/ whose structure contains "slices" of the original SVN tree. Specifically, you will find:

```
./[internal/]c++/include/subtree_name
./[internal/]c++/src/subtree_name
```

The src and include directories will contain all of the requested subtree's source and header files along with any hierarchically defined subdirectories. In addition, the script will create new makefiles with the suffix *_*app*. These makefiles are generated from the original customized makefiles (Makefile.*.app) located in the original src subtrees. The customized makefiles were designed to work only in conjunction with the build directories in the larger NCBI C++ tree; the newly created makefiles can be used directly in your new working directories.

You can re-run import_project to add multiple projects to your tree.

Note: If you'd like to import both internal and public projects into a single tree, you'll need to use the -topdir option, which will locate the public project within the internal tree, for example:

```
import_project internal/demo/misc/xmlwrapp
import_project -topdir trunk/internal/c++ misc/xmlwrapp
pushd trunk/internal/c++/src/misc/xmlwrapp
make
popd
pushd trunk/internal/c++/src/internal/demo/misc/xmlwrapp
make
```

In this case, your public projects will be located in the internal tree. You must build in each imported subtree, in order from most-dependent to least-dependent so that the imported libraries will be linked to rather than the pre-built libraries.

The NCBI C++ Toolkit project directories, along with the libraries they implement and the logical modules they entail, are summarized in the Library Reference.

Two project directories, internal and objects, may have some subdirectories for which the import_project script does not work normally, if at all. The internal subdirectories are used for in-house development, and the author of a given project may customize the project for their own needs in a way that is incompatible with import_project. The objects subdirectories are used as the original repositories for ASN.1 specifications (which are available for use in your application as described in the section Processing ASN.1 Data), and subsequently, for writing the object definitions and implementations created by the datatool program. Again, these projects can be altered in special ways and some may not be compatible with import_project. Generally, however, import_project should work well with most of these projects.

Working within the C++ source tree

The following topics are discussed in this section:

- Checkout the source tree and configure a build directory
- The project's directories and makefiles
- Makefile.in meta files
- An example meta-makefile and its associated project makefiles
- Executing make
- Custom project makefile: Makefile.myProj
- Library project makefile: Makefile.myProj.lib
- Application project makefile: Makefile.myProj.app
- Defining and running tests

- The configure scripts

Most users will find that working in a checked-out subtree or a private directory is preferable to working directly in the C++ source tree. There are two good reasons to avoid doing so:

- Building your own versions of the extensive libraries can be very time-consuming.
- There is no guarantee that the library utilities your private code links to have not become obsolete.

This section is provided for those developers who must work within the source tree. The Library Reference provides more complete and technical discussion of the topics reviewed here, and many links to the relevant sections are provided. This page is provided as an overview of material presented in the Library Reference and on the Working with Makefiles pages.

Checkout (*) the source tree and configure a build directory

To checkout full Toolkit tree:

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

or, if you don't need internal projects:

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

Once you have done so, you will need to run one of the configure scripts in the Toolkit's root directory. For example, to configure your environment to work with the gcc compiler (on any platform), just run: ./configure.

Users working under Windows should consult the MS Visual C++ section in the chapter on Configuring and Building the Toolkit.

The configure script is a multi-platform configuration shell script (generated from configure.in using autoconf). Here are some pointers to sections that will help you configure the build environment:

- Wrapper scripts supporting various platforms
- Optional configuration flags

The configure script concludes with a message describing how to build the C++ Toolkit libraries. If your application will be working with ASN.1 data, use the --with-objects flag in running the configure script, so as to populate the include/objects and src/objects subdirectories and build the objects libraries. The objects directories and libraries can also be updated separately from the rest of the compilation, by executing make inside the build/objects directory. Prior to doing so however, you should always verify that your build/bin directory contains the latest version of datatool.

The project's directories and makefiles

To start a new project ("myProj"), you should begin by creating both a src and an include subtree for that project inside the C++ tree. In general, all header files that will be accessed by multiple source modules outside the project directory should be placed in the include directory. Header files that will be used solely inside the project's src directory should be placed in the src directory, along with the implementation files.

In addition to the C++ source files, the src subtrees contain meta-makefiles named Makefile.in, which are used by the configure script to generate the corresponding makefiles in the build subtrees. Figure 1 shows slices of the directory structure reflecting the correspondences between the meta-makefiles in the src subtrees and makefiles in the build subtrees. Figure 2 is a sketch of the entire C++ tree in which these directories are defined.

During the configuration process, each of the meta-makefiles in the top-level of the src tree is translated into a corresponding makefile in the top-level of the build tree. Then, for each project directory containing a Makefile.in, the configure script will: (1) create a corresponding subdirectory of the same name in the build tree if it does not already exist, and (2) generate a corresponding makefile in the project's build subdirectory. The contents of the project's Makefile.in in the src subdirectory determine what is written to the project's makefile in the build subdirectory. Project subdirectories that do not contain a Makefile.in file are ignored by the configure script.

Thus, you will also need to create a meta-makefile in the newly created src/myProj directory before configuring your build directory to include the new project. The configure script will then create the corresponding subtree in the build directory, along with a new makefile generated from the Makefile.in you created. See Makefile Hierarchy (Chapter 4, Figure 1) and Figure 1.

Makefile.in meta files

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

- USR_PROJ (optional) - a list of names for user-defined makefiles.
  This macro is provided for the usage of ordinary stand-alone makefiles which do not utilize the make commands contained in additional makefiles in the top-level build directory. Each p_i listed in USR_PROJ = p_1 ... p_N must have a corresponding Makefile.p_i in the project's source directory. When make is executed, the make directives contained in these files will be executed directly to build the targets as specified.
- LIB_PROJ (optional) - a list of names for library makefiles.
  For each library l_i listed in LIB_PROJ = l_1 ... l_N, you must have created a corresponding project makefile named Makefile.l_i.lib in the project's source directory. When make is executed, these library project makefiles will be used along with Makefile.lib and Makefile.lib.tmpl (located in the top-level of the build tree) to build the specified libraries.
- APP_PROJ (optional) - a list of names for application makefiles.
  Similarly, each application (p1, p2, ..., pN) listed under APP_PROJ must have a corresponding project makefile named Makefile.p*.app in the project's source directory. When make is executed, these application project makefiles will be used along with Makefile.app and Makefile.app.tmpl to build the specified executables.
- SUB_PROJ (optional) - a list of names for subproject directories (used on recursive makes).
  The SUB_PROJ macro is used to recursively define make targets; items listed here define the subdirectories rooted in the project's source directory where make should also be executed.

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

An example meta-makefile and its associated project makefiles

A simple example should help to make this more concrete. Assuming that myProj is used to develop an application named myProj, myProj/Makefile.in should contain the following:

```
####### Example: src/myProj/Makefile.in
APP_PROJ = myProj
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

The last two lines in Makefile.in should always be exactly as shown here. These two lines specify make variable templates using the @var_name@ syntax. When generating the corresponding makefile in the build directory, the configure script will substitute each identifier name bearing that notation with full path definitions.

The corresponding makefile in build/myProj generated by the configure script for this example will then contain:

```
####### Example: myBuild/build/myProj/Makefile
# Generated automatically from Makefile.in by configure.
APP_PROJ = myProj
srcdir = /home/zimmerma/internal/c++/src/myProj
include /home/zimmerma/internal/c++/myBuild/build/Makefile.meta
```

As demonstrated in this example, the @srcdir@ and @builddir@ aliases in the makefile template have been replaced with absolute paths in the generated makefile, while the definition of APP_PROJ is copied verbatim.

The only build target in this example is myProj. myProj is specified as an application - not a library - because it is listed under APP_PROJ rather than under LIB_PROJ. Accordingly, there must also be a file named Makefile.myProj.app in the src/myProj directory. A project's application makefile specifies:

- APP - the name to be used for the resulting executable
- OBJ - a list of object files to use in the compilation
- LIB - a list of NCBI C++ Toolkit libraries to use in the linking
- LIBS - a list of other libraries to use in the linking

There may be any number of application or library makefiles for the project, Each application should be listed under APP_PROJ and each library should be listed under LIB_PROJ in Makefile.in. A suitable application makefile for this simple example might contain just the following text:

```
####### Example: src/myProj/Makefile.myProj.app
APP = myProj
OBJ = myProj
LIB = xncbi
```

In this simple example, the APP_PROJ definition in Makefile.in is identical to the definitions of both APP and OBJ in Makefile.myProj.app. This is not always the case, however, as the APP_PROJ macro is used to define which makefiles in the src directory should be used during compilation, while APP defines the name of the resulting executable and OBJ specifies the names of object files. (Project makefiles for applications are described in more detail below.)

Executing make

Given these makefile definitions, executing make all_r in the build project subdirectory indirectly causes build/Makefile.meta to be executed, which sets the following chain of events in motion:

1. For each proj_name listed in USR_PROJ, Makefile.meta first tests to see if Makefile.proj_name is available in the current build directory, and if so, executes:

   ```
   make -f Makefile.proj_name builddir="$(builddir)"
   srcdir="$(srcdir)" $(MFLAGS)
   ```

   Otherwise, Makefile.meta assumes the required makefile is in the project's source directory, and executes:

   ```
   make -f $(srcdir)/Makefile.proj_name builddir="$(builddir)" srcdir="$(srcdir)" $
   (MFLAGS)
   ```

   In either case, the important thing to note here is that the commands contained in the project's makefiles are executed directly and are **not** combined with additional makefiles in the top-level build directory. The aliased srcdir, builddir, and MFLAGS are still available and can be referred to inside Makefile.proj_name. By default, the resulting libraries and executables are written to the build directory only.
2. For each lib_name listed in LIB_PROJ,

   ```
   make -f $(builddir)/Makefile.lib.tmpl
   ```

   is executed. This in turn specifies that $(builddir)/Makefile.mk, $(srcdir)/ Makefile.lib_name.lib, and $(builddir)/Makefile.lib should be included in the generated makefile commands that actually get executed. The resulting libraries are written to the build subdirectory and copied to the lib subtree.
3. For each app_name listed in APP_PROJ,

   ```
   make -f $(builddir)/Makefile.app.tmpl
   ```

   is executed. This in turn specifies that $(builddir)/Makefile.mk, $(srcdir)/ Makefile.app_name.app, and $(builddir)/Makefile.app should be included in the generated makefile commands that actually get executed. The resulting executables are written to the build subdirectory and copied to the bin subtree.
4. For each dir_name listed in SUB_PROJ (on make all_r),

   ```
   cd dir_name
   make all_r
   ```

   is executed. Steps (1) - (3) are then repeated in the project subdirectory.

More generally, for each subdirectory listed in SUB_PROJ, the configure script will create a relative subdirectory inside the new build project directory, and generate the new subdirectory's Makefile from the corresponding meta-makefile in the src subtree. Note that each subproject directory must also contain its own Makefile.in along with the corresponding project makefiles. The recursive make commands, make all_r, make clean_r, and make purge_r all refer to this definition of the subprojects to define what targets should be recursively built or removed.

Custom project makefile: Makefile.myProj (*)

As described, regular makefiles contained in the project's src directory will be invoked from the build directory if their suffixes are specified in the USR_PROJ macro. This macro is originally defined in the project's src directory in the Makefile.in meta file, and is propagated to the corresponding Makefile in the build directory by the configure script.

For example, if USR_PROJ = myProj in the build directory's Makefile, executing make will cause Makefile.myProj (the project makefile) to be executed. This project makefile may be located in either the current build directory **or** the corresponding src directory. In either case, although the makefile is executed directly, references to the source or object files (contained in the project makefile) must give complete paths to those files. In the first case, make is invoked as: make -f Makefile.myProj, so the makefile is located in the current working (build) directory but the source files are not. In the second case, make is invoked as:

make -f $(srcdir)/Makefile.myProj,

so both the project makefile **and** the source files are non-local. For example:

```
####### Makefile.myProj
include $(NCBI)/ncbi.mk
# use the NCBI default compiler for this platform
CC = $(NCBI_CC)
# along with the default include
INCPATH = $(NCBI_INCDIR)
# and library paths
LIBPATH = $(NCBI_LIBDIR)
all: $(srcdir)/myProj.c
 $(CC) -o myProj $(srcdir)/myProj.c $(NCBI_CFLAGS) -I($INCPATH) \
 -L($LIBPATH) -lncbi
 cp -p myProj $(builddir)/bin
clean:
 -rm myProj myProj.o
purge: clean
 -rm $(builddir)/bin/myProj
```

will cause the C program myProj to be built directly from Makefile.myProj using the default C compiler, library paths, include paths, and compilation flags defined in ncbi.mk. The executables and libraries generated from the targets specified in USR_PROJ are by default written to the current build directory only. In this example however, they are also explicitly copied to the bin directory, and accordingly, the purge directives also remove the copied executable.

Library project makefile: Makefile.myProj.lib (*)

Makefile.lib_name.lib should contain the following macro definitions:

- $(SRC) - the names of all source files to compile and include in the library
- $(OBJ) - the names of any pre-compiled object files to include in the library
- $(LIB) - the name of the library being built

In addition, any of the make variables defined in build/Makefile.mk, such as $CPPFLAGS, $LINK, etc., can be referred to and/or redefined in the project makefile, e.g.:

```
CFLAGS = $(ORIG_CFLAGS) -abc -DFOOBAR_NOT_CPLUSPLUS
CXXFLAGS = $(ORIG_CXXFLAGS) -xyz
CPPFLAGS = $(ORIG_CPPFLAGS) -UFOO -DP1_PROJECT -I$(NCBI_C_INCLUDE)
LINK = purify $(ORIG_LINK)
```

For an example from the Toolkit, see Makefile.corelib.lib, and for a documented example, see example 1 above. This customized makefile can be used to build both static and dynamic (DLL) versions of the library. To build as a DLL on the appropriate platforms, you can explicitly specify:

```
LIB_OR_DLL = dll
```

Conversely, if you want the library to always be built as static, specify:

```
LIB_OR_DLL = lib
```

Application project makefile: Makefile.myProj.app (*)

Makefile.app_name.app should contain the following macro definitions:

- $(SRC) - the names of the object modules to build and link to the application
- $(OBJ) - the names of any pre-compiled object files to include in the linking
- $(LIB) - specific libraries in the NCBI C++ Toolkit to include in the linking
- $(LIBS) - all other libraries to link to (outside the C++ Toolkit)
- $(APP) - the name of the application being built

For example, if C Toolkit libraries should also be included in the linking, use:

```
LIBS = $(NCBI_C_LIBPATH) -lncbi $(ORIG_LIBS)
```

The project's application makefile can also redefine the compiler and linker, along with other flags and tools affecting the build process, as described above for Makefile.*.lib files. For an example from the Toolkit, see Makefile.coretest.app, and for a documented example, see example 2 above.

Defining and running tests

The definition and execution of unit tests is controlled by the CHECK_CMD macro in the test application's makefile, Makefile.app_name.app. If this macro is not defined (or commented out), then no test will be executed. If CHECK_CMD is defined, then the test it specifies will be included in the automated test suite and can also be invoked independently by running "make check".

To include an application into the test suite it is necessary to add just one line into its makefile Makefile.app_name.app:

```
CHECK_CMD =
```

or

```
CHECK_CMD = command line to run application test
```

For the first form, where no command line is specified by the CHECK_CMD macro, the program specified by the makefile variable APP will be executed (without any parameters).

For the second form: If your application is executed by a script specified in a CHECK_CMD command line, and it doesn't read from STDIN, then the script should invoke it like this:

```
$CHECK_EXEC app_name arg1 arg2 ...
```

If your application *does* read from STDIN, then CHECK_CMD scripts should invoke it like this:

```
$CHECK_EXEC_STDIN app_name arg1 arg2 ...
```

Note: Applications / scripts in the CHECK_CMD definition should **not** use ".", for example:

```
$CHECK_EXEC ./app_name arg1 arg2 ... # Do not prefix app_name with ./
```

Scripts invoked via CHECK_CMD should pass an exit code to the testing framework via the exitcode variable, for example:

```
exitcode=$?
```

If your test program needs additional files (for example, a configuration file, data files, or helper scripts referenced in CHECK_CMD), then set CHECK_COPY to point to them:

```
CHECK_COPY = file1 file2 dir1 dir2
```

Before the tests are run, all specified files and directories will be copied to the build or special check directory (which is platform-dependent). Note that all paths to copied files and directories must be relative to the application source directory.

By default, the application's execution time is limited to 200 seconds. You can set a new limit using:

```
CHECK_TIMEOUT = <time in seconds>
```

If application continues execution after specified time, it will be terminated and test marked as FAILED.

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

For information about using Boost for unit testing, see the "Boost Unit Test Framework" chapter.

The configure scripts

A number of compiler-specific wrappers for different platforms are described in the chapter on configuring and building. Each of these wrappers performs some pre-initialization for the tools and flags used in the configure script before running it. The compiler-specific wrappers are in the c++/compilers directory. The configure script serves two very different types of function: (1) it tests the selected compiler and environment for a multitude of features and generates #include and #define statements accordingly, and (2) it reads the Makefile.in files in the src directories and creates the corresponding build subtrees and makefiles accordingly.

Frequently during development it is necessary to make minor adjustments to the Makefile.in files, such as adding new projects or subprojects to the list of targets. In these contexts however, the compiler, environment, and source directory structures remain unchanged, and configure is actually doing much more work than is necessary. In fact, there is even some risk of failing to re-create the same configuration environment if the user does not exactly duplicate the same set of configure flags when re-running configure. In these situations, it is preferable to run an auxiliary script named config.status, located at the top level of the build directory in a subdirectory named status.

In contrast, changes to the src directory structure, or the addition/deletion of Makefile.in files, all require re-running the configure script, as these actions require the creation/deletion of subdirectories in the build tree and/or the creation/deletion of the associated Makefile in those directories.

Working with the serializable object classes

The following topics are discussed in this section:

- Serializable Objects
- Locating and browsing serializable objects in the C++ Toolkit
- Base classes and user classes
- Adding methods to the user classes
  - Checking out source code, configuring the working environment, building the libraries
  - Adding methods

Serializable Objects

All of the ASN.1 data types defined in the C Toolkit have been re-implemented in the C++ Toolkit as serializable objects. Header files for these classes can be found in the include/objects directories, and their implementations are located in the src/objects directories. and

The implementation of these classes as serializable objects has a number of implications. It must be possible to use expressions like: instream >> myObject and outstream << myObject, where specializations are entailed for the serial format of the iostreams (ASN.1, XML, etc.), as well as for the internal structure of the object. The C++ Toolkit deploys several object stream classes that specialize in various formats, and which know how to access and apply the type information that is associated with the serializable object.

The type information for each class is defined in a separate static CTypeInfo object, which can be accessed by all instances of that class. This is a very powerful device, which allows for the implementation of many features generally found only in languages which have built-in class reflection. Using the Toolkit's serializable objects will require some familiarity with the usage of this type information, and several sections of this manual cover these topics (see Runtime Object Type Information for a general discussion).

Locating and browsing serializable objects in the C++ Toolkit

The top level of the include/objects subtree is a set of subdirectories, where each subdirectory includes the public header files for a separately compiled library. Similarly, the src/objects subtree includes a set of subtrees containing the source files for these libraries. Finally, your build/objects directory will contain a corresponding set of build subtrees where these libraries are actually built.

If you checked out the entire C++ SVN tree, you may be surprised to find that initially, the include/objects subtrees are empty, and the subdirectories in the src/objects subtree contain only ASN.1 modules. This is because both the header files and source files are auto-generated from the ASN.1 specifications by the datatool program. As described in Working within the C++ source tree, you can build everything by running make all_r in the build directory.

Note: If you would like to have the objects libraries built locally, you **must** use the --with-objects flag when running the configure script.

You can also access the pre-generated serializable objects in the public area, using the source browsers to locate the objects you are particularly interested in. For example, if you are seeking the new class definition for the Bioseq struct defined in the C Toolkit, you can search for the CBioseq class, using either the LXR identifier search tool, or the Doxygen class hierarchy browser. Starting with the name of the data object as it appears in the ASN.1 module, two simple rules apply in deriving the new C++ class name:

- The one letter 'C' (for class) prefix should precede the ASN.1 name
- All hyphens ('-') should be replaced by underscores ('_')

For example, Seq-descr becomes CSeq_descr.

Base classes and user classes

The classes whose names are derived in this manner are called the user classes, and each also has a corresponding base class implementation. The name of the base class is arrived at by appending "_Base" to the user class name. Most of the user classes are empty wrapper classes that do not bring any new functionality or data members to the inherited base class; they are simply provided as a platform for development. In contrast, the base classes are **not** intended for public use (other than browsing), and should never be modified.

More generally, the base classes should *never* be instantiated or accessed directly in an application. The relation between the two source files and the classes they define reflects a general design used in developing the object libraries: the base class files are auto-generated by datatool according to the ASN.1 specifications in the src/objects directories; the inherited class files (the so-called user classes) are intended for developers who can extend these classes to support features above and beyond the ASN.1 specifications.

Many applications will involve a "tangled hierarchy" of these objects, reflecting the complexity of the real world data that they represent. For example, a CBioseq_set contains a list of CSeq_entry objects, where each CSeq_entry is, in turn, a choice between a CBioseq and a CBioseq_set.

Given the potential for this complexity of interactions, a critical design issue becomes how one can ensure that methods which may have been defined only in the user class will be available for all instances of that class. In particular, these instances may occur as contained elements of another object which is compiled in a different library. These inter-object dependencies are the motivation for the user classes. As shown in Figure 2, all references to external objects which occur inside the base classes, access external user classes, so as to include any methods which may be defined only in the user classes:

In most cases, adding non-virtual methods to a user class will **not** require re-compiling any libraries except the one which defines the modified object. Note however, that adding non-static data members and/or virtual methods to the user classes **will change** the class layouts, and in these cases only, will entail recompiling any external library objects which access these classes.

Adding methods to the user classes

Note: This section describes the steps currently required to add new methods to the user classes. It is subject to change, and there is no guarantee the material here is up-to-date. In general, it is not recommended practice to add methods to the user classes, unless your purpose is to extend these classes across all applications as part of a development effort.

The following topics are discussed in this section:

- Checking out source code, configuring the working environment, building the libraries.
- Adding methods

Checking out source code, configuring the working environment, building the libraries

- Create a working directory (e.g. Work) and check out the C++ tree to that directory:, using either SVN checkout or the shell script, svn_core.
- Configure the environment to work inside this tree using one of the configure scripts, according to the platform you will be working on. Be sure to include the --with-objects flag in invoking the configure script.
- Build the xncbi, xser and xser libraries, and run datatool to create the objects header and source files, and build all of the object module libraries:

```
# Build the core library
cd path_to_compile_dir/build/corelib
make
# Build the util library
cd path_to_compile_dir/build/util
make
# might as well build datatool and avoid possible version skew cd
path_to_compile_dir/build/serial make all_r
# needed for a few projects
cd path_to_compile_dir/build/connect
make
cd path_to_compile_dir/build/objects
make all_r
```

Here path_to_compile_dir is set to the compile work directory which depends on the compiler settings (e.g: ~/Work/internal/GCC-Debug). In addition to creating the header and source files, using make all_r (instead of just make) will build all the libraries. All libraries that are built are also copied to the lib dir, e.g.:~/Work/internal/c++/GCC-Debug/lib. Similarly, all executables (such as asn2asn) are copied to the bin dir, e.g.: ~/Work/internal/c++/GCC-Debug/bin.

You are now ready to edit the user class files and add methods.

Adding methods

As an example, suppose that we would like to add a method to the CSeq_inst class to calculate sequence length, e.g.:CSeq_inst::CalculateLength(). We begin by adding a declaration of this method to the public section of the user class definition in Seq_inst.hpp:

```
class CSeq_inst : public CSeq_inst_Base
{
public:
 CSeq_inst(void);
 ~CSeq_inst(void);
 static CSeq_inst* New(void)
 {
 return new CSeq_inst(eCanDelete);
 }
 int CalculateLength() const;
protected:
 CSeq_inst(ECanDelete);
};
```

and in the source file, Seq_inst.cpp, we implement

```
int CSeq_inst::CalculateLength() const
{
 // implementation goes here
}
```

These files are in the include/objects/seq and src/objects/seq subdirectories, respectively. Once you have made the modifications to the files, you need to recompile the seq library, libseq.a, i.e.:

```
cd path_to_compile_dir/GCC-Debug/build/objects/seq
make
```

Here path_to_compile_dir is set to the compile work directory which depends on the compiler settings (e.g: ~/Work/internal/GCC-Debug). The new method can now be invoked from within a CBioseq object as: myBioseq.GetInst().CalculateLength().

The key issue that determines whether or not you will need to rebuild any external libraries that use the modified user class involves the class layout in memory. All of the external libraries which reference the object refer to the class layout that existed prior to the changes you have made. Thus, if your modifications do **not** affect the class layout, you do not have to rebuild any external libraries. Changes that *do* affect memory mapping include:

- The addition of new, non-static data members
- The addition of virtual methods

If you have added either of the above to the user class, then you will need to identify all external objects which use your object, and recompile the libraries in which these objects are defined.

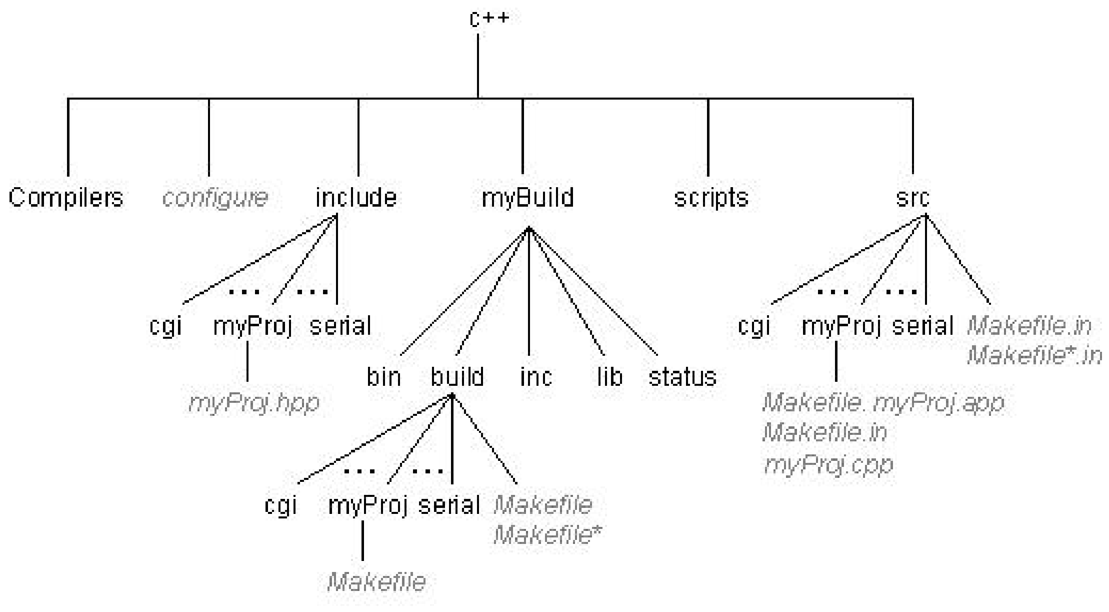

Figure 2

Figure 1. Meta makefiles and the makefiles they generate

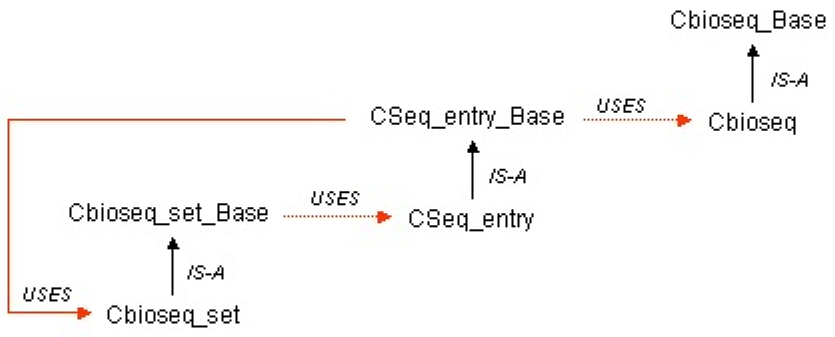

Figure 2. Example of complex relationships between base classes and user classes

Table 1. Build Directories

Directory	Compiler	Version
/netopt/ncbi_tools/c++/Debug/build	Sun Workshop	Debug
/netopt/ncbi_tools/c++/Debug64/build	Sun Workshop	Debug (64 bit)
/netopt/ncbi_tools/c++/DebugMT/build	Sun Workshop	Debug (Multi-thread safe)
/netopt/ncbi_tools/c++/Release/build	Sun Workshop	Release
/netopt/ncbi_tools/c++/ReleaseMT/build	Sun Workshop	Release (Multi-thread safe)
/netopt/ncbi_tools/c++/GCC-Debug/build	GCC	Debug
/netopt/ncbi_tools/c++/GCC-Release/build	GCC	Release