7: Programming Policies and Guidelines

Last Update: July 8, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discusses policies and guidelines for the development of NCBI software.

Chapter Outline

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

Choice of Language

C++ is typically the language of choice for C++ Toolkit libraries and applications. The policy for language choice in other areas within NCBI is:

- **C/C++** -- for high-performance standalone backend servers and CGIs, computationally intensive algorithms and large data flow processing tools used in production.
- **sh** or **bash** -- for primitive scripting.
- **Python** -- for advanced scripting. See its usage policy here.
- **Perl** -- for advanced scripting. The Python usage policy can be applied to Perl as well.
- **Java** -- for Eclipse programming and in-house QA and testing tools.

See the "Recommended programming and scripting languages" Wiki page for more information and updates to this policy. Send proposals for corrections, additions and extensions of the policy on language choice to the languages mailing list, languages@ncbi.nlm.nih.gov.

Source Code Conventions

This section contains C++ style guidelines, although many of these guidelines could also apply, at least in principle, to other languages. Adherence to these guidelines will promote uniform coding, better documentation, easy to read code, and therefore more maintainable code.

The following topics are discussed in this section:

- Public Domain Notice
- Naming Conventions
    - Type Names
    - Preprocessor Define/Macro
    - Function Arguments and Local Variables
    - Constants
    - Class and Structure Data Members (Fields)
    - Class Member Functions (Methods)
    - Module Static Functions and Data
    - Global ("extern") Functions and Data
- Name Prefixing and/or the Use of Namespaces
- Use of the NCBI Name Scope

Public Domain Notice

All NCBI-authored C/C++ source files **must** begin with a comment containing NCBI's public domain notice, shown below. Ideally (subject to the developer's discretion), so should any other publicly released source code and data (including scripting languages and data specifications).

```
/* $Id$
 *
===========================================================================
 *
 * PUBLIC DOMAIN NOTICE
 * National Center for Biotechnology Information
 *
 * This software/database is a "United States Government Work" under the
 * terms of the United States Copyright Act. It was written as part of
 * the author's official duties as a United States Government employee and
 * thus cannot be copyrighted. This software/database is freely available
 * to the public for use. The National Library of Medicine and the U.S.
 * Government have not placed any restriction on its use or reproduction.
 *
 * Although all reasonable efforts have been taken to ensure the accuracy
 * and reliability of the software and data, the NLM and the U.S.
 * Government do not and cannot warrant the performance or results that
 * may be obtained by using this software or data. The NLM and the U.S.
 * Government disclaim all warranties, express or implied, including
 * warranties of performance, merchantability or fitness for any particular
 * purpose.
 *
 * Please cite the author in any work or product based on this material.
 *
 *
===========================================================================
 */
```

If you have questions, please email to cpp-core@ncbi.nlm.nih.gov.

Naming Conventions

Table 1. Naming Conventions

SYNOPSIS	EXAMPLE
Type Names	

***C**ClassTypeName*	class CMyClass { ..... };
***I**InterfaceName*	class IMyInterface { ..... };
***S**StructTypeName*	struct SMyStruct { ..... };
***U**UnionTypeName*	union UMyUnion { ..... };
***E**EnumTypeName*	enum EMyEnum { ..... };
***F**FunctionTypeName*	typedef int (*FMyFunc)(void);
***P**PredicateName*	struct PMyPred { bool operator() (.... , ...); };
***T**AuxiliaryTypedef* $^{(*)}$	typedef map<int,string> TMyMapIntStr;
T**Iterator_**I	typedef list<int>::iterator TMyList_I;
T**ConstIterator_**CI	typedef set<string>::const_iterator TMySet_CI;
***N**Namespace* (see also)	namespace NMyNamespace { ..... }
Preprocessor Define/Macro	
MACRO_NAME	#define MY_DEFINE 12345
macro_arg_name	#define MY_MACRO(x, y) (((x) + 1) < (y))
Function Arguments and Local Variables	

func_local_var_name	void MyFunc(int foo, const CMyClass& a_class)
	{
	   size_t foo_size;
	   int bar;

Constants	
***k**ConstantName*	const int kMyConst = 123;

***e**EnumValueName*	enum EMyEnum {
	   eMyEnum_1 = 11,
	   eMyEnum_2 = 22,
	   eMyEnum_3 = 33
	};

***f**FlagValueName*	enum EMyFlags {
	   fMyFlag_1 = (1<<0), ///< = 0x1 (describe)
	   fMyFlag_2 = (1<<1), ///< = 0x2 (describe)
	   fMyFlag_3 = (1<<2) ///< = 0x4 (describe)
	};
	typedef int TMyFlags; ///< holds bitwise OR of "EMyFlags"

Class and Structure Data Members (Fields)	
***m_**ClassMemberName*	class C { short int m_MyClassData; };
struct_field_name	struct S { int my_struct_field; };
***sm_**ClassStaticMemberName*	class C { static double sm_MyClassStaticData; };
Class Member Functions (Methods)	
ClassMethod	bool MyClassMethod(void);
***x_**ClassPrivateMethod*	int x_MyClassPrivateMethod(char c);
Module Static Functions and Data	
***s_**StaticFunc*	static char s_MyStaticFunc(void);

***s**_StaticVar*	static int s_MyStaticVar;
Global (*"extern"*) Functions and Data	
***g**_GlobalFunc*	double g_MyGlobalFunc(void);
***g**_GlobalVar*	short g_MyGlobalVar;

(*) The auxiliary typedefs (like ***T**AuxiliaryTypedef*) are usually used for an ad-hoc type mappings (especially when using templates) and not when a real type definition takes place.

Name Prefixing and/or the Use of Namespaces

In addition to the above naming conventions that highlight the nature and/or the scope of things, one should also use prefixes to:

- avoid name conflicts
- indicate the package that the entity belongs to

For example, if you are creating a new class called "Bar" in package "Foo" then it is good practice to name it "CFooBar" rather than just "CBar". Similarly, you should name new constants like "kFooSomeconst", new types like "TFooSometype", etc.

Use of the NCBI Name Scope

<ncbistl.hpp>

All NCBI-made "core" API code must be put into the "ncbi::" namespace. For this purpose, there are two preprocessor macros, BEGIN_NCBI_SCOPE and END_NCBI_SCOPE, that must enclose **all** NCBI C++ API code -- both declarations and definitions (see examples). Inside these "brackets", all "std::" and "ncbi::" scope prefixes can (and must!) be omitted.

For code that does not define a new API but merely **uses** the NCBI C++ API, there is a macro USING_NCBI_SCOPE; (semicolon-terminated) that brings all types and prototypes from the "std::" and "ncbi::" namespaces into the current scope, eliminating the need for the "std::" and "ncbi::" prefixes.

Use macro NCBI_USING_NAMESPACE_STD; (semicolon-terminated) if you want to bring all types and prototypes from the "std::" namespace into the current scope, without bringing in anything from the "ncbi::" namespace.

Use of Include Directives

If a header file is in the local directory or not on the INCLUDE path, use quotes in the include directive (e.g. #include "foo.hpp"). In all other cases use angle brackets (e.g. #include <bar/foo.hpp>).

In general, if a header file is commonly used, it must be on the INCLUDE path and therefore requires the bracketed form.

Code Indentation and Bracing

4-space indentation only! Tabulation symbol **must not** be used for indentation.

Try not to cross the "standard page boundary" of **80** symbols.

In if, for, while, do, switch, case, etc. and type definition statements:

```
if (...) {
 .....;
} else if (...) {
 .....;
} else {
 .....;
}

if (...) {
 .....;
}
else if (...) {
 .....;
}
else {
 .....;
}

for (...; ...; ...) {
 .....;
}

while (...) {
 .....;
}

do {
 .....;
}
while (...);

switch (...) {
case ...: {
 .....;
 break;
}
} // switch

struct|union|enum <[S|U|E]TypeName> {
 .....;
};

class | struct | union <[C|I|P|S|U]TypeName>
{
 .....;
};

try {
 .....;
}
catch (exception& e) {
 .....;
}
```

Class Declaration

Class declarations should be rich in Doxygen-style comments. This will increase the value of the Doxygen-based API documentation.

```
/// @file FileName
/// Description of file -- note that this is _required_ if you want
/// to document global objects such as typedefs, enums, etc.

///////////////////////////////////////////////////////////////////////
///
/// CFooClass
///
/// Brief description of class (or class template, struct, union) --
/// it must be followed by an empty comment line.
///
/// A detailed description of the class -- it follows after an empty
/// line from the above brief description. Note that comments can
/// span several lines and that the three /// are required.

class CFooClass
{
public:
 // Constructors and Destructor:

 /// A brief description of the constructor.
 ///
 /// A detailed description of the constructor.
 CFooClass(const char* init_str = NULL); ///< describe parameter here

 /// A brief description for another constructor.
 CFooClass(int init_int); ///< describe parameter here

 ~CFooClass(void); // Usually needs no Doxygen-style comment.

 // Members and Methods:

 /// A brief description of TestMe.
 ///
 /// A detailed description of TestMe. Use the following when
 /// parameter descriptions are going to be long, and you are
 /// describing a complex method:
 /// @param foo
 /// An int value meaning something.
 /// @param bar
 /// A constant character pointer meaning something.
 /// @return
 /// The TestMe() results.
 /// @sa CFooClass(), ~CFooClass() and TestMeToo() - see also.
 float TestMe(int foo, const char* bar);

 /// A brief description of TestMeToo.
 ///
 /// Details for TestMeToo. Use this style if the parameter
 /// descriptions are going to be on one line each:
 /// @sa TestMe()
 virtual void TestMeToo
 (char par1, ///< short description for par1
 unsigned int par2 ///< short description for par2
 ) = 0;

 /// Brief description of a function pointer type
 /// (note that global objects like this will not be documented
 /// unless the file itself is documented with the @file command).
 ///
 /// Detailed description of the function pointer type.
 typedef char* (*FHandler)
 (int start, ///< argument description 1 -- what start means
 int stop ///< argument description 2 -- what stop means
 );

 // (NOTE: The use of public data members is
 // strictly discouraged!
 // If used they should be well documented!)
 /// Describe public member here, explain why it’s public.
 int m_PublicData;

protected:
 /// Brief description of a data member -- notice no details are
 /// given here since a brief description is adequate.
 double m_FooBar;

 /// Brief function description here.
 /// Detailed description here. More description.
 /// @return Return value description here.
 static int ProtectedFunc(char ch); ///< describe parameter here

private:
 /// Brief member description here.
 /// Detailed description here. More description.
 int m_PrivateData;

 /// Brief static member description here.
 static int sm_PrivateStaticData;

 /// Brief function description here.
 /// Detailed description here. More description.
 /// @return Return value description here.
 double x_PrivateFunc(int some_int = 1); ///< describe parameter here
 // Friends
 friend bool SomeFriendFunc(void);
 friend class CSomeFriendClass;

 // Prohibit default initialization and assignment
 // -- e.g. when the member-by-member copying is dangerous.

 /// This method is declared as private but is not
 /// implemented to prevent member-wise copying.
 CFooClass(const CFooClass&);

 /// This method is declared as private but is not
 /// implemented to prevent member-wise copying.
 CFooClass& operator= (const CFooClass&);
};
```

Function Declaration

Doxygen-style comments for functions should describe what the function does, its parameters, and what it returns.

For global function declarations, put all Doxygen-style comments in the header file. Prefix global functions with g_.

```
/// A brief description of MyFunc2.
///
/// Explain here what MyFunc2() does.
/// @return explain here what MyFunc2() returns.
bool g_MyFunc2
(double arg1, ///< short description of "arg1"
 string* arg2, ///< short description of "arg2"
 long arg3 = 12 ///< short description of "arg3"
 );
```

Function Definition

Doxygen-style comments are not needed for member function definitions or global function definitions because their comments are put with their declarations in the header file.

For static functions, put all Doxygen-style comments immediately before the function definition. Prefix static functions with s_.

```
bool g_MyFunc2
(double arg1,
 string* arg2,
 long arg3
 )
{
 .......
 .......
 return true;
}
```

```
/// A brief description of s_MyFunc3.
///
/// Explain here what s_MyFunc3() does.
/// @return explain here what s_MyFunc3() returns.
static long s_MyFunc3(void)
{
 .......
 .......
}
```

Use of Whitespace

As the above examples do not make all of our policies on whitespace clear, here are some explicit guidelines:

- When reasonably possible, use spaces to align corresponding elements vertically. (This overrides most of the rules below.)
- Leave one space on either side of most binary operators, and two spaces on either side of boolean && and ||.
- Put one space between the names of flow-control keywords and macros and their arguments, but no space after the names of functions except when necessary for alignment.
- Leave two spaces after the semicolons in for (...; ...; ...).
- Leave whitespace around negated conditions so that the ! stands out better.
- Leave two blank lines between function definitions.

Standard Header Template

A standard header template file, header_template.hpp, has been provided in the include/common directory that can be used as a template for creating header files. This header file adheres to the standards outlined in the previous sections and uses a documentation style for files, classes, methods, macros etc. that allows for automatic generation of documentation from the source code. It is strongly suggested that you obtain a copy of this file and model your documentation using the examples in that file.

Doxygen Comments

Doxygen is an automated API documentation tool. It relies on special comments placed at appropriate places in the source code. Because the comments are in the source code near what they document, the documentation is more likely to be kept up-to-date when the code changes. A configuration and parsing system scans the code and creates the desired output (e.g. HTML).

Doxygen documentation is a valuable tool for software developers, as it automatically creates comprehensive cross-referencing of modules, namespaces, classes, and files. It creates inheritance diagrams, collaboration diagrams, header dependency graphs, and documents each class, struct, union, interface, define, typedef, enum, function, and variable (see the NCBI C++ Toolkit Doxygen browser). However, developers must write meaningful comments to get the most out of it.

Doxygen-style comments are essentially extensions of C/C++ comments, e.g. the use of a triple-slash instead of a double-slash. Doxygen-style comments refer to the entity following them by default, but can be made to refer to the entity preceding them by appending the '<' symbol to the comment token (e.g. '///<').

Doxygen commands are keywords within Doxygen comments that are used during the document generation process. Common commands are @param, @return, and @sa (i.e. 'see also').

Please do not use superfluous comments, such as '/// Destructor'. Especially do not use the same superfluous comment multiple times, such as using the same '/// Constructor' comment for different constructors!

Please see the Doxygen manual for complete usage information. More information can also be found in the chapter on Toolkit browsers.

C++ Guidelines

This section discusses the following topics:

Introduction to Some C++ and STL Features and Techniques

C++ Implementation Guide

Use of STL (Standard Template Library)

Use the Standard Template Library (STL), which is part of ANSI/ISO C++. It'll make programming easier, as well as make it easier for others to understand and maintain your code.

Use of C++ Exceptions

- Exceptions are useful. However, since exceptions unwind the stack, you must be careful to destroy all resources (such as memory on the heap and file handles) in every intermediate step in the stack unwinding. That means you must always catch exceptions, even those you don't handle, and delete everything you are using locally. In most cases it's very convenient and safe to use the auto_ptr template to ensure the freeing of temporary allocated dynamic memory for the case of exception.
- Avoid using exception specifications in function declarations, such as:

```
void foo(void) throw ();

void bar(void) throw (std::exception);
```

Design

- Use abstract base classes. This increases the reusability of code. Whether a base class should be abstract or not depends on the potential for reuse.
- Don't expose class member variables, rather expose member functions that manipulate the member variables. This increases reusability and flexibility. For example, this frees you from having the string in-process -- it could be in another process or even on another machine.
- Don't use multiple inheritance (i.e. class A: public B, public C {}) unless creating interface instead of implementation. Otherwise, you'll run into all sorts of problems with conflicting members, especially if someone else owns a base class. The best time to use multiple inheritance is when a subclass multiply inherits from abstract base classes with only pure virtual functions.

NOTE: Some people prefer the Unified Modelling Language to describe the relationships between objects.

Make Your Code Readable

Use NULL instead of 0 when passing a null pointer. For example:

```
MyFunc(0,0); // Just looking at this call, you can't tell which
 // parameter might be an int and which might be
 // a pointer.

MyFunc(0,NULL); // When looking at this call, it's pretty clear
 // that the first parameter is an int and
 // the second is a pointer.
```

Avoid using bool as a type for function arguments. For example, this might be hard to understand:

```
// Just looking at this call, you can't tell what
// the third parameter means:
CompareStrings(s1, s2, true);
```

Instead, create a meaningful enumerated type that captures the meaning of the parameter. For example, try something like this:

```
//////////////////////////////////////////////////////////////////////
///
/// ECaseSensitivity --
///
/// Control case-sensitivity of string comparisons.
///
enum ECaseSensitivity {
 eCaseSensitive, ///< Consider case when comparing.
 eIgnoreCase ///< Don't consider case when comparing.
};
```

```
.....

/// Brief description of function here.
/// @return
/// describe return value here.
int CompareStrings
(const string& s1, ///< First string.
 const string& s2, ///< Second string.
 ECaseSensitivity comp_case); ///< Controls case-sensitivity
 ///< of comparisons.

.....

// This call is more understandable because the third parameter
// is an enum constant rather than a bool constant.
CompareStrings(s1, s2, eIgnoreCase);
```

As an added benefit, using an enumerated type for parameters instead of bool gives you the ability to expand the enumerated type to include more variants in the future if necessary - without changing the parameter type.

C++ Tips and Tricks

- Writing something like map<int, int, less<int>> will give you weird errors; instead write map<int, int, less<int> >. This is because >> is reserved word.
- Do use pass-by-reference. It'll cut down on the number of pointer related errors.
- Use const (or enum) instead of #define when you can. This is much easier to debug.
- Header files should contain what they contain in C along with classes, const's, and in-line functions.

See the C++ FAQ

Standard Template Library (STL)

The STL is a library included in ANSI/ISO C++ for stream, string, and container (linked lists, etc.) manipulation.

STL Tips and Tricks

end() does not return an iterator to the last element of a container, rather it returns a iterator just beyond the last element of the container. This is so you can do constructs like

```
for (iter = container.begin(); iter != container.end(); iter++)
```

If you want to access the last element, use "--container.end()". Note: If you use this construct to find the last element, you must first ensure that the container is not empty, otherwise you could get corrupt data or a crash.

The C++ Toolkit includes macros that simplify iterating. For example, the above code simplifies to:

```
ITERATE(Type, iter, container)
```

For more info on ITERATE (and related macros), see the ITERATE macros section.

Iterator misuse causes the same problems as pointer misuse. There are versions of the STL that flag incorrect use of iterators.

Iterators are guaranteed to remain valid after insertion and deletion from list containers, but not vector containers. Check to see if the container you are using preserves iterators.

If you create a container of pointers to objects, the objects are not destroyed when the container is destroyed, only the pointers are. Other than maintaining the objects yourself, there are several strategies for handling this situation detailed in the literature.

If you pass a container to a function, don't add a local object to the container. The local variable will be destroyed when you leave the function.

C++/STL Pitfalls and Discouraged/Prohibited Features

STL and Standard C++ Library's Bad Guys

Non-Standard STL Classes

- Don't use the rope class from some versions of the STL. This is a non-standard addition. If you have questions about what is/isn't in the standard library, consult the C++ standards.
- The NCBI C++ Toolkit includes hash_map, hash_multimap, hash_set, and hash_multiset classes (from headers <corelib/hash_map.hpp> and <corelib/hash_set.hpp>). These classes are more portable than, and should be used instead of, the STL's respective hash_* classes.

C++ Bad Guys

Operator Overload

Do not use operator overloading for the objects where they have unnatural or ambiguous meaning. For example, the defining of operator==() for your class "CFoo" so that there exist { CFoo a,b,c; } such that (a == b) and (b == c) are true while (a == c) is false would be a very bad idea. It turns out that otherwise, especially in large projects, people have different ideas of what an overloaded operator means, leading to all sorts of bugs.

Assignment and Copy Constructor Overload

Be advised that the default initialization {CFoo foo = bar;} and assignment {CFoo foo; ...; foo = bar;} do a member-by-member copying. This is not suitable and can be dangerous sometimes. And if you decide to overwrite this default behavior by your own code like:

```
class CFoo {
 // a copy constructor for initialization
 CFoo(const CFoo& bar) { ... }
 // an overloaded assignment(=) operator
 CFoo& operator=(const CFoo& bar) { if (&bar != this) ... }
};
```

it is **extremely important** that:

- **both** copy constructor and overloaded assignment be defined
- they have **just the same** meaning; that is {CFoo foo = bar;} is equivalent to {CFoo foo; foo = bar;}
- there is a check to prevent self-assignment in your overloaded assignment operator

In many cases when you don't want to have the assignment and copy constructor at all, just add to your class something like:

```
class CFoo {
 ............................
private:
 // Prohibit default initialization and assignment
 CFooClass(const CFooClass&);
 CFooClass& operator=(const CFooClass&);
};
```

Omitting "void" in a No-Argument Function Prototype

Do not omit "void" in the prototype of a function without arguments (e.g. always write "int f (void)" rather than just "int f()").

Do Not Mix malloc and new

On some platforms, malloc and new may use completely different memory managers, so never "free()" what you created using "new" and never "delete" what you created using "malloc()". Also, when calling C code from C++ **always** allocate any structs or other items using "malloc ()". The C routine may use "realloc()" or "free()" on the items, which can cause memory corruption if you allocated using "new."

Source Code Repositories

The following Subversion repositories have been set up for general use within NCBI:

Repository	**Purpose**
toolkit	C++ Toolkit (core and internal) development
gbench	GUI / GBENCH
staff	individuals' projects (not parts of any official projects)
misc_projects	projects not falling into any of the other categories

Note for NCBI developers: Using these repositories has the additional advantages that they are:

- backed up;
- partially included in automated builds and tests (along with reporting via email and on the intranet) on multiple platforms and compiler configurations; and

- integrated with JIRA and FishEye.

Testing

Unit testing using the Boost Unit Test Framework is strongly encouraged for libraries. Within NCBI, unit tests can be incorporated into the nightly automated testsuite by using the CHECK_CMD macro in the makefile. All testsuite results are available on the testsuite web page. Users can also be automatically emailed with build and/or test results by using the WATCHERS macro. Please see the chapter on Using the Boost Unit Test Framework for more information.

Applications should also be tested, and shell scripts are often convenient for this purpose.

Data files used for testing purposes should be checked into SVN with the source code unless they are very large.