C++ Coding Standard

Table of Contents

Do not edit the standard without the approval.

1. Style

1.1 Use consistent naming conventions throughout the project

Use one consistent naming convention throughout the project or, if necessary, a library. Naming conventions for a project are defined by its developers.

1.2. Use English-language names for identifiers

Assign correct English names to variables, functions, and classes. Avoid using any characters from other languages or transliteration.

1.3. Use appropriate prefixes or suffixes in the names of global variables, static variables and members of the class.

It should be made easy to tell global and static variables, as well as private data of the class, from local variables. To achieve this, the project naming convention should include standards of using special prefixes or suffixes for listed types of variables. Specific prefixes or suffixes are written down in the naming convention and are effective within the project.

Examples

Sample 1:

```
int g_globalVariable = 0;
int s_staticVariable = 0;
int m_privateClassMember = 0;
```

Sample 2:

```
int gGlobalVariable = 0;
int sStaticVariable = 0;
int privateClassMember_ = 0;
```

1.4. Use UPPERCASE_NAMES for macro names

Use uppercases and underscore characters for macro names. This naming scheme is not appropriate and should not be used for anything else. This approach makes it easy to tell macros from the rest, thus preventing the preprocessor from accidentally substituting a variable or function with a macro.

Examples

```
// OK

#define DO_ADDITIONAL_LOGGING

// wrong

#define max(x,y) (x) > (y) ? (x) : (y)
```

1.5. Use 4 spaces for indentation

Use 4 spaces for indentation. Avoid using tabs.

1.6. Avoid using constant literals (magic constants) unless their purpose is obvious

For example, the purpose of 0 index is nearly always obvious, as well as the purpose of incrementing the counter value by 1. If the purpose of a literal might be misinterpreted, use self-documenting language constructs (named constants, sizeof) or, at least, write a comment.

1.7. Avoid defining functions, classes, or structures in the body of a class except when the class size does not exceed 40 strings.

Classes of a large size are hard to read and individualize from the rest of the file content. Moreover, when reading a class that has another class declared within its scope, it is visually difficult to individualize the members of a nested class from those of the enclosing one. To prevent this, you should move the definitions of nested classes/structures and methods of a class to a separate module.

Exceptions

A template class can enclose the definitions of its methods.

1.8. Remove commented-out code

Commented-out code brings forth a lot of questions like:

- Should it be uncommented?
- Is it not needed anymore?
- When will it be needed?
- What is it for?

So, here is a simple solution – remove any commented-out code. You can always retrieve it via the revision control system.

2. Functions and Variables

2.1. Avoid passing large objects by value

Avoid passing large objects (over 2*sizeof(size_t)) to a function as input parameters **by value** as it results in taking up an unnecessarily large amount of memory. For the same reason, avoid returning large objects using **return**.

If you need to return a large object from a function, use out parameters (as a pointer or reference). If you need to pass a large object to a function, use a **constant** reference (or a constant pointer).

Exceptions

It is allowed to return std::string and std::wstring by value, especially if required for creating an error message or while writing to a log.

Examples

```
// OK

void ReturnVector(std::vector<char>* out);
void ProcessVector(const std::vector<char>& out);
std::wstring utils::string2wstring(const std::string& str);
std::string FormatMessage(const char* reason, DWORD code);

// wrong

std::vector<char> ReturnVector();
void ProcessVector(std::vector<char> out);
```

2.2. Always initialize variables

It is not allowed to use uninitialized variables. Always initialize variables using any given value. It is best to declare and initialize variables as close as possible to use.

2.3. Store global variables of non-PODType in auto_ptr/unique_ptr

All global variables, unless they are PODType, should be stored in unique_ptr or auto_ptr to control the lifetime and the order of calling destructors of such global objects. And anyway, try to avoid using global variables.

Examples

```
// wrong
int g_SomeNumber;
std::map<std::string, HANDLE> g_lotsOfOpenedFiles;
std::list<ATL::CEvent> g_bunchOfEvents;
HugeClass g_ComplicatedPieceOfTrash;

// OK
int g_SomeNumber = 0;
std::auto_ptr<std::map<std::string, HANDLE> > g_lotsOfOpenedFiles;
std::auto_ptr<std::list<ATL::CEvent> > g_bunchOfEvents;
std::auto_ptr<HugeClass> g_ComplicatedPieceOfTrash;
```

3. Types

3.1. Avoid using functions with ellipsis (…) as arguments

Even though functions with variable arguments are quite convenient, using unknown arguments (denoted with an ellipsis) is not the most efficient way to have such functions. When using unknown arguments, the compiler can check neither the types nor the number of passed arguments. It is necessary to come up with a way to pass this data to the called function. And the calling code should guarantee that the number and types of arguments are correct.

It is possible to use functions with unknown arguments from a standard library (for example, printf and its numerous derivatives). However, use typesafe alternatives when you can.

Examples

```
// wrong

void Foo(size_t count, ...);

// OK

void Foo(int a);
void Foo(int a, int b, int c);
void Foo(int a, int b, int c, int d, int h);
// wrong

LOG("%d little Soldier boys went out to dine;\nOne choked his little self and then
there were %d", 10, 9);
```

```
// OK

LOG(10 << " little Soldier boys went out to dine;\nOne choked his little self and
then there were " << 9);
```

3.2. Avoid using C-style cast

C-style cast uses one syntax for various type castings depending on the context, which raises up the risks of having difficult-to-locate bugs.

Please note that if you suspect that there is a type casting error, reinterpret_cast is easier to locate than C-style cast.

Example

```
// wrong, the programmer has forgotten to dereference the pointer, and the compiler
doesn't care

void foo(double *val)
{
    int v = (int)val;
}

// OK, the code won't compile, the error can be fixed at compilation stage

void foo(double *val)
{
    int v = static_cast<int>(val);
}
```

4. Classes and Inheritance

4.1. All class member data must be private

It is not allowed to declare data members with *public* and *protected* modifiers. *Public* data members violate class encapsulation in an evident way, thus making changes to the class code more complicated. *Protected* data members violate encapsulation in a less evident way, moreover creating unclear code of referring to ancestor data in descendant classes. Such code is not only difficult to modify but also harder to debug.

Instead, create Getter and Setter methods that allow accessing the class data.

4.2. Declare class members in the following order: public, protected, private

Always declare public class members first, then protected ones, and then private ones. This way, the person reading your code will see the interface first (*public)* and then, if necessary, the implementation details (*private*). Also, avoid declaring methods and data in the same section. If a class has private data and private methods, individual *private* sections are made for the methods and data.

4.3. Use initialization lists in constructors

Initialization lists allow you to initialize the class fields once and not twice as when you use assignment. They also make it simpler to read the constructor code and have no exception-safety related flaws.

Example

```
class File
{
public:


File(const Session* session, const NetworkRequest* packet);

...

private:
 const Session* session_;
 std::wstring fileName_;
};


// OK

File::File(const Session* session, const NetworkRequest* packet) :
session_(session)
, fileName_(file->GetPath())
{
}

// wrong

File::File(const Session* session, const NetworkRequest* packet) {

    session_ = session; fileName_
    = file->GetPath();

}
```

4.4. Use virtual destructors in interfaces (in classes that intend to use virtual behavior)

If you intend to create class descendants, the destructor of this class should be made virtual. A particular case is that **there must be a virtual destructor in each interface**.

However, you should not go to the other extreme and have virtual destructors everywhere “just in case”. Whether or not you are going to create class descendants should be decided on the stage of designing a class. Thus, if a class does not intend to use inheritance, it should not have a virtual destructor at all.

4.5. Declare all single-argument constructors as explicit

Using explicit for single-argument constructors prevents unevident type casting and difficult-to-locate bugs related to it.

Examples

// OK

```
class MegaString
{
public:
     explicit MegaString(unsigned int size);
     explicit MegaString(const char* str, bool someFlag = true);
...
};

// wrong
class MegaString
{
public:
 MegaString(unsigned int size);
 MegaString(const char* str, bool someFlag = true);
...
};
```

4.6. If there is only one class in a file, the file name contains the class name

If a file or a pair of files (h+cpp) contains only one class, the file name should contain the class name. If the C prefix is used for classes, the file name should not include it.

Examples

smb::Parser class is in files named *Parser.h* and *Parser.cpp*

cmn::Searcher class is in in files named *cmnSearcher.h* and *cmnSearcher.cpp*

CAboutDialog class is in files named *AboutDialog.h* and *AboutDialog.cpp*

5. Resource Management

5.1. Use wrappers for any resources (memory, handles, etc.)

Resource wrappers make refactoring much simpler, prevent resource leak, and expedite error handling.

Example

```
// OK

std::auto_ptr<SomeClass> ptr(new SomeClass());
HANDLE file = CreateFile(...);
HandleGuard fileGuard(file);

// wrong

SomeClass* ptr = new SomeClass();
HANDLE file = CreateFile(...);
```

5.2. Define ownership clearly using unique_ptr/auto_ptr. Avoid using shared_ptr unless there is real necessity

shared_ptr conceals resource ownerwhip and makes the process of its creation and destruction less apparent. Since shared_ptr does not allow to pass ownership, excessive use of it quickly results in shared resource ownership using shared_ptr's becoming the only ownership strategy in the project. In this case, it becomes impossible to define resource ownership in a clear way, which makes the project architecture too complicated.

Besides, shared_ptr imposes some penalties and might result in difficult-to-locate bugs with circular references.

5.3. It is not allowed to pass ownership using bare resources

Passing ownership using bare resources might result in difficult-to-diagnose resource leaks. Besides, it hides the very fact of passing ownership. Passing ownership using an appropriate wrapper, on the contrary, shows it clearly that resource ownership is being passed.

Example

```
// wrong
void take_ownership(my_class* object)
{
    ...
    delete object;
}

// OK
void take_ownership(std::auto_ptr<my_class> object) {
    ...
}
void doesnt_take_ownership(my_class* object);
{

    ...
}
```

6. Exceptions and Error Handling

6.1. Use *assert* for documenting inner assumptions. Avoid using *assert* for checking input data in public interface and API return results

Use *asserts* for documenting assumptions that must always execute. You can use *assert* only if the *calling* and *called* code are maintained by the same person/team. If the data that must satisfy some condition comes from an external environment, you should be ready for it not satisfying this condition.

When using *asserts*, you should also remember that you cannot perform any actions inside the check except for the comparison itself. Otherwise, the behavior in the debug build will be different from one in the release build.

Example

```
// OK

string Date::DayOfWeek() const
{

    assert( day_ > 0 && day_ <= 31 );
    assert( month_ > 0 && month_ <=
    12 ); ...
}

// wrong

HANDLE file = CreateFile(...);
assert(file != INVALID_HANDLE_VALUE);
```

6.2. All types of exceptions must be inherited from std::exception

Any exceptions thrown in the code must be inherited from std::exception. std::exception class objects cannot be thrown as this option is not guaranteed by the standard. No objects other than std::exception descendants can be thrown.

Example

```
// OK

class ParsingException: public std::exception {

    ...
};

throw std::runtime_error(...);
throw ParsingException(...);

// wrong

class ParsingException
{
    ...
}

throw ParsingException(...);
throw 5;
throw L"Catch me if you can";
throw CAtlException(...)
```

6.3. Avoid using catch(...)

catch(...) hides errors and, moreover, hides any changes happening to errors.

Exception

Use catch(...) only if an exception is rethrown in the catch block body (throw;).

6.4. Exceptions must stay within a module (an executive file or a shared library - .dll, .so, .dylib)

In most cases, throwing an exception outside a module results in the application crashing. Throwing an exception outside a shared library may result in various unpredictable effects depending on where and how the library is used.

6.5. Throw exceptions by value, catch exceptions by reference

Avoid throwing exceptions allocating them in the heap because error handling must not require any extra resources.

Also, avoid catching exceptions by value as it may result in losing descendant class fields when copying it to a base class.

Example

```
// OK

throw MyException(...);

...

catch(const MyException& ex)
{
    ...
}

// wrong

catch(const MyException* ex) {
    ...
}
catch(MyException ex)
{

    ...

}
```

6.6. Always check the results of function returns

The result of function return indicates if the function was executed successfully. Ignoring function return results deprives the developer of useful information on the stage of debugging. It can also result in in difficult-to-reproduce errors and unpredictable behavior in the cases when the previous function was not executed successfully and the value it had to set remained uninitialized, and the next call depends on this value.

6.7. Avoid throwing exceptions out of deinitialization functions, rollback functions, and destructors

This is important because such code can be called in guard destructors and rethrowing an exception may result in the application crash.

6.8. Error handling should not require resource allocation

Error handling should never require allocation of any resources, such as capturing concurrency controls, memory allocation, opening files, etc. because in case of breakdown these resources might be unavailable.

Exceptions

You can allocate memory for a string with an error description (to throw an exception or write to a log) and use std::string and std::stringstream.

6.9. The code must provide at least a basic exception safety guarantee

Basic exception safety guarantee means no leaks of memory or other resources in case of an exception. However, it is desirable to always provide a strong exception safety guarantee, which means that, in case of an error on executing an operation, an object must remain in the same state as before the operation.

Example

```
 // wrong

void foo()
{
    MyClass* obj = new MyClass;
    bar(obj); // memory leak is possible
    delete obj;
}
// OK
void foo()
{
    std::auto_ptr obj(new MyClass);
    bar(obj.get());
}
```

6.10. Avoid using exception specifications

Avoid writing exception specifications for your functions except for an empty throw() specification.

6.11. Avoid using try-finally blocks in user-mode code. Instead, use guards placing code in their destructors

Try-finally duplicates the class destructor functionality but is not transferable. It makes the code more complicated and expands its width.

7. Header Files

7.1. When referring to header files, avoid using parent folders in relative paths

Avoid using parent folders in relative paths, for example, "..\..\..\header.h". This rule helps to make the project parts less dependent on each other. To refer to a header file from another directory, define external paths in the build configuration (for example, for Visual Studio it is Additional Include Path in the project compilation settings).

Examples

```
// wrong
#include "..\..\Common\Utils\cmnTextStuff.h"
#include "..\SmbUtils\Headers.h"
#include "..\..\..\..\..\boost\boost_1_47_0\boost\noncopyable.hpp"

//OK
#include <boost/noncopyable.hpp>
#include "cmnTextStuff.h"
#include "SmbUtils/Headers.h"
```

7.2. Protect header files against multiple inclusion using #ifndef/#define or #pragma once

To protect a header file against multiple inclusion, use the following code **inside** the header file:

```
#ifndef MY_COOL_HEADER_H
#define MY_COOL_HEADER_H

// ...

#endif
```

If the compiler allows, you can use #pragma once:

```
#pragma once

// ...
```

7.3. Avoid using *using namespace* in header files

Avoid using **using namespace** in header files and qualify all names completely. Also, avoid using **using** before including header files as it clutters up the global name space and may result in unpredictable compilation or linking errors.

If you really need to use **using**, do it in cpp files after all **include** directives.

7.4. Use forward declarations in header files if possible

Use forward declarations everywhere where a full definition of type is not required. (Full definition is required in two situations, such as 1) if you need to know the class size, 2) if you need to call its methods.) Do not include declarations using **#include** if a forward declaration is enough.

Recommendation

If a forward declaration of a stream is enough, you can use the <iosfwd> file.

8. Specific for Pure C

8.1. Use *goto clean* to clean up resources and cancel actions in case of an error in C

Such a technique allows to decrease the code width and avoid code duplication. This, in its turn, allows to avoid difficult-to-reproduce errors caused by incorrect release of resources in case of an error. The **clean** tag must be always at the end of a function, and this way of using **goto** is the only appropriate one.

Example

```
NTSTATUS PatchSomethingInCurrentProcess_hooks( ... )
{
     void * pHook = 0;
     NTSTATUS status = STATUS_SUCCESS;

     status = PL_CreateUserModeHookEx( ... ); if
     (status != STATUS_SUCCESS)
          goto exit_and_clean;

     ... // lots of letterz

exit_and_clean:
    if (pHook)

          PL_FreeUserMemory(pHook);
     return status;
}
```

9. Specific for Win32

9.1. In a DLL, all data that needs global access must be stored in *one* global variable

This allows to simplify the control over initialization and deinitialization of global objects and get rid of bugs related to dependencies between global objects and untimely initialization of global objects. Control over initialization and deinitialization of global objects is especially important for the code in a DLL as the code executed during the dynamic library loading undergoes some additional restrictions. For more details, see MSDN - Best Practices for Creating DLLs.

```
static std::auto_ptr<CMyDll> g_myDll;

BOOL WINAPI DllMain(

     IN HINSTANCE hinstDll,
     IN DWORD fdwReason,
     LPVOID lpvReserved
     )
{
```

```
....

case DLL_PROCESS_DETACH:

        if( lpvReserved == NULL ) {
            g_myDll.reset(0);
        }
        g_myDll.release();
        break;

}
```

9.2. In DllMain, avoid calling LoadLibrary and waiting on concurrency controls. Use functions only from kernel32 (and ntdll)

Do your best to keep DllMain as simple as possible. Do not ever call *LoadLibrary*, avoid waiting on *concurrency controls*, avoid using any functions other than those in *kernel32.dll* (and ntdll.dll) as all of these actions are prohibited in MSDN.

Recommendation

If possible, use your own DLL initialization and deinitialization functions instead of DllMain.

10. Other

10.1. Use constants, enums and inline functions instead of macros

Minimize the use of macros where possible. This applies primarily to constant values and inline functions.

Exceptions

1. Macros might be needed in constructs that use __FUNCTION__, __LINE__ and other predefined macros.
2. Macros should be used to protect header files from multiple inclusions.
3. Macros are used for conditional compilation.
4. Macros can be used (with caution) as code generation.

10.2. Always choose the capabilities of standard libraries over the similar ones of third-party frameforks (ATL, MFC, Qt, etc.) unless there are *serious* reasons not to

It will provide greater code portability and a lower learning curve of the project. It will also make it easier to handing over the project to another developer. If Boost and standard libraries have the same functionality (e.g. shared_ptr), choose the implementation from the standard library.

10.3. Avoid using tr1

Avoid using the functionality of tr1. Using tr1 as it is right now can make the code non-portable (gcc/llvm/VS). So it is reasonable to refrain from using tr1 in the production code. If the standard library of the compiler used supports C++11, use the functionality of std. If not, use the implementation from Boost.

10.4. Compile without warnings

You should be serious about the compiler warnings and never ignore them. Warnings often point to potential errors in the code so the cause of each warning must be eliminated. If possible, eliminate warnings by changing the code rather than lowering the warning level or disabling them. In the project settings, always set the warning level to not lower than W3 (for Visual Studio).

Obviously, this rule (as well as the whole standard) does not apply to the legacy code. Nevertheless, if you are editing some older code, try to reduce the number of warnings and, of course, do not add new ones. Even if there are 9000 of them, adding the 9001st one is unacceptable.

Recommendation

To make sure the project does not contain any warnings, you can enable the Treat Warnings As Errors option in the settings.

Examples

```
// warning C4127: conditional expression is constant
whlie (true)
{
...
}

// OK :-)

for (;;)

{
    ...
}



// warning C4018: '<' : signed/unsigned mismatch for
(int i = 0; i < myVector.size(); ++i)
{
    ...
}

// OK :-)
for (size_t i = 0; i < myVector.size(); ++i)
{
 ...
}
// warning C4244: 'argument' : conversion from 'ULONGLONG' to 'DWORD', possible
loss of data
ULONGLONG bufferSize = ...;
WriteFile(buffer, bufferSize, ...);

// OK :-)
ULONGLONG bufferSize = ...;
WriteFile(buffer, boost::numeric_cast<DWORD>(bufferSize), ...);




// warning C4512: 'boost::program_options::options_description' : assignment
operator could not be generated
#include <boost/program_options.hpp>


// OK :-)
#pragma warning(push)
#pragma warning(disable:4512)

#include <boost/program_options.hpp>

#pramga warning(pop)
```

10.5. The code of static libraries must not contain any references to the product name or the customer

The code of static libraries must contain neither references to the product name or the customer nor paths in the file system or registry. This information can be contained only within the code of executable files and dynamic libraries.

Recommendation

It is reasonable to put as little logic as possible into the code of executable file and dynamic libraries. It is better to place all the logic into static libraries. This way, it is even easier to test and reuse.