C++ Coding Standards and Practices

Tim Beaudet *(iam@timbeaudet.com)*
2018-11-30

Table of Contents

About these standards

These standards are put in place so all personal, collaboration and professional code will remain easily readable, clear and concise. This allows engineers using the standard to leverage the most from C++ compilers.

Above all else, code should be written for readability and maintainability. Never write a portion of code with “I am glad I will never see this again.” in mind -- it will be seen again and it will need to be maintained. Preparing code for readability will help avoid future mishaps.

Project

Must be built at the highest warning settings possible, with warnings treated as errors!
Files and directories must be named with all lower-case letters, underscore between words.

- my_file.txt

Source Control

SVN must be used on any project that;

- May possibly take more than 48 hours to complete.
- Has more than a single developer.

Build Automation

Note: The following is not yet in place as a standard, but is a goal that is being worked towards.

In the future every project should be capable of being added to a nightly build system that will pull the latest from the project repository, run an automated build script and run tests associated with the project. Given this is not currently implemented, the following should be taken lightly.

All projects should use premake4 to create the Visual Studio, XCode or other IDE project.

Const Correctness

Strive to use const wherever possible and correct to do so; const modifiers do not incur

performance penalties, but does make intent clear and prevents unintended changes to an object.

Local Variables

Create local variables as constants whenever they will not change within their scope, this includes holding a value from a called function, prefer holding a const reference over value.

```
const Type& value(GetSomeValue());
const Type value(GetSomeValue());

Correct:
{
	const int localVariable = 10;
	… //Never writes to localVariable
}
```

Passing / Returning Parameters

If a function does not modify an incoming parameter, it should be clearly marked using const.

```
void SomeFunction(const int withCount);
void SomeFunction(const Type& withCount);
void SomeFunction(const Type* const withCount);
```

Class Objects

Make the object itself const if a method will not change any internals by adding the const modifier to the end of the method declaration. This will force the this* within to be const.

```
class CSomeClass
{
public:
	int MethodOfClass(void) const;
};
```

Header Files

Header Protection

Every header should internally protect itself from multiple includes using the following technique:

Wrong:

```
#ifndef _ProjectName_FileName_h_
#include “FileName.h”
#endif
```

Correct:

```
#ifndef _ProjectName_FileName_h_
#define _ProjectName_FileName_h_
/* All header contents */
#endif /* _ProjectName_FileName_h_ */
```

Defining Functions (Inlining)

A header file should be used for declarations of types and functions only, however if code must be inlined it should be placed under all declarations within the header file. This is for readability, when looking through a header, one should not be concerned with implementation details.

Correct:

```
namespace Example
{
        class MyType
        {
        public:
                MyType& GetSomething(void);
        }
}

//---------------------------------//

inline MyType& Example::MyType::GetSomething(void)
{
         /* code */
}
```

The exception is for one line accessors and mutators, that simply set or return a value.

Dos and Don’ts

- Don't use 'using' within a header file. It will force any includers to automatically use.
- Do minimize including by forward declaring types whenever possible.
- Do include or forward declare everything necessary for the header file to compile.
- Do not define variables within a header file, only declare them as needed.

Source Files

Locally Defined Types

Locally defined types must be declared within an anonymous, unnamed or named namespace.

- This is to prevent undefined behavior if the type is declared in two separate translation units. Consider:
  In my_foo.cpp: struct foo { int x; char z; }
  In your_bar.cpp struct foo { float y[3], bool a; }
  This is not a linker error stating it has been defined, it might be a warning at best, but is undefined behavior.

Code Formatting

Indentation

Use tabs for indenting code, braces etc.
Tabs should be set to four spaces. *(This shouldn't matter if using tabs properly)*

Spacing

A space must precede and follow binary operators: +, -, *, << etc:

```
Wrong:    myVariable+=10;
Correct:  myVariable += 10;
```

Scope Braces

Scope braces should be placed on separate lines, and the contents of the scope tabbed.

Wrong:

```
while (condition) {
    //contents of loop
}
```

```
while(condition)
        {
        //contents of loop
        }
```

Correct:

```
while (condition)
{
        //contents of loop
}
```

Text Width

All lines of code should be less than 120 characters wide, use the IDE to insert a marking line. If a line extends a few characters beyond, then feel free to leave it if it more readable left that way, otherwise break it up into smaller lines as needed for readability.

Try to aim for lines of less than 120 characters, rather than breaking the line into multiple parts. Broken lines are difficult to read.

Classes

- Every class should resemble one simple concept, and only one concept.
- Hide members and implementation details as much as possible.
- Always explicitly disable, or enable, copy constructors and assignment operators.
- If a class has virtual methods, it must have a virtual destructor as well.
- Must have public members than methods, followed by protected and finally private.

Correct:

```
class Example
{
public:
        int mPublicData;
        /* Other public members */

        void PublicMethod(void) const;
        /* Other public methods */
protected:
```

```
        int mProtectedData;
        /* Other protected members */

        void ProtectedMethod(void) const;
        /* Other protected methods */
    private:
        int mPrivateData;
        /* Other private members */

        void PrivateMethod(void) const;
        /* Other private methods */
    };
```

- Overridden virtual functions must be marked with virtual, *(**and** override if using C++11).*
- Always use explicit constructors for single argument constructors to avoid implicit-casting.

  Wrong:

```
    class Example
    {
    public:
        Example(const int startValue);
    };
```

  Correct:

```
    class Example
    {
    public:
        explicit Example(const int startValue);
    };
```

- Never give direct access to contained objects. Even returning read-only contained objects makes code more dependant on other parts. Sometimes this is necessary, though it should never be necessary to give writable access.

  Wrong:

```
    class Example
    {
    public:
        ObjectType& GetObject(void) { return mObject; }
        ObjectType* GetObjectPtr(void) const { return &mObject; }
    private:
        ObjectType mObject;
```

```
};
```

Acceptable: (with reasoning)

```
class Example
{
public:
    const ObjectType& GetObject(void) const { return mObject; }
    const ObjectType* GetObjectPtr(void) const { return &mObject; }
private:
    ObjectType mObject;
}
```

You will notice in the wrong example, a non-const pointer to mObject can be returned from a const method on the Example object, which can be indirectly changing the Example object.

Namespaces

- Use namespaces to keep a collection of similar concepts contained.
- Never use a namespace or part of a namespace within a header file.
- Place function declarations within a namespace, then use the Scope-Resolution Operator to define the function. *Allow the compiler to help spot type-safety errors that would be linker errors.*

Header File:

```
namespace Example
{
    void DoSomething(const int withValue);
}
```

Source File:

Wrong:

```
namespace Example
{
    void DoSomething(const int withValue) { /* codes */ }
}
```

Correct:

```
void Example::DoSomething(const int withValue) { /* codes */ }
```

Functions

Naming

Must start with an uppercase letter, and uppercase first letter of each word thereafter.
Must be clear what the function does from reading the name.
Accessors should be named as follows: GetVariableName(), SetVariableName().
Never abbreviate function names when the full word is more descriptive.

HandleMouseClick();

Declarations / Prototypes

Must always exist in a class or namespace of similar functions.
Must use void when taking no parameters:

```
void DoSomethingWithNothing(void);
```

Getters and Setters should be named: GetVariable(), SetVariable()
IsVariable() is also an acceptable getter for a boolean value.

Function Separators

Before and after each function a separator line should be placed to enhance readability, a new line should exist before and after the separator line. Separator line should extend from left margin to the max width specified in the **Code Formatting** section of this document.

Correct:

```
//-----------------------------------------------//

void Example::DoSomething(void)
{
        /* insert code */
}

//-----------------------------------------------//
```

Documentation

Must follow the doxygen format, and be placed above the declaration of the function.

Variables

Naming

Must start with a lowercase, and uppercase first letter of each word thereafter.
Must be clear what the variable is holding or purpose is, based on the name.
Must never use single letter variables, even for temporary variables except for:
    loop counters, using (i, j or k)
Never abbreviate variable or function names when the full word is more descriptive, except for:
    itr within a loop iterating a container.

Member variables should start with an m following uppercase letter.
Constants should start with a k following uppercase letter.
Global variables should be used very rarely, if ever. Start with a g following uppercase letter, alternatively starting with 'the' is also acceptable, typically reserved for singletons.

Locals / Parameters: elapsedTime
Class Members: mIsVisible
Constants: kThisNeverChanges;
Globals: gVisualSystem, theVisualSystem

Type-Casting

Should be avoided when possible (including int to float, unsigned to signed, or base to derived). Allow the type-safety in the C++ language to show weak points in design by avoiding the need for type-casting. When a cast is necessary, use C++ style casts: static_cast<>, dynamic_cast<> etc. Avoid reinterpret_cast<> at all costs since it is essentially undefined behavior. When a cast is needed, be sure to check value ranges and values to be as safe as possible.

No C-Style casts are allowed. They are potentially very dangerous, properly use static_cast, const_cast, reinterpret_cast and dynamic_cast; in general stick with static_cast. *Note: dynamic_cast can only be used if RTTI is turned on, usually disabled by default.*

Wrong:

```
        float val = 30.0f;
        int intVal = (int)val;
```

Correct:

```
        float val = 30.0f;
        int intVal = static_cast<int>(val);
```

Dos and Don’ts

Do always initialize a variable to some expected value, NEVER allow it to be undefined!

Switch Statements

Should have a case for every value of an enum, consider if using the default case is appropriate to make this happen.

Standard Template Library

The STL is part of the C++ standard, use it instead of customized containers.
Do not “using namespace std”, instead be explicit by writing: std::string value;
When in need of a container use std::vector by default, otherwise choose the best container.
Use std::string instead of C-style char arrays. Use std::string::c_str() to exchange if needed.

Macros / Preprocessor

Macros should be avoided at nearly all cost, instead use Enums, Typedefs, Templates, and other built in C++ language features. When using a macro, be sure the use of a macro actually enhances readability, and avoid using clever preprocessing features that may change from compiler to compiler.

Macro names shall be all lowercase with underscores between words.