C++ Coding Standards

Version 1.1

September 2000

www.arcticlabs.com

Introduction

These standards represent what we believe is a good set of off-the-shelf construction guidelines for C++ development.

They are intended to be used as is, or as a base for people to create their own. If you do not agree with some of the points, you are free to make any change you wish. If you think you have a good point that we could incorporate in future versions mail us at feedback@arcticlabs.com.

If you create your own document that uses large parts of this standard, please add a reference to explain that your standard is based on the coding standards available from www.arcticlabs.com.

The most recent version can always be downloaded from http://www.arcticlabs.com/.

Variables

Variables should be made up of full words, not shortened. Eg numberOfPoints, not NumPts or np.

The wider the scope of the variable, the more descriptive it should be.

Don't use underscores in variable or function names.

Variables should not be reused for different purposes within the same routine.

Avoid using the unsigned types if you don't have to. The 'unsigned' keyword clutters function declarations and has no benefit if the extra headroom isn't required.

Variables should be declared close to where it is used. Don't put all the variable declarations at the beginning of a function.

Never use global variables in projects larger than trivial. You can't tell where or when they are used and they might have initialization sequence problems. Try using static class members and/or a singleton class to manage widely used variables.

Constants should be defined in the smallest scope possible, for example if a constant is only used in one function it should be defined in the function.

Use the C++ type bool for boolean variables.

Comments

Clear code is preferable to well-commented messy code. Every effort should be made to make code clear through good design, simplicity, good naming and layout. Comments are definitely required where code is not clear.

Avoid compulsory comments like author, date, class names, function names, and modification lists as they get outdated quickly and are usually unnecessary. If required, try to use source control or other tools to insert this type of information.

Use // to comment out code, this will allow you to comment out big blocks of code without interference using with the /**/ construct.

Files

Every class should have a separate source file and header file that matches the name of the class. For example class FooBar should be defined in FooBar.h and be located in FooBar.cxx.

The extension of the implementation file (eg .c .cc .cpp .cxx) will depend on your compiler. .Cpp is possibly the most popular, followed by .c.

Use precompiled headers. This can greatly improve compilation speed, but also provides a consistent place to #include all system and library headers, so that they don't have to be included anywhere else.

Avoid adding a full path or relative path to #include statements. Instead add the directory or root directory to the include directories list in the makefile. For example have “…/path/libraries/” as an include directory and have #include "LibrarySubDir/filename.h" in the precompiled header.

Do not put #includes in header files, except for base classes. This can greatly increase compilation speed. This can cause problems if the header has other classes as function parameters or class member variables. To allow this to work, pre-define the class in the header and hold a reference to members, where the member is created on the heap by the constructor (& of course deleted by the destructor).

Clear Code Style

There should be only one operation per line. For example the commonly seen line

```
if (doSomething()== false) ...
```

should be split into two lines.

```
result = doSomething();
if (result == false) ...
```

Avoid using 'goto', however it can be useful to have a goto to jump exit out of a series of nested loops. A preferable solution would be to place the loops into a separate function with a return statement to end the looping.

Don't use hungarian notation (where a code describing the variable's type is prefixed to the variable name) for variables. Hungarian notation can be useful where it aids code understanding, for example casting wide strings to a normal string.

Formatting & Layout

Layout should be written for maximum code readability.

Use #include <filename.h> for library & system headers.

Use #include “filename.h” for non-system headers.

Don't check equivalence backwards eg: if (false == variableA). It may be safer but it is more difficult to read.

Each variable declaration needs a line of its own.

If the variable declaration is a pointer, place the star next to the type, not the name. Eg char* p not char *p;

Header files should use the '#ifndef/#define/#endif' mechanism to prevent multiple inclusions. The defined name should be of the form _Filename_H.

Don't name variables with a leading underscore as this is used by systems to indicate non-standard extensions.

Macros (if you use them) should be in upper case.

Use spaces instead of tabs in a source file.

The tab size for source files should be set to every four spaces.

Classes should start with a capital letter.

Functions and variables should be mixed case and begin with a lower case letter.

Class-wide variables start with m eg mNumberOfBars.

Class-wide static variables start with ms eg msNumberOfInstances.

If you ever had a global variable it would start with a lowercase g.

Place a space between operators such as + and ||.

Use parentheses wherever they may remove possible ambiguity in the minds of the reader.

Matching braces ('{' '}') should line up vertically.

Place single spaces immediately after commas in routine parameter lists.

Safe C++ coding

Routines and their parameters should be declared 'const' wherever possible.

Use the 'assert' macro to test assumed or expected conditions.

Assignment operators should return a reference to the object eg
Foo& Foo::operator=(const Foo& other);

Classes that will be inherited from should have a virtual destructor.

If a class defines either an assignment operator or a copy constructor it should define both.

Don't throw objects allocated from the heap. You don't know if they will be freed.

If you catch an exception, make sure you have a default catch(...)so that all exceptions are caught at that point.

If you are writing a library or subsystem that will be used by a different project, it must be in a namespace.

Use set/get functions instead of public variables. If you have a getCount() function instead of a public member you can add a breakpoint to the function to see when it is being used. If you have a lot of public variables, consider a structure without code and separate the code from the data.

Use references instead of pointers if possible. A reference cannot be changed to point to another object so that you know it always points to the same object. This is especially useful for function parameters.

Reducing C++ complexity

Poorly written C++ can be very difficult to understand. It can be simplified by not using many of the features of C++: some of which are leftover from C, some which are poorly understood by many developers and some which are unnecessary.

#ifdef/#endif should not be used in a function to make it portable. Use wrapper classes to cover platform-dependent code.

Avoid macros. Always use constants or functions if possible as they are clearer and are more easily understood by software tools such as debuggers and source browsers.

Avoid c library functions which are superceded by modern objects, e.g. use string class methods rather than strcmp, use streams instead of c runtime functions.

Always try to use STL classes over proprietary or third party classes.

Don't use optimization hint keywords register and volatile.

Use if statements instead of ?:.

Don't use run time type information (RTTI) unless you have to. The overhead is unnecessary and such a mechanism shouldn't normally be needed with good software design and template-based collections.

Avoid using excessive use of templates. Templates are excellent for utility classes such as collections but abstract base classes are usually a better design and are clearer to understand.

Avoid operator over loading. Using a method on an object is much clearer and easier to follow than a called operator.

Portability

Try to avoid platform specific api calls. This will depend on the type of application, but a generic program core should call objects that can polymorphically adapt to different platforms.

Avoid pointer arithmetic.

Never assume a pointer length is a given length.

Use long instead of int.

If you know code will be ported to a new platform, begin porting early, so that you have no surprises when you have finished one platform and wish to port it to another.

Most Important Rule

Keep it simple. Elegant design is always better than complex code.