```
$Id: c++guidelines.mm,v 1.10 2013-01-03 18:42:15-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps109-wm/Coding-style
URL: http://www2.ucsc.edu/courses/cmps109-wm/:/Coding-style
```

The following are general programming guidelines for C++. This file should serve as a checklist for programmers. Some of these rules are requirements and some of them are general rules which may be ignored with suitable justification.

1. Prerequisite

It is assumed that the reader has already read the CMPS-012B coding style guidelines, available in the directory `/afs/cats.ucsc.edu/courses/cmps012b-wm/Coding-style/`.

2. Filenames

(1) Always submit a `Makefile` spelled exactly that way, with a capital "`M`". The first target should be `all` and perform the build.

(2) On `unix.ic`, use `g++ -g -O0 -Wall -Wextra -std=gnu++0x`.

(3) C++ includes from the standard library should not contain a dot. For example, use `<iostream>` and not `<iostream.h>`. Follow the includes with `using namespace std`.

(4) For C includes, generally delete the `.h` and put the letter `c` in front. So for example, use `<cstdlib>` and not `<stdlib.h>`.

(5) The only time that it is acceptable to use a standard C library include is when the same is not available in the C++ include headers.

(6) Implementation files should always end in `.cc` or `.cpp`, as in `hello.cc` or `hello.cpp`. But you must be consistent: all source files you submit must end with the ***same*** suffix. ***Do not*** use any of the suffices `.c`, `.C`, `.cxx`, `.c++`, or `.i`.

(7) All user-supplied header files must end with the suffix `.h`, as in `foo.h` as the interface to `foo.cc` or `foo.cpp`.

3. General

(1) No one letter identifiers. Ever. Identifiers should be long enough or unique enough so that a `grep`(1) for an identifier should pick up only those identifiers and specifically related identifiers, and possibly words in comments and strings which are spelled in the same way. Identifiers should always consist of words or phrases with no unusual abbreviations.

(2) No `goto`s. Ever. Especially no `goto`s to a pointer or to an array element! Use `break`, `continue`, or throw an exception.

(3) Do not call the function `exit`. Instead, throw a special exception to be caught in `main`, so that main can `return` an exit status.

(4) If you do use a `goto`, you will be haunted by the ghost of Edsger Dijkstra [EWD215]. On the other hand, he will probably haunt you on general principles just for using C++ [EWD498]. He didn't like UCSC anyway [EWD714].
`http://www.cs.utexas.edu/~EWD/`
`http://www.cs.utexas.edu/~EWD/ewd02xx/EWD215.PDF`
`http://www.cs.utexas.edu/~EWD/ewd04xx/EWD498.PDF`
`http://www.cs.utexas.edu/~EWD/ewd07xx/EWD714.PDF`
`http://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD215.html`
`http://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD498.html`

(5) Do not define external variables. However, due to the legacy libraries inherited by C++ from C, it is sometimes necessary to reference external variables defined by these libraries. An `extern` declaration which refers to a global variable documented in the C library is acceptable when there is no other way of accessing it. If you think you need to define an external variable, what

you probably really need is a private static data member. Static variables of file scope are deprecated, but not prohibited.

(6) Don't use `<stdio.h>`. Instead, use `<iostream>`, `<iomanip>`, etc.

(7) Other C header files may be used, but in their C++ `#include` form: `<cassert>`, `<cctype>`, `<climits>`, `<cmath>`, etc. You probably want to include the statement `using namespace std;` for most C++ code.

(8) Never use `malloc`(3c) or `free`(3c). Instead, use the C++ operators `new` and `delete`.

(9) If an object was created with `new`, then it should be destroyed with `delete`. If the call to `new` used `[]` to allocate an array, then use `delete[]` to destroy it.

(10) Indentation must be consistent. Anything inside of braces (`{` and `}`) must be indented three spaces more than what immediately precedes and follows the braces. An open brace (`{`) must never be on a line by itself, always being put at the end of a line that controls it. A closing brace (`}`) must always be on a line by itself and aligned underneath the keyword which introduced the construct that is being ended.

(11) Do not use C-style casts such as `(t)x`. Instead, use `static_cast<t>(x)` where possible. Use `dynamic_cast` when run time type information (RTTI) is needed.

(12) Avoid using `reinterpret_cast` unless you absolutely must. And then be aware that its behavior is implementation dependent. Don't use `union`s either, for the same reason. If you think you need a union, you probably want inheritance instead.

(13) Use of `++`, `--`, or any assignment operator, `=`, `+=`, `-=`, etc., inside of expressions is deprecated. A statement should alter one object and only one object.

4. Classes

(1) All data members should be private. There may be the occasional exception in very lightweight classes, but public data members are probably wrong. Protected data members are also probably wrong.

(2) All classes should be declared with a destructor in order to indicate that it was not forgotten. It may never be disabled, but its body may be just `{}`. All pointer variables in a class must be `delete`d from inside of the destructor in order to avoid memory leak. That is, if a pointer is the last pointer to a particular object.

(3) All classes should have a default constructor declared which properly initializes all of the data members. Default construction may be disabled by declaring the default constructor as a private member and not implementing its body.

(4) All classes should have a copy constructor declared. It's single argument is a constant reference to an object of its own class. It may be disabled if the client should not use it.

(5) All classes should have an `operator=` declared. The return type and argument should both be a constant reference to its class. It may be disabled. If a class contains pointers, the default `operator=` is certainly wrong.

(6) An implementation of `operator=` should always check if its argument is the same object as `this`.

(7) A class's single-parameter constructors and `operator=` should be semantically related so that the statements `T x; x = y;` mean exactly the same as `T x (y);` or `T x = y;`, although the former may be less efficient. This must be true no matter the type of `y`.

(8) All constructors which take a single parameter should be marked `explicit`, because otherwise they are also implicit conversion functions, unless a specific decision has been made that such a conversion is reasonable.

(9) Data members which have identical values in all instances of the class or which should be shared among all instances of the class are declared `static`. If they are of an integral type and their value is known at compile time, they should be `enum` constants instead of data members.

(10) A class should have good cohesion. That is, a class should have one purpose and serve it well. The general purpose of a class should be describable in one sentence without a long list of things. The same is true of programs. A text editor, for example, should not be able to read news, read mail, or do word processing; nor should an operating system contain a windowing system or a web browser.

(11) A class should have minimal coupling. That is, a class should keep as much as possible private and away from the client. The less that appears in the interface, the less that the client needs to remember in order to properly use the class.

(12) Avoid inline functions in header files where possible. Whenever you have a choice between putting something either in an interface (`.h`) file or in an implementation (`.cc`) file, put it in the implementation file.

5. Operators and Friends

(1) For a class `foo`, declare: **`friend ostream& operator<< (ostream&, const foo&)`**, which will be able to format and print an object in some reasonable default format.

(2) Operators should be defined only when they will not violate the Principle of Least Astonishment. They should be chosen to be semantically consistent with builtin and library uses of the operators.

(3) The operator and assign form of operators should be consistent with the base form. For example, `a += b` should always be interchangeable with `a = a + b` the only possible differences being matters of efficiency. Also, `++a` should always be identical to `a += 1`, and, of course, to `a = a + 1`. And `a++` should always be identical to `({t = a; a += 1; return t})`. (This last expression is a Gnuism.) Consider not implementing either the prefix nor postfix form of `++` and `--`.

(4) Binary operators must have one parameter if implemented as member functions and two parameters if implemented as non-member functions. Unary operators must have zero parameters if implemented as member functions and one parameters if implemented as non-member functions.

(5) The operators `++` and `--` are implemented as unary operators in their prefix form and as binary operators with a spurious `int` argument to indicate their postfix form.

(6) It would be nice to use `operator**` as the exponentiation operator, as is done in other languages, but you can't, because `a ** b` means `a * (*b)`. And don't define `operator^` as exponentiation either, since its precedence is wrong for that and it means exclusive or. An exponentiation operator should be a binary operator with a higher precedence than multiply and divide, but there are none.

(7) Relational operators are related. After the statement `a = b`, it should always be the case that `a == b`. Whenever `a == b` is true, `a != b` should always be false. Similar comments apply to `<`, `<=`, `>`, and `>=`. The relational operators and `!` should always return a `bool` result.

(8) If it is necessary to use an object in a boolean context, define an `operator bool` to implicitly convert the class to boolean. Do not use the older style of providing an implicit converstion to `void*`, which was a hack to get the same effect before `bool` was in the language. Better yet, consider member functions such as `good()` or `error()` instead, for use such as: **`if (x.good())`** ...

(9) Define related operators. For example, if `a + b` is meaningful, then implement the reverse operator so that `b + a` provides a meaningful result, although the second will have to be defined via the friend mechanism. Side note: `operator+` is not always commutative, as that depends on the objects being handled; e.g., not all string and matrix operators are commutative.

6. Single Inheritance

(1) Any class that has any virtual function must have its destructor declared as virtual.

(2) If a class is involved in inheritance, all function members probably should be virtual.