MFM C++ Coding Standards

Edward (Ned) Horn – July 16, 2015

# CODING REQUIREMENTS

## Code should be commented.

Comments should be concise and clear. A block should have a comment if its meaning is not obvious. Avoid repetitive comments of obvious things.

## Headers should be commented such that they act as a guide to the API.

Each public method should be commented (unless it is trivial and obvious). Ideally, the purpose, return values and method arguments should be documented.

The purpose of the component and an overall guide to the API should be given near the top of the header. Example usage code after the guide is highly encouraged.

## All temporary variables must be self-managing. All local resources in methods must be self-managing.

There should be no need for clean-up code at the end of a method. This is sometimes known as “RAII” (resource acquisition is initialization). It is accomplished by wrapping all resources in objects whose constructors and destructors manage the resource. For example, an ifstream opens a file in its constructor and closes it in its destructor. No local object should be new()-ed then delete()-ed on return. Use concrete instantiations.

For example:

void doStuff() {

Date date; //GOOD, constructed on stack, deleted when function returns

Cashflow \*flow = new Cashflow(); //BAD! If delete is not called it will leak.

std::fstream(“/tmp/foo”); //GOOD, opens file when instantiated and closes when function returns

FILE\* fp = fopen(“tmp/foo”, “+w”); //BAD! Must be closed on return

It is essential that all code do this before exceptions are turned on, since an exception may cause a function to return any time a subroutine is called!

## With rare exceptions, no new()except in a shared pointer.

## 4-space soft tab indentation.

But NOT for makefiles, must use tab for them.

## In general, always use {}s.

Simple return statements at end of methods can do without them.

if( x > 0 )

{ y = x; }

NOT

if( x > 0 )

y = x;

## Not “using std”.

In general no “using” unless there is a compelling reason. Get used to typing std::cout, std::vector, etc.

## When doing an equality comparison (), constants should be on the left of a variable.

E.g.: if(0 == X){}, NOT if (X == 0). This is to minimize the chance of accidentally setting the variable to the constant.

## 9) Naming: camel-case for classes, structs and methods, underscore for variables. d\_ (underscore) for object data members, s\_ (underscore) for static variables.

Types (such as classes and structs) begin with upper case, methods and functions with lower case.

Example:

class Something { // A type, leading letter capitalized

public:

void doSomething(); // A method, leading letter lower-case

private:

int d\_some\_thing; // A data member, d\_

static int s\_some\_thing\_else; // A static data member, s\_

}

Names should be descriptive, meaningful, but not overly-long, unless they are local variables, in which case short, simple names may be used.

## 10) In general, use good taste when coding. Remember you are writing code for others to read.

# DESIGN GOALS

## Each class should have one cohesive purpose.

Use manager classes to delegate logic to contained helper classes. Do not move implementation details for one class into another class.

## Classes should be minimal but complete.

Utility methods that can be implemented entirely in terms of the public interface should not be part of the class. Each class should provide all the methods needed to fulfill its basic functionality, but little more.

Avoid redundant methods. Do not expose implementation details in the public API.

## Headers should be uncluttered.

Each header should contain only one primary class. Iterators and other objects that encapsulate implementation can accompany it.

Helper methods, objects and classes used in the implementation should be members of a class ONLY if they MUST rely on private implementation details. Otherwise, move helper objects and functions into an anonymous name-space at file scope.

## Code should be const correct.

## Avoid cyclical dependency between classes.

No “Class A depends on class B which depends on Class A” or more complex cycles.

## Avoid monolithic functions

Break up long methods into helper functions and master logic. Each helper should have a single clear purpose, with no unexpected side-effects, and be clearly named.

# RECOMMEND READINGS

* “*API design for C++*”: Martin Reddy. Best single-volume guide to practical design issues. A must read. Be warned that it is mainly concerned with object-oriented design, not generic (template-based).
* “*C++ Coding Standards*”: Herb Sutter and Andrei Alexandrescu.
* “*The Practice of Programming*”: Brian Kernighan and Rob Pike.
* “*Large Scale C++*”: John Lakos. NOT for the faint-of-heart, but you will be a better programmer if you read it.
* “*Effective C++*”: Scott Meyers. Another must read.
* Google C++ Style Guide: <http://google-styleguide.googlecode.com/svn/trunk/cppguide.html>