C++ Coding Guidelines

Arjun Ray, 2015

# Reference

Sutter and Alexandrescu: *“C++ Coding Standards”*. This is a good collection of guidelines. It covers a lot of ground, and there are useful tips in the discussions.

# Managers and Workers

There are two kinds of functions: “manager” functions and “work” functions. Managers make decisions (apply policies) and workers execute (implement mechanism): workers do not make decisions, and managers do not execute mechanisms (tasks), except in the sense of being “workers” in relation to hierarchically superior managers. Code should be (re-)factored accordingly.

The worker functions are task oriented, and so, each function should perform exactly one clearly defined task (or *unit* of work). Doing two or more things makes the function at once useless when you need just one piece of functionality (yes, you will!), without any of the others.

Note: this is why “manager” functions typically consist of decision structures and function calls only, *dispatching* the actual work. For the same reason, a lot of the time manager functions will have no local variables other than perhaps a return value.

# Law of Demeter

Google has a nice collection of hits: <https://www.google.com/#q=law+of+demeter>

The basic idea is to avoid relying on the implementation details of other objects. In particular, “going through” one object to get to another object on which to make a method call. Or even to get to a third object in order to do something. “Rule of two dots”.

# “Tell, Don’t Ask”

Google this: [https://www.google.com/#q=”tell+don’t+ask](https://www.google.com/#q=)”

Don’t ask an object for information on the basis of which you will then command the object to do something (such as modify its state). Tell the object to update itself on the basis of the information you provide.

Our code base violates this all over the place. Objects are treated as haphazard grab bags of data elements that are then made public so that they can be worked with and modified from elsewhere, or just about anywhere except the logical and obvious place, which would be in a method of that object.

# Avoid Getters and Setters (they are evil!)

Google hits: <https://www.google.com/#q=getters+setters+bad>

This will be a touch habit to break!

The basic idea is the very first principle of encapsulation: members are part of an object’s state, and an object’s state should never be manipulated from outside the object. If you find yourself “needing” getters and/or setters, then the *design* is flawed.

Exception: intentional pod, such as in the Named Parameter Idiom (but note: no getters).

<https://isocpp.org/wiki/faq/ctors#named-parameter-idiom>

<http://www.drdobbs.com/cpp/pure-c-options-classes/224700827>

# Need to Know Principle

A function interface should have the minimal input to do its job without external calls or other dependencies. The greater the number of dependencies, the less general and the less reusable the function becomes. See the example in the attached file.

# Minimize the number of local variables

A large number of local variables can be symptomatic of several problems:

* The function is doing more than one thing or unit of work.
* Failure to modularize subtasks into subroutines and/or helper objects.

And, static locals are rarely, if ever, correct in object methods. There is *always* a better way to accomplish the goal. Quite often, those statics are better off as the members of a helper object. (And use the constructor for the “is this the first time through?” case.)

# Avoid zero-argument constructors + separate init() functions

APIs that separate creation and initialization are bad, if not very bad, because they endanger a race condition between an object’s lifetime and the invariants it’s supposed to maintain. What if someone forgot to call that public init() method? For robustness, every useful public method of that object would have to waste time too, on *every* invocation, checking to see if the object has been initialized. In effect, an object’s methods couldn’t even trust its *own* data! Which is ridiculous.

The motivation for init() functions, i.e. for the existence of uninitialized (== unusable) objects, actually comes from a fear of exception handling. (The only way to signal constructor failure is to throw an exception.)

Custom exceptions:

<http://www.boost.org/more/error_handling.html>

<http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836>

<http://www.gotw.ca/gotw/066.htm>

# Think and practice RAII

Quite seriously, if you aren’t doing RAII, you aren’t doing C++. It is Rule Number 0 of C++ programming.

Objects should not live any longer than necessary, and should be usable throughout their lifetime. Their creation should be as late as possible and their destruction should be as early as possible. In this vein: all stack allocations should be in the innermost block consistent with the required semantics.

Prefer stack allocation to heap allocation. The C++ language *guarantees* cleanup of objects allocated on the stack when they go out of scope, regardless of whether this is due to normal flow of control or due to an exception being thrown.

Heap allocations should be through smart pointers only (preferably, use std::make\_shared). When the smart pointer is instantiated on the stack, RAII will take care of releasing the underlying heap allocation.

With return codes, use UROE, not UCOE. Make the clean\_up: goto target redundant – there should be no “cleanup code” section there, because RAII will be taking care of the issue(s) in the main line. In particular, std::ostream’s do not need to be explicitly .closed(). (Why? Because those objects are themselves RAII-compliant!!)

Recommend usage of smart pointers:

<http://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/>

# No Raw Loops

See Sean Parent’s talk (1hr 17 mins): <https://www.youtube.com/watch?v=qH6sSOr-yk8> (slides) <https://github.com/sean-parent/sean-parent...seasoning/cpp-seasoning.pdf>

This talk prompts a number of ideas:

* Any non-trivial loop-body is a code block. A code block can become a function block, which enables unit tests and re-use.
* It is generally useful to disengage the mechanics of traversing a container from what is to be done with each element.
* Use STL algorithms – they are tested, proven and designed to minimize bugs (e.g. off-by-one errors).
* An object aggregating a container should provide a “travers()” or “apply()” or “iterate()” functional interface to pass each element of the collection to that function. Thus, on one side you have a pure traversing interface and on the other a function implementing business logic independent of how the object is stored or accessed.
* Containers (mostly, vectors and maps) should be composed/wrapped inside objects, and hidden behind interfaces: don’t hand out the container for some external code to traverse.

How to iterate when you must:

* Using loop indexes is thinking in Fortran (definitely do not do this).
* Using pointers is thinking in C (this is passable).
* Using iterator is thinking in old-fashioned C++ (this is better).
* Using STL algorithms and/or ranged-for is the modern way to practice C++ (this is best).

Other ideas:

1. Delete dead code! It’s just clutter and slows everything down, including trouble shooting.
2. Comment more often, please!
3. Ned Horn is writing up a guideline document. Heeding it will help in the longer haul. Maybe a little pain now to save a lot of pain down the road.