[25] Coding standards

(Part of *C++ FAQ Lite*, Copyright © 1991-2000, Marshall Cline, cline@parashift.com)

FAQs in section [25]:

[25.1] What are some good C++ coding standards?

Thank you for reading this answer rather than just trying to set your own coding standards.

But beware that some people on `comp.lang.c++` are very sensitive on this issue. Nearly every software engineer has, at some point, been exploited by someone who used coding standards as a "power play." Furthermore some attempts to set C++ coding standards have been made by those who didn't know what they were talking about, so the standards end up being based on what *was* the state-of-the-art when the standards setters where writing code. Such impositions generate an attitude of mistrust for coding standards.

Obviously anyone who asks this question wants to be trained so they *don't* run off on their own ignorance, but nonetheless posting a question such as this one to `comp.lang.c++` tends to generate more heat than light.

[Top | Bottom | Previous section | Next section]

[25.2] Are coding standards necessary? Are they sufficient?

Coding standards do not make non-OO programmers into OO programmers; only training and experience do that. If coding standards have merit, it is that they discourage the petty fragmentation that occurs when large organizations coordinate the activities of diverse groups of programmers.

But you really want more than a coding standard. The structure provided by coding standards gives neophytes one less degree of freedom to worry about, which is good. However pragmatic guidelines should go well beyond pretty-printing standards. Organizations need a consistent *philosophy* of design and implementation. E.g., strong or weak typing? references or pointers in interfaces? stream I/O or stdio? should C++ code call C code? vice versa? how should ABCs be used? should inheritance be used as an implementation technique or as a specification technique? what testing strategy should be employed? inspection strategy? should interfaces uniformly have a `get()` and/or `set()` member function for each data member? should interfaces be designed from the outside-in or the inside-out? should errors be handled by `try`/`catch`/`throw` or by return codes? etc.

What is needed is a "pseudo standard" for detailed *design.* I recommend a three-pronged approach to achieving this standardization: training, mentoring, and libraries. Training provides "intense instruction," mentoring allows OO to be caught rather than just taught, and high quality C++ class libraries provide "long term instruction." There is a thriving commercial market for all three kinds of "training." Advice by organizations who have been through the mill is consistent: *Buy, Don't Build.* Buy libraries, buy training, buy tools, buy consulting. Companies who have attempted to become a self-taught tool-shop as well as an application/system shop have found success difficult.

Few argue that coding standards are "ideal," or even "good," however they are necessary in the kind of organizations/situations described above.

The following FAQs provide some basic guidance in conventions and styles.

[Top | Bottom | Previous section | Next section]

[25.3] Should our organization determine coding standards from our C experience?

No!

No matter how vast your C experience, no matter how advanced your C expertise, being a good C programmer does not make you a good C++ programmer. Converting from C to C++ is more than just learning the syntax and semantics of the `++` part of C++. Organizations who want the promise of OO, but who fail to put the "OO" into "OO programming", are fooling themselves; the balance sheet will show their folly.

C++ coding standards should be tempered by C++ experts. Asking `comp.lang.c++` is a start. Seek out experts who can help guide you away from pitfalls. Get training. Buy libraries and see if "good" libraries pass your coding standards. Do *not* set standards by yourself unless you have considerable experience in C++. Having no standard is better than having a bad standard, since improper "official" positions "harden" bad brain traces. There is a thriving market for both C++ training and libraries from which to pull expertise.

One more thing: whenever something is in demand, the potential for charlatans increases. Look before you leap. Also ask for student-reviews from past companies, since not even expertise makes someone a good communicator. Finally, select a practitioner who can teach, not a full time teacher who has a passing knowledge of the language/paradigm.

[Top | Bottom | Previous section | Next section]

[25.4] What's the difference between <xxx> and <xxx.h> headers? UPDATED

[Recently changed `<xyz.h>` to `<xxx.h>` and misc wordsmithing thanks to Stan Brown (on 7/00). Click here to go to the next FAQ in the "chain" of recent changes.]

The headers in ISO Standard C++ don't have a `.h` suffix. This is something the standards committee changed from former practice. The details are different between headers that existed in C and those that are specific to C++.

The C++ standard library is guaranteed to have 18 standard headers from the C language. These headers come in two standard flavors, `<cxxx>` and `<xxx.h>` (where `xxx` is the basename of the header, such as `stdio`, `stdlib`, etc). These two flavors are identical except the `<cxxx>` versions provide their declarations in the `std` namespace only, and the `<xxx.h>` versions make them available both in `std` namespace and in the global namespace. The committee did it this way so that existing C code could continue to be compiled in C++. However the `<xxx.h>` versions are deprecated, meaning they are standard now but might not be part of the standard in future revisions. (See clause D.5 of the ISO C++ standard.)

The C++ standard library is also guaranteed to have 32 additional standard headers that have no direct counterparts in C, such as `<iostream>`, `<string>`, and `<new>`. You may see things like `#include <iostream.h>` and so on in old code, and some compiler vendors offer `.h` versions for that reason. But be careful: the `.h` versions, if available, may differ from the standard versions. And if you compile some units of a program with, for example, `<iostream>` and others with `<iostream.h>`, the program may not work.

For new projects, use only the `<xxx>` headers, not the `<xxx.h>` headers.

When modifying or extending existing code that uses the old header names, you should probably follow the practice in that code unless there's some important reason to switch to the standard headers (such as a facility available in standard `<iostream>` that was not available in the vendor's `<iostream.h>`). If you need to standardize existing code, make sure to change all C++ headers in all program units including external libraries that get linked in to the final executable.

All of this affects the standard headers only. You're free to name your own headers anything you like; see [25.8].

[Top | Bottom | Previous section | Next section]

[25.5] Is the ?: operator evil since it can be used to create unreadable code? UPDATED

[Recently changed so it uses new-style headers and the `std::` syntax (on 7/00). Click here to go to the next FAQ in the "chain" of recent changes.]

No, but as always, remember that readability is one of the most important things.

Some people feel the `?:` ternary operator should be avoided because they find it confusing at times compared to the good old `if` statement. In many cases `?:` tends to make your code more difficult to read (and therefore you should replace those usages of `?:` with `if` statements), but there are times when the `?:` operator is clearer since it can emphasize what's really happening, rather than the fact that there's an `if` in there somewhere.

Let's start with a really simple case. Suppose you need to print the result of a function call. In that case you should put the real goal (printing) at the beginning of the line, and bury the function call within the line since it's relatively incidental (this left-right thing is based on the intuitive notion that most developers think the first thing on a line is the most important thing):

```
// Preferred (emphasizes the major goal — printing):
std::cout << funct();

// Not as good (emphasizes the minor goal — a function call):
functAndPrintOn(std::cout);
```

Now let's extend this idea to the `?:` operator. Suppose your real goal is to print something, but you need to do some incidental decision logic to figure out what should be printed. Since the printing is the most important thing conceptually, we prefer to put it first on the line, and we prefer to bury the incidental decision logic. In the example code below, variable `n` represents the number of senders of a message; the message itself is being printed to `std::cout`:

```
int n = /*...*/;      // number of senders

// Preferred (emphasizes the major goal — printing):
std::cout << "Please get back to " << (n==1 ? "me" : "us") << " soon!\n";

// Not as good (emphasizes the minor goal — a decision):
std::cout << "Please get back to ";
if (n==1)
  std::cout << "me";
else
  std::cout << "us";
std::cout << " soon!\n";
```

All that being said, you can get pretty outrageous and unreadable code ("write only code") using various combinations of `?:`, `&&`, `||`, etc. For example,

```
// Preferred (obvious meaning):
if (f())
  g();

// Not as good (harder to understand):
f() && g();
```

Personally I think the explicit `if` example is clearer since it emphasizes the major thing that's going on (a decision based on the result of calling `f()`) rather than the minor thing (calling `f()`). In other words, the use of `if` here is *good* for precisely the same reason that it was *bad* above: we want to major on the majors and minor on the minors.

In any event, don't forget that readability is the goal (at least it's one of the goals). Your goal should *not* be to avoid certain syntactic constructs such as `?:` or `&&` or `||` or `if` — or even `goto`. If you sink to the level of a "Standards Bigot," you'll ultimately embarass yourself since there are always counterexamples to any syntax-based rule. If on the other hand you emphasize broad goals and guidelines (e.g., "major on the majors," or "put the most important thing first on the line," or even "make sure your code is obvious and readable"), you're usually much better off.

Code must be written to be read, not by the compiler, but by another human being.

[Top | Bottom | Previous section | Next section]

[25.6] Should I declare locals in the middle of a function or at the top?

Declare near first use.

An object is initialized (constructed) the moment it is declared. If you don't have enough information to initialize an object until half way down the function, you should create it half way down the function when it can be initialized correctly. Don't initialize it to an "empty" value at the top then "assign" it later. The reason for this is runtime performance. Building an object correctly is faster than building it incorrectly and remodeling it later. Simple examples show a factor of 350% speed hit for simple classes like `String`. Your mileage may vary; surely the overall system degradation will be less that 350%, but there *will* be degradation. *Unnecessary* degradation.

A common retort to the above is: "we'll provide `set()` member functions for every datum in our objects so the cost of construction will be spread out." This is worse than the performance overhead, since now you're introducing a maintenance nightmare. Providing a `set()` member function for every datum is tantamount to `public` data: you've exposed your implementation technique to the world. The only thing you've hidden is the physical *names* of your member objects, but the fact that you're using a `List` and a `String` and a `float`, for example, is open for all to see.

Bottom line: Locals should be declared near their first use. Sorry that this isn't familiar to C experts, but new doesn't necessarily mean bad.

[Top | Bottom | Previous section | Next section]

[25.7] What source-file-name convention is best? `foo.cpp`? `foo.C`? `foo.cc`?

If you already have a convention, use it. If not, consult your compiler to see what the compiler expects. Typical answers are: `.C`, `.cc`, `.cpp`, or `.cxx` (naturally the `.C` extension assumes a case-sensitive file system to distinguish `.C` from `.c`).

We've often used both `.cpp` for our C++ source files, and we have also used `.C`. In the latter case, we supply the compiler option forces .c files to

be treated as C++ source files (`-Tdp` for IBM CSet++, `-cpp` for Zortech C++, `-P` for Borland C++, etc.) when porting to case-insensitive file systems. None of these approaches have any striking technical superiority to the others; we generally use whichever technique is preferred by our customer (again, these issues are dominated by business considerations, not by technical considerations).

[Top | Bottom | Previous section | Next section]

[25.8] What header-file-name convention is best? `foo.H`? `foo.hh`? `foo.hpp`?

If you already have a convention, use it. If not, and if you don't need your editor to distinguish between C and C++ files, simply use `.h`. Otherwise use whatever the editor wants, such as `.H`, `.hh`, or `.hpp`.

We've tended to use either `.hpp` or `.h` for our C++ header files.

[Top | Bottom | Previous section | Next section]

[25.9] Are there any lint-like guidelines for C++? UPDATED

[Recently changed so it uses new-style headers and the `std::` syntax (on 7/00). Click here to go to the next FAQ in the "chain" of recent changes.]

Yes, there are some practices which are generally considered dangerous. However none of these are universally "bad," since situations arise when even the worst of these is needed:

- A `class Fred`'s assignment `operator` should return `*this` as a `Fred&` (allows chaining of assignments)
- A class with any `virtual` functions ought to have a virtual destructor
- A class with any of {destructor, assignment `operator`, copy constructor} generally needs all 3
- A `class Fred`'s copy constructor and assignment `operator` should have `const` in the parameter: respectively `Fred::Fred(const Fred&)` and `Fred& Fred::operator= (const Fred&)`
- When initializing an object's member objects in the constructor, always use initialization lists rather than assignment. The performance difference for user-defined classes can be substantial (3x!)
- Assignment operators should make sure that self assignment does nothing, otherwise you may have a disaster. In some cases, this may require you to add an explicit test to your assignment operators.
- In classes that define both `+=` and `+`, `a += b` and `a = a + b` should generally do the same thing; ditto for the other identities of built-in types (e.g., `a += 1` and `++a`; `p[i]` and `*(p+i)`; etc). This can be enforced by writing the binary operations using the `op=` forms. E.g.,

```
Fred operator+ (const Fred& a, const Fred& b)
{
  Fred ans = a;
  ans += b;
  return ans;
}
```

  This way the "constructive" binary operators don't even need to be friends. But it is sometimes possible to more efficiently implement common operations (e.g., if `class Fred` is actually `std::string`, and `+=` has to reallocate/copy string memory, it may be better to know the eventual length from the beginning).

[Top | Bottom | Previous section | Next section]

[25.10] Which is better: identifier names `that_look_like_this` or identifier names `thatLookLikeThis`? UPDATED

[Recently improved the precision and added the last two paragraphs thanks to Chris Hurst (on 7/00). Click here to go to the next FAQ in the "chain" of recent changes.]

It's a precedent thing. If you have a Pascal or Smalltalk background, `youProbablySquashNamesTogether` like this. If you have an Ada background, `You_Probably_Use_A_Large_Number_Of_Underscores` like this. If you have a Microsoft Windows background, you probably prefer the "Hungarian" style which means you `jkuidsPrefix vndskaIdentifiers ncqWith ksldjfTheir nmdsadType`. And then there are the folks with a Unix C background, who `abbr evthng n use vry srt idntfr nms`. (`AND THE FORTRN PRGMRS LIMIT EVRYTH TO SIX LETTRS`.)

So there is no universal standard. If your project team has a particular coding standard for identifier names, use it. But starting another Jihad over this will create a lot more heat than light. From a business perspective, there are only two things that matter: The code should be generally readable, and everyone on the team should use the same style.

Other than that, th difs r minr.

One more thing: don't import a coding style onto platform-specific code where it is foreign. For example, a coding style that seems natural while using a Microsoft library might look bizarre and random while using a UNIX library. Don't do it. Allow different styles for different platforms. (Just in case someone out there isn't reading carefully, don't send me email about the case of common code that is designed to be used/ported to several platforms, since that code wouldn't be platform-specific, so the above "allow different styles" guideline doesn't even apply.)

Okay, one more. Really. Don't fight tools that generate code. Some people treat coding standards with religious zeal, and they try to get tools to generate code in their local style. Forget it: if a tool generates code in a different style, don't worry about it. Remember money and time?!? This whole coding standard thing was supposed to *save* money and time; don't turn it into a "money pit."

[Top | Bottom | Previous section | Next section]

[25.11] Are there any other sources of coding standards? UPDATED

[Recently fixed some URLs thanks to James S. Adelman and Stan Brown (on 7/00). Click here to go to the next FAQ in the "chain" of recent changes.]

Yep, there are several.

Here are a few sources that you might be able to use as starting points for developing your organization's coding standards:

- www.cs.princeton.edu/~dwallach/CPlusPlusStyle.html
- cpptips.hyperformix.com/conventions/cppconventions_1.html
- www.oma.com/ottinger/Naming.html
- v2ma09.gsfc.nasa.gov/coding_standards.html
- fndaub.fnal.gov:8000/standards/standards.html
- cliffie.nosc.mil/~NAPDOC/docprj/cppcodingstd/
- www.possibility.com/cpp/
- groucho.gsfc.nasa.gov/Code_520/Code_522/Projects/DRSL/documents/templates/cpp_style_guide.html
- www.wildfire.com/~ag/Engineering/Development/C++Style/
- The Ellemtel coding guidelines are available at
  - www.cs.umd.edu/users/cml/cstyle/Ellemtel-rules.html
  - www.doc.ic.ac.uk/lab/cplus/c++.rules/
  - www.mgl.co.uk/people/kirit/cpprules.html

Note: I do *NOT* warrant or endorse these URLs and/or their contents. They are listed as a public service only. I haven't checked their details, so I don't know if they'll help you or hurt you. Caveat emptor.

[Top | Bottom | Previous section | Next section]

E-mail the author

[*C++ FAQ Lite* | Table of contents | Subject index | About the author | © | Download your own copy]

Revised Jul 10, 2000