# Code Quality Guidelines

Defining exactly what constitutes good code quality is difficult and very subjective. Also, it is far easier to detect poor quality than good quality. It may take a while to discover that the code you are studying is actually of high-quality, at least until you've studied a large amount of code. However, it usually takes less than 10 seconds to determine that the code you are looking at is really bad. There are a few points about code quality that most developers agree on. This is not an exhaustive list, but it covers most of the issues that new programmers run into.

**Code reuse** – Probably one of the worst advances in text processing for the programmer has been the "copy and paste" feature. Although it works quite well, it is actually counter-productive to repeat identical sections of code and is a short-term solution at best. The better solution is to locate the common code blocks and create a function that can be called on-demand (reused). Sometimes the code isn't quite identical, but is very similar. In this case, parameterizing the function is the correct approach.

**Magic Numbers** – Having seemingly "random" numbers scattered throughout the code is difficult to maintain. It is better to use some sort of #define, enum, or const declaration to hold the values. It also can lead to self-documenting code.

**Redundant or meaningless code** – Simply put, code that has no end effect on the program. For example, a conditional statement that checks to see if a value is less than 0. Then within the body of the conditional, the unchanged value is checked to see if it is less than 0. Unnecessary and hard to maintain. It's not so much that the time required to perform the check at runtime is unacceptable, but rather, additional time is required by another programmer wondering why the value is checked again and hunting around for the place where it would have changed. Other examples would be creating a non-virtual destructor that has an empty body or checking if an unsigned integer value is less than zero. (Some compilers may even warn about the latter since it can never be true.)

**Scalability** – If any code you write may, at some time, be used within a larger program, writing it so that it can be used again requires some thought. Maybe not much thought, but some additional thought is required. The linear algorithm you write to find a value in a sorted list of 10 values might be pathetically slow when the list contains millions of values.

**Reinventing the wheel** – Don't write functions that already exist in the Standard C/C++ library. An example is **strcpy**. Don't write code that copies the contents of one string to another. Use the library. This is a form of code reuse. Not only is the code available, most of the time it is of the highest quality and has been tested and debugged a million times over.

**Performance** – This can be a component of code quality. For example, say you need to fill a buffer with different data and process it many times using a loop of some sort. Suppose you need to dynamically allocate the buffer. Allocating and deallocating it over and over again in the loop is certainly going to affect performance. The better approach is to allocate it once before the loop, and then deallocate it after the loop. Or, something trivial like this is a problem:

```
for (i = 0; i < strlen(text); i++)
  /* do something with text */
```

In C/C++, **strlen** uses a very inefficient linear-time algorithm, which is especially noticeable if the length of the string is large. (Other languages implement strings more efficiently.) Note that in this context, performance is on a more coarse level. I'm not talking about micro-optimizing (e.g. code such as using shift operators for multiplying an integer by 2). Obviously, a better solution to the one above is something like this:

```
int length = strlen(text);
for (i = 0; i < length; i++)
  /* do something with text */
```

**const –** Use **const** wherever it makes sense. The **const** keyword is probably one of the must underutilized keywords. Even if your program "runs correctly", failing to use **const** will make it harder to use the code elsewhere, which is the whole idea behind Code Reuse.

**Descriptive Variable Names** – This, of course, can be very subjective. And, like quality code, it's easy to spot poorly named variables:

```
int CopyFile(const char *Source, const char *Destination)
{
  FILE* temp1 = fopen(Source, "rb");        // temp1 ???
  FILE* temp2 = fopen(Destination, "wb");   // temp2 ???
  ...
}
```

Better names would be:

```
int CopyFile(const char *Source, const char *Destination)
{
  FILE* input_file = fopen(Source, "rb");
  FILE* output_file = fopen(Destination, "wb");
  ...
}
```

**Keep Implementation Details Private** – If you need some helper functions for your code, don't "advertise" them in the header file. Put them in the implementation file only. For C code, mark them with the **static** keyword. In C++, you should put them in the **private** section of the class, or, if you don't want them to be part of the class, put them in an unnamed namespace in the implementation file.

**Enable Maximum Compiler Warnings** – Make sure to build your code with all (or more than all) warnings enabled. If you have code that is "safe" but still generates a warning, change your code so it doesn't emit a warning. Don't use **#pragma**s to disable warnings unless instructed to do so.

**Never Submit/Check-in Code with Warnings –** This is related to the previous rule. The compiler identifies each location in your code that is suspicious, so you hardly have to think at all. Fix the code before proceeding. As a corollary, run, don't walk, from a team that allows its members to check in code with warnings. You've got better things to do than to examine all of your teammates' code for suspicious behavior every time you check out their work.

**Avoid Global Variables –** There are times that global variables are needed. However, those times never occur within any assignment I give. Think of global variables as an advanced feature disguised as a newbie tool (much like "goto" statements.) Along these lines, know the difference between global symbols and file scope symbols. File scope symbols are not the same as globals, but their use should be warranted. In other words, use them when you can't cleanly implement the solution without them.

**Disorganized Code –** Any code that lacks consistent formatting is very difficult to read. It is usually a symptom of a bigger problem: the programmer doesn't know what he's doing and is trying to figure it out while typing. Don't do this. If you can't describe your thought process (solution) on paper in English, you are not ready to be sitting in front of the code editor. Also, adding tons of comments around this kind of code doesn't make it any better. In fact, if you find you have to write a lot of comments to understand your own code, your code is probably the problem.

This is just a partial list of all of the things “quality” related. If you have doubts about your code, don't be afraid to ask.