Optimal code writing in C++

Gulyamova Dilfuza Rahmatullaevna
Tashkent University of Informational Technologies, senior teacher

Maxammadjonov Maxammadjon Alisher o'g'li
Tashkent University of Informational Technologies, student

E-mail: mr.mahammadjon@mail.ru

Programming technology develops. As a result, new programs are being developed. The development of programs written in code in different programming languages. This procedure codes to write their own. For example, based on the C ++ language features, it has the potential to create a number of programs. Follow the rules of writing software codes are helping to create incredible programs. This article is dedicated to write optimal code.

What follows are 10 of the more important things to keep in mind if you want to write polished, professional C++ code that is easy to maintain and less likely to need debugging.

These guidelines are in no particular order except that the earlier items are addressed more to mistakes that beginners have trouble with.

1. Don't Confuse Assign (=) with Test-for-Equality (==).

This one is elementary, although it might have baffled Sherlock Holmes. The following looks innocent and would compile and run just fine if C++ were more like BASIC:

```
if (h = q)

cout << "h is equal to q.";
```

Because this looks so innocent, it creates logic errors requiring hours to track down within a large program unless you're on the lookout for it. In C and C++, the following is not a test for equality: h = q. What this does, of course, is assign the value of b to a and then evaluate to the value assigned. The problem is that a = b does not generally evaluate to a reasonable true/false condition—with one major exception I'll mention later. But in C and C++, any numeric value can be used as a condition for "if" or "while.

Assume that a and b are set to 0. The effect of the previously-shown if statement is to place the value of b into h; then the expression h = q evaluates to 0. The value 0 equates to false. Consequently, h and q are equal, but exactly the wrong thing gets printed:

```
if (h = q) cout << "h and q are equal.";

else cout << "h and q are not equal.";
```

The solution, of course, is to use test-for-equality when that's what you want. Note the use of double equal signs (==). This is correct inside a condition.

```
if (h == q) cout << "h and q are equal.";
```

2. Do Use Data Promotion to Control Results

In mixed integer/floating-point expressions, integer operands are promoted to type double. Consequently, this expression produces what we want:

```
cout << results / (n / 10.0);
```

Note that 10.0, though having no fraction, is stored as type double; this causes C++ to promote n and results to type double as well, and then carries out floating-point division.

Other ways to produce this effect include use of data casts:

```
cout << results / (n / (double) 10);
```

```
cout << results / (n / static_cast<double>(10));
```

3. Don’t Use Non-Boolean Conditions (Except with Care)

Designed originally to help write operating systems, the C language was meant to give programmers freedom—not only to manipulate data at a machine-code level (through the use of pointers) but also to write shortcuts. Shortcuts are dangerous for beginners but sometimes nice for programmers who know what they’re doing. If they’re willing to live dangerously. One of the most elegant tricks is the following, a slick way of saying “Do something N times:”

```
while (n—) { do_something();}
```

You can make this even shorter:

```
while (n—) do_something();
```

4. Don’t Use Global Variables Except to Communicate Between Functions

Only a few words need be said about this one. Programmers sometimes ask themselves whether to make a variable local or global. The rule is simple: If a variable is used to store information that communicates between functions, either make the variable into a parameter (part of an argument list), passing the information explicitly, or make it global.

When information is to be shared between several functions, it’s often most practical to just go with global variables. In syntactic terms, this means declaring them outside all function definitions.

The reason for using relatively few variables global is clear: With global variables, the internal actions of one function can interfere with the internal actions of other functions—often unintentionally, especially in a large program. So try to make most variables local if you can. Enough said.

5. Do Use Local Variables with the for Statement

With all but the most ancient versions of C++, the slickest way to localize a variable is to declare a loop variable inside of a for statement. This ability was not always supported, so old-timers like me sometimes need reminding that you can do this.

```
for (int i = 1; i <= n; i++) { // Declare i inside the loop.
cout << i << " "; // Print out numbers 1 to n.
}
```

In the case of for loops, it’s rare that the final value of i, the loop variable, will be used later in the program (although that does happen occasionally). Generally speaking, you’re going to want to use a variable such as i for a loop counter and then throw it away when you’re done. Making i local to the loop itself is not only a slick way to save some space, but it’s safer programming.

One of the benefits of this use of for is that it automatically takes care of having to initialize i, a local variable. Yes, it can be extra work, but initializing a local variable is something you ideally ought to do—although it is tempting to leave it uninitialized if the very first statement in the function is going to set the variable’s value. Nonetheless, the most correct programs tend to initialize their locals:

```
void do_stuff() {
int n = 0; // Number of repetitions
```

int i = 0; // Loop counter.

Remember that initialized global variables (including objects) contain all-zero values, while uninitialized locals (including objects) contain garbage—garbage being a technical term for “meaningless, useless value that is likely to blow up your program.”

Now we can state the full syntax rule in C/C++:

1. Terminate each statement with a semicolon.

2. Don’t follow a closing brace with a semicolon unless it ends a class declaration, in which case it’s required.

Programming in C++ (or for that matter, in any language) is a life-long pursuit, and you never stop learning. In this article, I’ve looked at just the tip of the iceberg; however, in my programming experience going all the way back to the 1980s and even earlier, the issues in these article are points that, for me at least, come up again and again.

Among the more important ideas: Use the right operator for the right task (don’t confuse = and ==); pay attention to the effect of data types in math operations; be extra-careful about tempting shortcuts, such as while(n—); use local variables within for loops; and use meaningful symbolic names as much as possible. To be honest, I sometimes cut corners myself for very simple programs, but when you get into complex programming, following these guidelines will save you a lot of headache—if not heartache!

References

1. “Modern abilities of programming language C++”, “Научная перспектива” 2015
2. Top 15 best practices for writing super readable code https://code.tutsplus.com /tutorials/top-15-best-practices-for-writing-super-readable-code—net-8118