4.2a — Why global variables are evil

BY ALEX ON MARCH 23RD, 2015 | LAST MODIFIED BY ALEX ON FEBRUARY 14TH, 2017

If you were to ask a veteran programmer for *one* piece of advice on good programming practices, after some thought, the most likely answer would be, “Avoid global variables!”. And with good reason: global variables are one of the most abused concepts in the language. Although they may seem harmless in small academic programs, they are often hugely problematic in larger ones.

New programmers are often tempted to use lots of global variables, because they are easy to work with, especially when many functions are involved (passing stuff through function parameters is a pain). However, this is generally a bad idea. Many developers believe non-const global variables should be avoided completely!

But before we go into why, we should make a clarification. When developers tell you that global variables are evil, they’re not talking about ALL global variables. They’re mostly talking about non-const global variables.

Why (non-const) global variables are evil

By far the biggest reason non-const global variables are dangerous is because their values can be changed by *any* function that is called, and there is no easy way for the programmer to know that this will happen. Consider the following program:

```
// declare global variable
int g_mode;

void doSomething()
{
    g_mode = 2; // set the global g_mode variable to 2
}

int main()
{
    g_mode = 1; // note: this sets the global g_mode variable to 1.  It does not declare a local g_mode variable!

    doSomething();

    // Programmer still expects g_mode to be 1
    // But doSomething changed it to 2!

    if (g_mode == 1)
        std::cout << "No threat detected.\n";
    else
        std::cout << "Launching nuclear missiles...\n";

    return 0;
}
```

Note that the programmer set g_mode to 1, and then called doSomething(). Unless the programmer had explicit knowledge that doSomething() was going to change the value of g_mode, he or she was probably not expecting doSomething() to change the value! Consequently, the rest of main() doesn’t work like the programmer expects (and the world is obliterated).

Non-const global variables make every function call potentially dangerous, and the programmer has no easy way of knowing which ones are dangerous and which ones aren’t! Local variables are much safer because other functions can not affect them directly.

There are plenty of other good reasons not to use non-const globals.

With global variables, it’s not uncommon to find a piece of code that looks like this:

```
void foo()
{
    // useful code

    if (g_mode == 4) // do something good
}
```

Your program is broken because g_mode is set to 3, not 4. How do you fix it? Now you need to find all of the places g_mode could possibly get set to 3, and trace through how it got set in the first place. It's possible this may be in a totally unrelated piece of code!

One of the reasons to declare local variables as close to where they are used as possible is because doing so minimizes the amount of code you need to look through to understand what the variable does. Global variables are at the opposite end of the spectrum -- because they can be used anywhere, you might have to look through a significant amount of code to understand their usage.

For example, you might find g_mode is referenced 442 times in your program. Unless g_mode is well documented, you'll potentially have to look through every use of g_mode to understand how it's being used in different cases, what its valid values are, and what its overall function is.

Global variables also make your program less modular and less flexible. A function that utilizes nothing but its parameters and has no side effects is perfectly modular. Modularity helps both in understanding what a program does, as well as with reusability. Global variables reduce modularity significantly.

In particular, avoid using global variables for important "decision-point" variables (e.g. variables you'd use in a conditional statement, like variable g_mode in the example above). Your program isn't likely to break if a global variable holding an informational value changes (e.g. like the user's name). It is much more likely to break if you change a global variable that impacts how your program operates.

Rule: Use local variables instead of global variables whenever reasonable, and pass them to the functions that need them.

So what are very good reasons to use non-const global variables?

There aren't many. In many cases, there are other ways to solve the problem that avoids the use of non-const global variables. But in some cases, judicious use of non-const global variables *can* actually reduce program complexity, and in these rare cases, their use may be better than the alternatives.

For example, if your program uses a database to read and write data, it may make sense to define the database globally, because it could be needed from anywhere. Similarly, if your program has an error log (or debug log) where you can dump error (or debug) information, it probably makes sense to define that globally, because you're mostly likely to only have one log and it could be used anywhere. A sound library would be another good example: you probably don't want to pass this to every function that needs it. Since you'll probably only have one sound library managing all of your sounds, it may be better to declare it globally, initialize it at program launch, and then treat it as read-only thereafter.

Protecting yourself from global destruction

If you do find a good use for a non-const global variable, a few useful bits of advice will minimize the amount of trouble you can get into.

First, prefix all your global variables with "g_", and/or put them in a namespace, both to reduce the chance of naming collisions and raise awareness that a variable is global.

For example, instead of:

```cpp
double gravity (9.8); // unclear if this is a local or global variable from the name

int main()
{
    return 0;
}
```

Do this:

```cpp
double g_gravity (9.8); // now clear this is a global variable from the name

int main()
{
    return 0;
}
```

Second, instead of allowing direct access to the global variable, it's a better practice to "encapsulate" the variable. First, make the variable static, so it can only be accessed directly in the file it's declared. Second, provide external global "access functions" to work with the variable. These functions can ensure proper usage is maintained (e.g. do input validation, range checking, etc…). Also, if you ever decide to change the underlying implementation (e.g. move from one database to another), you only have to update the access functions instead of every piece of code that uses the global variable directly.

For example, instead of:

```
double g_gravity (9.8); // can be exported and used directly in any file
```

Do this:

```
static double g_gravity (9.8); // restrict direct access to this file only
double getGravity() // this function can be exported to other files to access the global outside of this file
{
    // We could add logic here if needed later
    return g_gravity;
}
```

Third, when writing a standalone function that uses the global variable, don't use the variable directly in your function body. Pass it in as a parameter, and use the parameter. That way, if your function ever needs to use a different value for some circumstance, you can simply vary the parameter. This helps maintain modularity.

Instead of:

```
// This function is only useful for calculating your instant velocity based on the global gravity
double instantVelocity(int time)
{
    return g_gravity * time;
}
```

Do this:

```
// This function can calculate the instant velocity for any gravity value (more useful)
// pass in the return value from getGravity() for parameter gravity if you want to use the global gravity
double instantVelocity(int time, double gravity)
{
    return gravity * time;
}
```

Finally, changing the value of a global variable is the thing that is most likely to cause problems. Structure your code to assume a global variable's value may change. Try to minimize the number of places where you change your global's value -- treat the global as read-only as much as possible. If you can set your global's value at program startup and then not change it afterward, you'll minimize the chance of unexpected issues occurring.

A joke

What's the best naming prefix for a global variable?

Answer: //

C++ jokes are the best.

Summary

Avoid use of non-const global variables if at all possible! If you do have to use them, use them sensibly and cautiously.

Const global variables (symbolic constants) are fine to use, so long as you use proper naming conventions.

In other cases, favor local variables. Pass those local variables to the functions that need them.

4.3 -- Static duration variables