6.9 — Why global variables are evil

BY ALEX ON MARCH 23RD, 2015 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 18TH, 2020

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 historically abused concepts in the language. Although they may seem harmless in small academic programs, they are often 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 calls to different functions are involved (passing data 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 usually 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:

```
int g_mode; // declare global variable (will be zero-initialized by default)

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 loc

    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 variable 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).

In short, global variables make the program's state unpredictable. Every function call becomes 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 someFunction()
{
    // useful code

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

After debugging, you determine that your program isn't working correctly because g_mode has value 3, not *4*. How do you fix it? Now you need to find all of the places g_mode could possibly be 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 key 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 accessed anywhere, you might have to look through the entire program to understand their usage. In small programs, this might not be an issue. In large ones, it will be.

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 actually functions.

Best practice

Use local variables instead of global variables whenever possible.

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

There aren't many. In most 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.

A good example is a log file, where you can dump error or debug information. It probably makes sense to define this as a global, because you're likely to only have one log in a program and it will likely be used everywhere in your program.

For what it's worth, the std::cout and std::cin objects are implemented as global variables (inside the *std* namespace).

As a rule of thumb, any use of a global variable should meet at least the following two criteria: There should only ever be one of the thing the variable represents in your program, and its use should be ubiquitous throughout your program.

Many new programmers make the mistake of thinking that something can be implemented as a global because only one is needed *right now*. For example, you might think that because you're implementing a single player

game, you only need one player. But what happens later when you want to add a multiplayer mode (versus or hotseat)?

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 non-namespaced global variables with "g" or "g_", or better yet, put them in a namespace (discussed in lesson **6.2 -- User-defined namespaces**), to reduce the chance of naming collisions.

For example, instead of:

```
constexpr double gravity { 9.8 }; // unclear if this is a local or global variable from the nam

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

Do this:

```
namespace constants
{
    constexpr double gravity { 9.8 };
}

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 in which it is 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:

```
namespace constants
{
    extern const double gravity { 9.8 }; // has external linkage, is directly accessible by oth
}
```

Do this:

```
namespace constants
{
    static const double gravity { 9.8 }; // has internal linkage, is accessible only by this f
}

double getGravity() // this function can be exported to other files to access the global outsi
{
    // We could add logic here if needed later
    // or change the implementation transparently to the callers
    return constants::gravity;
}
```

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

Instead of:

```
#include <iostream>

namespace constants
{
    constexpr double gravity { 9.8 };
}

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

int main()
{
    std::cout << instantVelocity(5);
}
```

Do this:

```
#include <iostream>

namespace constants
{
    constexpr double gravity { 9.8 };
}

// This function can calculate the instant velocity for any gravity value (more useful)
double instantVelocity(int time, double gravity)
{
    return gravity * time;
}

int main()
{
    std::cout << instantVelocity(5, constants::gravity); // pass our constant to the function
}
```

A joke

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

Answer: //

C++ jokes are the best.

6.10 -- Static local variables

Index

6.8 -- Global constants and inline variables

C++ TUTORIAL | PRINT THIS POST

141 comments to 6.9 — Why global variables are evil

« Older Comments 1 2 3

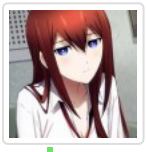
Nikola
January 18, 2020 at 6:37 am · Reply

Hey guys. Still going through the tutorials when I get the time to do so. Noticed a mistake - minor and irrelevant to any examples - but I thought you might want to fix it.

Section "So what are very good reasons to use non-const global variables?", second paragraph:
"A good example is an log file,..." should be "A good example is a log file,..."

nascardriver
January 18, 2020 at 7:45 am · Reply

Lesson amended, thanks!



Agna
January 10, 2020 at 2:37 pm · Reply

```
1 namespace constants
2 {
3     extern constexpr double gravity { 9.8 }; // has external linkage, is directly accessible
4 }
```

Isn't this invalid because constexpr can't be forward declared (and it's not inline)? I think the example should be changed to "const" or clarify in the comment.

nascardriver
January 12, 2020 at 3:19 am · Reply

It's valid, but it doesn't make sense. I changed this example to use `const` instead, thanks for your suggestion!

Chandler
December 3, 2019 at 8:32 pm · Reply

Parenthesis used in variable initialization instead of the { }

nascardriver
December 4, 2019 at 4:19 am · Reply

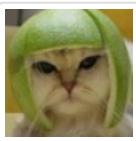
Lesson updated, thanks again :)

Chandler
December 4, 2019 at 5:12 pm · Reply

You're welcome. Teamwork =)

G-S
November 25, 2019 at 1:30 am · Reply

"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!" -- could something like this not be fixed with shadowing>? for example for the nuclear missile example reinitializing the variable g_mode within the dosomething function like so int g_mode{} causes "missile" not to launch due to the shadowed value being hidden and g_mode(2) to be destroyed once the dosomething function is exited.



Alex
November 25, 2019 at 10:25 pm · Reply

Presumably doSomething() has a legitimate reason to change the global g_mode state. If it doesn't, it would be better off using a local variable named "mode", not a shadowing variable named "g_mode" that is actually a local variable.

« Older Comments 1 2 3