Sample Chapter:

Chapter 19
Global Variables Are Evil

- Global variables are memory locations that are directly visible to an entire software system.
- The problem with using globals is that different parts of the software are coupled in ways that increase complexity and can lead to subtle bugs.
- Avoid globals whenever possible, and at most use only a handful of globals in any system.

Contents:

19.1. Overview

"Global Variables Are Evil" is a pretty strong statement! Especially for something that is so prevalent in older embedded system software. But, if you can build your system without using global variables, you'll be a lot better off.

Global variables (called *globals* for short) can be accessed from any part of a software system, and have a globally visible scope. In its plainest form, a global variable is accessed directly by name rather than being passed as a parameter. By way of contrast, non-global variables can only be seen from a particular module or set of related methods. We'll show how to recognize global variables in detail in the following sections.

19.1.1. Importance of avoiding globals

The typical motivation for using global variables is efficiency. But sometimes globals are used simply because a programmer learned his trade with a non-scoped language (for example, classical BASIC doesn't support variable scoping – all variables are globals). Programmers moving to scoped languages such as C need to learn new approaches to take full advantage of the scoping and parameter passing mechanisms available.

The main problem with using global variables is that they create implicit couplings among various pieces of the program (various routines might set or modify a variable, while several more routines might read it). Those couplings are not well represented in the software design, and are not explicitly represented in the implementation language. This type of opaque data coupling among modules results in difficult to find and hard to understand bugs.

Global variables are evil.

19.1.2. Possible symptoms

You may have problems with global variable use if you observe any of the following when reviewing your implementation:

- ✘ More than a handful of variables are defined as globally visible (ideally, there are none). In C or C++, they are defined outside the scope of any procedure, so they are visible to all procedures. An indicator is the use of the keyword extern to access global variables defined outside the scope of your compiled module.
- ✘ Variables are used in a routine that are neither defined locally, nor passed as parameters. (This is just another way of saying they are global.)
- ✘ In assembly language, variables are accessed by label from subroutines or modules rather than via being passed on the stack as a parameter value. Any access to a labeled memory location (other than in a single module that has exclusive access to that location) is using a global.

The use of global variables is sometimes defensible. But their usage should be for a very few, special values, and not a matter of routine. Consider their occasional use a necessary evil.

19.1.3. Risks of using globals

The problem with global variables is that they make programs unnecessarily complex. That can lead to:

➤ Bugs caused by hidden coupling between modules. For example, misbehaving code in one place breaks things in another place.

➤ Bugs created in one module due to a change in a seemingly unrelated second module. For example, a change to a program that writes a global variable might break the behavior of code that reads that variable.

It may be necessary to have variables that are global, or at least serve a similar purpose as global variables. The risk comes from using global variables too freely, and not using mitigation strategies that are available.

A few globals might be worth the risk if used carefully.

19.2. Why are global variables evil?

Global variables should be avoided to the maximum extent possible. At a high level, you can think of global variables as analogous to GOTO statements in programming languages (this idea dates back at least as far as Wulf (1973)). Most of us reflexively avoid GOTO statements. It's odd that we still think globals are OK.

First, let's discuss what globals are, then talk about why they should be avoided.

19.2.1. Definition of global variables

The strict definition of a global variable is a variable that has global scope – meaning it can be seen everywhere in the entire program. As a practical matter, a global variable is one that can be accessed outside the scope of the routine that defines it or accepts it as a parameter. In other words, a global variable seems to appear out of thin air when reading a particular piece of code. It isn't defined; it isn't in the input parameter list; the reference to it is just there, with no easy way (other than a text search) of finding out what is going on with it. We will call non-global variables *locals* for convenience.

Global variables are defined in a variety of ways depending on the programming language. Here are some examples:

- **Assembly:** any variable accessed via a label is a global. For example,

```
LDA MyVar
```

loads the accumulator from MyVar – and MyVar is a global variable. This means that in assembler programs pretty much anything that isn't a reference to the stack or a register value is a global.

- **Classical BASIC:** all variables are global variables. They can be read or written from anywhere within a program. You may be using a version of BASIC which supports local variables, but that is a comparatively recent addition to the language.
- **C and C++:** all variables defined outside the scope of a procedure: For example:

```
 int GlobVar;        /* this is a global variable */
void MyFunc(void)
{  GlobVar = 5; }
```

- **Java:** variables aren't normally global unless you use a trick, such as declaring variables public static to make them visible outside the declaring object.

It is common to confuse the distinction between dynamic *vs.* static variables compared to local *vs.* global variables. A dynamic variable is one that is automatically created on the stack at run-time, especially in a scoped language such as C or Java. In these languages, temporary variable storage space is created on the stack every time a subroutine or method is called, and released when a return operation is performed. When dynamically allocated variables are released (for example, when you do a return from the subroutine that defined them), their value becomes undefined. Static variables, on the other hand, are given a permanent address and can be counted upon to retain their values indefinitely without being thrown away. In C, all variables defined within a routine are dynamic (unless the static keyword is used), and all variables defined outside any routine are static.

A static variable is not necessarily a global variable. Its memory address alone doesn't determine which other modules can see it. A dynamic variable is not necessarily local either, if a pointer to it is passed around. Thus, while it is most common in C that global variables are also static variables, it is not accurate to say all static variables are global. Here is an example in C of a variable that is static, but not global:

```
int IncrErrCount (void)
{ static int ErrCount = 0; // zeroed at startup
  ErrCount += 1;         // tally another error
  return(ErrCount);      // return error count to date
}
```

The variable ErrCount is permanently allocated, but is local to the subroutine. No other routine can see (know the address of) that variable. This use of a static variable is not a problem, because it is only visible inside the defining routine.

19.2.2. Quasi-global variables

In languages that support separate compilation of source code files, there are *quasi-global* variables which are visible everywhere inside a single source code file, but not visible to code in other source code files. In C code this is accomplished by using the keyword *static* with a top-level variable, as in this example:

```
static int GlobVar;   /* a quasi-global variable */
void MyFunc(void)
{ GlobVar = 5;
}
```

The static keyword in this example hides the variable from other compiled modules, keeping it hidden from most of the system if we assume there are a large number of different source code files compiled to make the entire system. A quasi-global variable is better than a completely global variable, because at least you know that it can't be accessed from outside the code in the file you're looking at. But it still has all the other disadvantages of a global variable, especially if the file it is used in has a lot of code that references that variable.

Quasi-globals are better than full globals.

Quasi-globals are better than regular globals because their scope (code that can see them as a variable) is restricted to a single compiled file. But, the bigger that file is, the more code can see the variable, and thus the higher the risk.

19.2.3. Global variable problem: implicit coupling

Why are globals such a big deal? The main issue is that global variables cause implicit coupling between potentially far-flung pieces of code. Consider, for example, a C program that has perhaps 30 different ".c" files written by a half-dozen different programmers. You have this situation:

```
file:   GlobalDefs.c
…
int TurboFlag = 0; //defined as a global
…

file:   PerformanceMonitor.c
…
if(somecondition){TurboFlag=1;} //turbo mode on
…

file:   UserInterface.c
…
if(somecondition){TurboFlag=0;} //turbo mode off
…
```

```
file:   Compute.c
…
// use TurboFlag to choose fast vs. accurate
if(TurboFlag) { SomethingFast; }
else   { SomethingSlow; }
…
```

In this example, which is representative of how many programs use global operating mode variables, the global variable is defined in one place, changed in two other places, and used to control program flow in a fourth. For all we know, there are other places it can be changed too. (A global text search might reveal those other places, but how can you keep that type of information in your head for more than one or two global variables in a program?)

Here are some example situations where this approach can get you into trouble:

- You want to know what a global does, or even just its data type. You have to search for the definition, which could be anywhere. If you are lucky it is in a file named something like GlobalsDefs.c instead of tucked away someplace obscure.
- You need to change how the variable works, but have trouble finding all the places the variable is used. A text search will find all the places in the code it is used unless there are pointers to it. If you use pointers you're in for a tough time, because a text search won't be enough to find everywhere the variable is accessed.
- At run-time, something is changing the variable in an unexpected way, and you don't know how that is happening. Because the variable could be changed from anywhere, there is no way to look for routines that have the variable in their parameter list or otherwise isolate the problem. Solving this requires a debugger with watchpoint capability, which interrupts execution when a particular memory address is accessed. But, you have to reproduce the problem during debugging for this to work, which isn't always easy to do.

In other words, the fact that you can access the global from anywhere makes it difficult to isolate or identify the places where you actually do access it.

A related issue is the spreading of bugs via implicit links through global variables. If you change code at one place, you have no easy way of knowing what other parts of the code will be affected. Yes, you can do a text search of everywhere the global is accessed, and hope that no pointers or difficult-to-search-for references exist. But it is all too easy to get that wrong.

19.2.4. Global variable problem: bugs due to interference

In some small processors where global variables are used for speed, there isn't enough RAM for every routine to have its own independent variable space. This means that programmers map several global variables onto the same RAM space, and try to keep straight which values are being used at any given time. Such an approach is just begging to have very nasty bugs, and should be avoided.

If you don't have a lot of RAM and need to re-use it from one routine to the next, the stack mechanism built in to many programming languages, such as C, is a much less risky way to do the same thing. If you only have 128 bits of RAM to work with, there might not be much you can do. But, if this is the case, see *Chapter 18. The Cost of Nearly Full Resources*, because what you are doing is increasing your software development cost (and risk) to save hardware costs, which is very often the wrong tradeoff to be making.

19.2.5. Global variable problem: lack of mutex

Another popular use for global variables is as a mechanism to pass values from one process to another. For example, a global variable might be used as a place for an interrupt service routine to store an input byte until a processing routine can retrieve it. The reason a global is used is convenience, since an ISR doesn't accept parameters. One problem with this approach is that you can't actually be sure that the only code that sets the variable is the ISR unless you search the entire code base and verify no other routines do that. A way to mitigate this risk is to use a quasi-global variable in a file which defines only the ISR and the routine to which the ISR passes data, restricting the global variable visibility to be within that file and nowhere else.

More generally, it is common for global variables used to communicate between processes to have concurrency issues. Using a fully global variable for communication can almost always be avoided. When it must be done, it is important to protect that global variable with a mutex to ensure no strange and elusive bugs appear due to concurrency problems. (See *Chapter 20. Mutexes and Data Access Concurrency*.)

19.2.6. Pointers to globals

Sometimes you need to find all references to a global to understand or modify a program. A text search will find all references to a global – but only if there are no pointers to it. If there are pointers to a global variable passed around, then it is even more difficult to figure out where the global might be read or written. Forbidding pointers to globals is the best way to avoid such problems.

Pointers to globals are even more evil.

19.3. Ways to avoid or reduce the risk of globals

Fortunately, there are ways to avoid globals, or at least make them more manageable if they simply must be used. Below are some typical approaches. Not all of these techniques are mutually compatible. So, this should be considered a bag of tricks that can be used, rather than a list of good practices that should all be followed. Most programs will need more than one of these techniques, depending on the situation.

19.3.1. Get a more capable processor

In some cases, the reason global variables are used is to speed up program execution. For example, some processors have page zero operations that provide shorter instructions and faster access to the first 256 bytes of RAM. Yes, there are some cases where this savings in speed or size is worth doing. But such situations are fewer than there used to be. You should consider that using globals in the first 256 bytes to exploit this feature is likely to significantly increase your software development costs, just as any other optimization technique that involves exploiting limited hardware resources will. *Chapter 18. The Cost of Nearly Full Resources* discusses this in more detail. This technique should be used only on very large production runs (millions of hardware units manufactured).

> If optimization forces you to use globals,
> get a better compiler or a faster processor.

Note that if a compiler is smart enough to map variables into page zero for you, *and* that complexity is completely hidden from the programmer, *and* the compiler doesn't have bugs that get this mapping wrong, *then* it is perfectly fine to use this optimization. We have no problem with using smaller instructions that run faster! The problem comes when programmers have to spend a lot of extra effort to do this. This puts them at increased risk of bugs because the optimization requires using risky data structures (globals) instead of less risky data structures (locals) in source code.

19.3.2. Move the global inside the procedure that needs it

Some globals are created not because they truly need to be global, but because they need to retain their values across multiple invocations of the procedure that uses them (in other words, they need to be static). In C++ it is common to use constructors and destructors to do this sort of thing. But, in C it is a little trickier to do. One approach in C is to simply move the definition of the global into the highest level routine that needs it, then pass references to it down into subroutines.

> Use a static local variable instead of a global when possible.

Here's an example routine that uses a global:

```
int ErrCount = 0;  // initialized to zero at startup
int IncrErrCount (void)
{ ErrCount += 1;     // tally another error
  return(ErrCount);  // accumulated error count
}
```

But, there is no need to make ErrCount a global if IncrErrCount() is the only procedure that needs to modify it. Instead, it could be rewritten as:

```
int IncrErrCount (void)
{ static int ErrCount = 0; // initialized at startup
  ErrCount += 1;      // tally another error
  return(ErrCount);  // accumulated error count
}
```

In this example, ErrCount is only visible to the routine that needs it, so it is no longer a global variable. The C keyword static is used to tell the compiler to allocate a variable in memory that retains its values between calls to the procedure rather than making the variable temporary and placing it on the stack. (Note that in C, the initialization of a static variable happens only once when the program is first run, not every time the routine is called.)

If IncrErrCount called other routines that needed to know what the value of ErrCount was, it could pass the value or a pointer to that value down to the called routines.

19.3.3. Make a shared variable quasi-global instead of global.

If you must use a global variable to share a value across several routines, then a way to contain the spread of problems is to put only the routines that access that value into a single source code file, and declare the variable as a quasi-global using the static keyword. For example:

```
static int ErrCount = 0;  // initialized at startup

int IncrErrCount (void)
{ ErrCount += 1;    // tally another error
  return(ErrCount);  // return error count to date
}

int GetErrCount(void)
{  return(ErrCount); // return error count to date
}

void ClearErrCount(void)
{  ErrCount = 0;  // reset error count to zero
}
```

In this case of defining a top-level global variable (one that is not inside a procedure), the static keyword means "not visible outside this program file." So that guarantees the variable ErrCount can't be accessed outside the file it is included in. This limits the damage, and in C is often a necessary evil.

C++ has the concept of a friend to document what is going on with such shared variables, and provide stricter control over who sees and who doesn't see a shared variable. While using friends involves some of the risks of globals, it's far better to use a friend than a fully global variable. You can think of friends as a more sophisticated version of quasi-globals.

19.3.4. Make the global an object, and provide access methods

If a global variable must be accessed in many places, and there is just no way around it, then making it a globally visible object rather than an ordinary global variable may have several advantages.

Make globals into objects rather than directly accessible variables.

If you are using an object oriented language such as C++, then this is relatively easy. But even in a non-OO language such as C, this can be done without too much trouble. For example, let's say you want to create a global error counter along the lines of previous examples. You could do the following:

- Create a separate .c file that only contains one related set of globally visible data objects and their access routines.
- For each object, use a static definition at the top level. This makes it visible to all procedures in that .c file, but nowhere else in the program (in other words, it makes it quasi-global).
- Provide procedure calls to read, write, or otherwise operate on the data.

The net result is that you have a variable that can't be accessed except via the access methods you've provided. This is a close approximation to how C++ handles objects in many respects. For example, a global count of errors encountered might have this code in the separate global variables .c file:

```
static int ErrCount = 0;  // initialized at startup

inline int IncrErrCount (void)
{ ErrCount += 1;     // tally another error
  return(ErrCount);  // return error count to date
}

inline int GetErrCount(void)
{  return(ErrCount); // return error count to date
}

inline void ClearErrCount(void)
{  ErrCount = 0;  // reset error count to zero
}
```

The **inline** keyword is used to encourage the compiler to put the code to access the variable in-line with other code, eliminating the overhead of a procedure call. With a good optimizing compiler, this code will compile with the same runtime speed as an ordinary global variable. (This can also be true for C++ approaches that put the global variable into an object with access methods.)

While not as syntactically clean as it would be in C++, the above code gets the job done. A more C++-like notation might be to call the procedures: ErrCount_Incr, ErrCount_Get, and ErrCount_Clear, but the naming is a matter of choice for your style guidelines.

The obvious question is, why would we want to go to all the trouble of defining the access routines when we could have just defined a global and been done with it? Here are some reasons. You may or may not find them compelling, but they are worth considering.

- **It makes it impossible to get a pointer to the variable.** Because the pointer is hidden behind the access functions (it is declared static at the top level), it is impossible for any module outside the definition file to see it. Thus, it is impossible to point to. This could be helpful, because it makes it much more difficult for a developer to violate a no pointer to globals rule. With this approach, identifying places in the code that access the variable still requires a text search for the name of the access method. But, at least there are no pointers to worry about.
- **It provides debugging hooks.** Debugging of problems with globals can be difficult because they can be accessed from anywhere. Debugging via address matching on loads and stores for globals (watch-point debugging based on the global variable address) isn't always possible, and needs on-CPU hardware support or an emulator to work well. With the access methods we've described, it's impossible to access the variable without going through a defined access method. The access routines can therefore be instrumented to show what is happening. Debugging via breakpoints based on program counter values (with a breakpoint at the start of an access routine) is a universal debugger capability. If you don't have a debugger at all, you can at least put print statements in the access methods to see what's happening with the variable by modifying only one spot in the software system.
- **It simplifies concurrency management.** A tricky part of globals is making sure that multiple concurrent tasks accessing them don't have problems with sharing. A mutex or other locking mechanism is used (see *Chapter 20. Mutexes and Data Access Concurrency*). With this method you only need to put the mutex logic in one place – the access procedure. With other methods, you need to make sure you get the mutex right every place the variable is touched, which gives many more places to verify, and many more chances

to make a mistake (most commonly, the mistake of forgetting to use the mutex).

19.3.5. If you must use a global, document it well

For whatever reason, you might find you still must have a global variable in your program. Hopefully you won't need many! But for those you need, here are some things you can do to reduce the chance of having problems.

- Use a naming convention to make it easier to recognize all globals in a text search.
- If the global is shared by concurrent tasks, use a naming convention to remind developers that it must be protected by a mutex or lock.
- Don't allow taking a pointer to a global.
- Use comments where a global is defined to keep an up-to-date list of everywhere it is used, and any rules for manipulating the variable. It shouldn't be too hard to have automated tools generate this list dynamically, if you have access to all the source code.

If you still have to use a global, spend a lot of effort making it as clean and well documented as possible.

19.4. Pitfalls

Globals are evil, and it's better to avoid using them. As with any idea, avoiding globals can be carried to absurd extremes, but the usual pitfall is using too many globals instead of too few.

19.5. For more information

19.5.1. General topic search keywords

- Global variable
- Scoping

19.5.2. Recommended reading

Ganssle, J., "A pox on globals," *Embedded System Design,* October 2006, pp. 57-60.
Jack hates globals about as much as we do. Here's another take on the same topic.

Jones, N., "Efficient C code for eight-bit MCUs," *Embedded Systems Programming,* November 1998, pp. 66-83.
This article discusses a number of efficiency techniques, including using static non-global variables instead of globals.

19.5.3. Additional reading

Wulf, W. & Shaw, M., "Global variables considered harmful," *ACM SIGPLAN Notices,* February 1973, pp. 28-34

An early paper explaining why global variables are bad.

Topic Quick Reference

ISBN-13: 978-0-9844490-0-2
ISBN-10: 0-9844490-0-0
Drumnadrochit Education LLC

Printed on acid-free paper in the United States of America
2010 2011 2012 2013 2014 2015 9 8 7 6 5 4 3 2 1