Electrical & Computer ENGINEERING

18-642:

Global Variables Are Evil!

9/19/2019

EDGE CASE RESEARCH

"Global variables are responsible for much undebuggable code, reentrancy problems, global warming, and male pattern baldness. Avoid them!"

— Jack Ganssle

Carnegie Mellon University

Global Variables Are Evil!

Edge Case Research

- **Anti-Patterns:**
  - **More than a few read/write globals**
  - **Globals shared between tasks/threads**
  - **Variables have larger scope than needed**

- **Global variables are visible everywhere:**
  - Use of globals indicates poor modularity
    - Globals are prone to tricky bugs and race conditions
  - Local static variables are best if you need persistence
    - File static variables can be OK if used properly
    - Don’t make procedures globally visible if not needed

Global vs. Static Variables

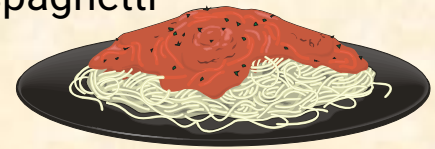
Edge Case Research

Globals:

```
uint32_t gVar = 0;
void gProc(…)   { … }
```

Global risks

- Written from anywhere
  - Debugging: who wrote it?
- Read from anywhere
  - Changes break everything
- Multithreaded race conditions
- Increased complexity
  - Data flow “spaghetti”



File Static:

```
static uint32_t fsVar = 0;
static void fsProc(…)   { … }
```

- Only inside .c file
- Use with small .c files
- Like C++ “private”

Local Static:

```
void gProc(…)
{ static uint32_t sVar = 0;
… }
```

- Persistent variable value
- Can’t be seen outside procedure

Avoiding And Removing Globals

Edge Case Research

- **Define smallest scope possible (variables and procedures)**
  - Change global to file static; file static to local static
- **Arrange .c files based on access to data**
  - Example: time of day updated by ISR
    - File static time of day variable in TimeOfDay.c
    - Put timer tick ISR in TimeOfDay.c
    - Put procedure to disable interrupts & read time of day in TimeOfDay.c

- **Configuration values & constants**
  - Use const keyword – prevents multiple writers
  - Read-only access to global configuration data structure
  - Limit visibility to need-to-know within relevant .h file

Best Practices For Avoiding Globals

- **Use smallest practical scope for variables & procedures**
  - Ideally, zero global variables
  - Use file static if you must; local static if you can
  - A good compiler will generate efficient code

- **Reorganize code to reduce scope**
  - Write anything except locking variables only in one place
  - File static variables for small groups of functions
    - More or less the idea of C++ private keyword
    - Take care of data locking when reading
- **Global Variable Pitfalls**
  - Lots of global variables is a sign of bad code

Example

https://betterembsw.blogspot.com/2013/09/getting-rid-of-global-variables.html

You have a "globals.c" file that defines a mess of globals, including:

```
int g_ErrCount;
```

which might be used to tally the number of run-time errors seen by the system. I've used a "g_" naming convention to emphasize that is a global, which means that every .c file in the program can read and write this variable with wild abandon.

Let's say you also have the following places this variable is referenced, including globals.c just mentioned:

- **globals.c:** int g_ErrCount; // define the variable
- **globals.h:** extern int g_ErrCount; // other files include this
- **init.c:** g_ErrCount = 0; // init when program starts
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

Create an Error Counting Module

Create separate "object" for error counting: ErrCount.c

- **globals.c:** // not needed any more for this variable
- **ErrCount.c:** int g_ErrCount; // define the variable
- **ErrCount.h:** extern int g_ErrCount; // other files include this
- **init.c:** g_ErrCount = 0; // init when program starts
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

Initialize Where Defined

- **ErrCount.c:** int g_ErrCount = 0; // define and init variable
- **ErrCount.h:** extern int g_ErrCount; // other files include this
- **init.c:** // no longer needed
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

Convert to File Static

- **ErrCount.c:** static int ErrCount = 0; // only visible in this file
- **ErrCount.h:** // static variables are invisible outside .c file
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

Add Accessor Function

Edge Case Research

- **ErrCount.c:** static int ErrCount = 0; // only visible in this file
  - inline void ErrCount_Incr() { ErrCount++; }
  - inline int ErrCount_Get() { return(ErrCount); }
  - inline void ErrCount_Reset() { ErrCount = 0; }
- **ErrCount.h:**
  - inline void ErrCount_Incr(); // increment the count
  - inline int ErrCount_Get(); // get current count value
  - inline void ErrCount_Reset(); // reset count
  - // Note that there is NO access to ErrCount directly

- **moduleX.c:** ErrCount_Incr(); // tally another error
- **moduleY.c:** XVar = ErrCount_Get(); // get current number of errors
- **moduleZ.c:** ErrCount_Reset(); // clear number of reported errors

Advantages of this Approach

- Software authors can only perform intended functions specific to an error counter: increment, read, and reset. Setting to an arbitrary value isn't allowed. If you don't want the value changed other than via incrementing, you can just delete the reset function. This prevents some types of bugs from ever happening.
- If you need to change the data type or representation of the counter used that all happens inside ErrCount.c with no effect on the rest of the code. For example, if you find a bug with error counts overflowing, it is a lot easier to fix that in one place than every place that increments the counter!
- If you are debugging with a breakpoint debugger it is easier to know when the variable has been modified, because you can get rid of the "inline" keywords and put a breakpoint in the access functions. Otherwise, you need watchpoints, which aren't always available.
- If different tasks in a multitasking system need to access the variable, then it is a lot easier to get the concurrency management right inside a few access functions than to remember to get it right everywhere the variable is read or written (get it right once, use those functions over and over). Don't forget to make the variable volatile and disable interrupts when accessing it if concurrency is an issue.