# Tip 47: static versus global variables

Ben Klemens

3 January 2012

**level**: writing longer programs
**purpose**: write clearly

A *global* variable is one whose scope is the entire program. We generally consider global variables to be evil, because they impose cognitive load as you try to remember which variables are relevant where and how the variable you are looking at in `file1.c` might have changed in `file2.c`. The rule of thumb is to keep the scope of variables as small as practicable, and a global variable is by definition as large a scope as you can get.

Meanwhile, a variable declared outside of a function is in the *static* storage class. These varaibles are initialized on startup, and there are a few other annoying restrictions about them. See the table in Tip #20 (Entry #070).

Converting a static variable into a global variable takes a few steps:

- Declare the variable outside of all the functions in one file; let that be `int glovar`.
- Set up a header that declares the variable with the `extern` modifier: `extern int glovar`; let that header be `globals.h`
- In every file where `glovar` should be visible, `#include globals.h` at the top of the file.

I'm rehashing this procedure to point out how you have to work to get truly global variables in C. The norm is that you have variables whose scope starts wherever you've declared them and continues to the end of the file. Is this intermediate level of scope evil or not?

Below is an example. It will repeatedly evaluate a function to produce Gnuplottable output. Run via:

```
./a.out | gnuplot -p
```

There's a throwaway counter that tells us how often the function itself got evaluated. This is a popular diagnostic that you'll see here and there, and is mostly useful for debugging.

```
#include <stdio.h>

typedef double (*one_d_fn)(double);
int counter=0; //a quick variable for debugging

double eval_f(one_d_fn f, double x){
    counter++;
    return f(x);
}

void plot(one_d_fn f, double min, double max){
    printf("plot '-'\n");
    for (double x=min; x<max; x+=0.05)
        printf("%g %g\n", x, eval_f(f, x));
    printf("e");
    fprintf(stderr, "you evaluated the function %i times.\n", counter);
}

//an arbitrary polynomial. Rewrite to whatever you want to plot.
double f(double x){ return x*x + .4*x*x*x -3;}

int main(){
    plot(f, -1, 1);
}
```

I contend that the `counter` variable is not all that bad. Æsthetically, the mathematical ideal of a function does not in any way depend on context or state. You give it the same inputs, you get the same outputs, no matter the day of the week. So that's a failure: Incrementing the `counter` is a side-effect in functional terms, and it doesn't need to be.

If I decide to include this little counter for debugging, should I modify the header for `eval_f` and all the calls; if I decide later that I'm done with this diagnostic, should I re-modify everything? Having to keep track of things outside the header is error-prone, but so is all that modifying and fixing. Back to æsthetics, the header is clearer without lots of extra inputs for housekeeping. We're calling a side-effect a side-effect.

So the tip for the day is to bear in mind that static functions have scope from declaration to the end of the file, which may not be very much real estate, which puts them in a middle-ground between truly global variables and local-to-function variables. Used with reasonable caution and common sense, we can sometimes use these static variables to communicate between functions more legibly and with less cognitive load than the alternative of passing absolutely everything as an argument.

**Globals: not so evil** As an aside, I opened with the premise that global variables are universally evil, whereas static-scope variables are a middle-ground that's not so evil. But I don't think that the *global=evil* premise is entirely true either, albeit for reasons that are orthogonal to the discussion of static variables.

Consider a program translated to both English and French. Every time a function has to print a message to the screen, it needs to have on hand a variable telling the function what language to use. Sending that variable as an argument to every function is a waste of time and pure clutter which obfuscates what the functions are really intended to do. In fact, the custom with every program I've ever seen is to use a global variable—an environment variable with a name like `LANG` or `LC_LOCALE`. This is the right thing to do: there really is a global state that affects every variable, so there should be a global variable that annotates that state.

The real problem is not that you have global variables, but that your program depends upon a global state—the specification of your program requires that functions behave differently depending on whether the user speaks English or French. Rewriting the program to eliminate the global variables (like generating a giant structure with lots of state variables and passing that around) is treating a symptom. We've left functional Eden, and some of our functions have to evaluate differently depending on the hardware, users, and the rest of the environment.