Tip 64: Consistency-check external variables

Ben Klemens

7 February 2012

level: multiple-file programs
purpose: avert pain

I keep a bug log. It's a simple idea, and I don't know if there's a single origin to it [the famed Donald Knuth had one]. But by taking notes on my errors, I can try to spot errors and can get a sense of the things that I personally should be looking out for, and the first thing to suspect when something goes wrong.

That's where today's tip comes from. I won't bore you with the play-by-play, but this is one of those cases where, when the author says *People have trouble with the following gotcha*, what the author means is *I now have or have had trouble with the following gotcha*.

Here's the gotcha. In file 1, we declare a struct and/or a function:

```
//file1.c
typedef struct {
   double a, b;
   int c, d;
} alphabet;

alphabet abcd;

int pain(void){
    return 3;
}
```

Now we make use of these things in another file. We need to declare all terms, so we paste in the typedef and function header. Let's use the `extern` keyword to refer to `abcd`, a variable declared in file one.

This sample is valid because we can use the `extern` keyword anywhere, even within a function, meaning that we have as much control over the scope of an `extern` as we do over the scope of a native-to-the-file variable.

```
//file2.c
typedef struct {
   double a, b;
   int c, d;
} alphabet;

extern alphabet abcd;

int pain(void);

abcd.d = pain();
```

Do you see how this is going to go horribly awry yet? A week later, file 1 evolves: `c` is now floating-point and the `pain` function takes in an alphabet struct:

```
//file1.c
typedef struct {
   double a, b, c;
   int d;
} alphabet;

alphabet abcd;

int pain(alphabet alpha){
    return alpha.d;
}
```

The problem: `file2` will still compile without errors or warnings. Both files are internally consistent, and each is compiled separately into its own internally consistent `.o` file with no warnings. Then, the linker that joins the object files to form an executable doesn't know enough C to do any but the most rudimentary consistency checks; it finds all relevant symbols and throws no errors. [Remember how the GCC is the GNU Compiler Collection, and in theory `file1.o` could be from C source and `file2.o` from ADA or Fortran and the linker will still do its job.] I can attest that a bug at this stage, after the compiler threw out zero errors, is hard to reverse-engineer.

We establish consistency via header files. Rewrite:

```
   //sharedinfo.h
typedef struct {
   double a, b, c;
   int d;
} alphabet;

extern alphabet abcd;
int pain(alphabet alpha);


   //file1.c
#include "sharedinfo.h";
alphabet abcd;
```

```
int pain(alphabet alpha){
    return alpha.d;
}
```

```
   //file2.c
#include "sharedinfo.h";
abcd.d = pain();
```

When `file1.c` compiles, the system checks that the declarations in `file1.c` match the declarations in `sharedinfo.h`; when `file2.c` compiles, the system checks that the declarations in `file2.c` match the declarations in `sharedinfo.h`; equality is transitive here, so we conclude that the declarations in the two `.c` files match.

Notice, by the way, that `abcd` is now declared as `extern` in `file1.c` via the included header, and is then declared as a not-external file-local variable. C doesn't care, and treats the `extern` declaration as just harmless redundancy.

At this point, you probably think this too easy and how could anybody mess this up? Well, people do have trouble with this gotcha. Textbooks advise that you control scope by declaring the `extern` near its use. But you gain that close control at the price of consistency-checking, which is more than it's worth.

It's also easy to be lazy and just cut/paste a function header; versus creating a new header file, moving the declarations, adding the new file to the repository and makefiles, then adding `#include`s in both places. But the consistency-checking is worth the extra effort.