16

Scope

In which our hero learns that limits can be a good thing, at least, in C programming, and perhaps in life as well, or maybe not?

16.1 Boundaries

There is an old adage that states “Good fences make good neighbors.” It’s often attributed to Robert Frost [F14] but actually has roots in saying much older – people have known for a long time that a little separation can be a good thing.

Computer systems know this too, and provide a number of tools and techniques to separate parts of computations that don’t need to interact. A classic example is the operating systems [AD17], which provides a strong degree of separation between each running program, giving each the illusion that it owns the entire machine. Distributed systems do this as well; by running separate parts of a larger-scale computation on different machines, a distributed application or service can tolerate the crash of a machine [GD02].

Most programming languages, including C, provide some facilities to help keep parts of a program distinct from one another as needed. Specifically, the **scoping rules** of a language enable or prevent a programmer from accessing parts of a program from other parts of a program.

In this chapter, we’ll provide a little introduction to C scoping rules, using examples to show how they work. We won’t provide full coverage of **scope** here, however; that’ll have to wait until a little later when we know a little bit more about how larger-scale C programs are constructed.

16.2 A Simple Example

Let’s start with a simple example. Here, we just have a function `foo()` and it defines a two variables, `x` and `y`:
At Line 3 of the code, the variable \( x \) is defined. At this point, we can refer to variable \( x \) for the rest of this function and it is legal to do so; thus, \( x \) is in scope. The variable \( x \) is what we call a local variable, because it can only be directly referred to by the function in which it was created; this is why we use the terms stack-allocated variables and local variables interchangeably.

Later on in the same function we define \( y \), and now it too is in scope and can be referred to from within \( \text{foo}() \). Thus, at Line 2 and beyond within \( \text{foo}() \) we can use \( x \) freely, and at Line 4 and beyond we can use \( y \) freely. We can use them both, in fact, until this function returns; at this point, space for them is deallocated and they both fall out of scope.

16.3 Same Name, Different Place

Let’s look at another example. In this example, we have two functions, \( \text{main}() \) and a function \( \text{main}() \) calls, \( \text{foo}() \):

```c
void foo() {
    int x;
    ...
}

int main() {
    int x = 0;
    foo(x);
    return 0;
}
```

As you can see from the example, the variable \( x \) is defined in two places, in \( \text{main}() \) and in \( \text{foo}() \). Is this legal? Does it make sense? Who invented liquid soap and why?\(^1\)

The answers: this is completely legal, makes perfect sense, and William Shephard [C13]. The variable \( x \) within \( \text{main}() \) refers to one particular spot in the stack, and the variable \( x \) within \( \text{foo}() \) refers to another spot on the stack. They are independent variables that just happen to have the same name.

\(^1\)Obscure movie reference here. Be forewarned: it’s not really even a good movie.
We also note their life times overlap. The x within main() is defined when main() begins and is alive until the program exits; within that time, x within foo is allocated, presumably used by foo(), and then deallocated when foo() returns. Figure 16.1 presents a depiction of the stack before, during, and after the call to foo().

With this trivial example, you can see some of the real benefits of scoping. When you write a function, you can use (mostly) whatever variable names you like. You don’t have to worry if other functions have local variables with the same name; rather, you just focus on the task at hand, of writing the clearest, most readable, and most concise code you can.

16.4 Block Scope

In C, it turns out any block of code that begins with a left curly brace and ends with a right curly brace defines a scope known as block scope (this includes the examples we saw above, where the function definition serves as the main block). For example, you could define a while loop as follows:

```c
...
while (x < 10) {
    int x_squared = x * x;
    ...
}
...
```

In this code, if you only need x_squared inside of the while loop, it is better to define it within the loop instead of beforehand. Doing so avoids polluting the name space of the current function; why would you want x_squared to be accessible (perhaps by accident) outside the loop if it is only needed within it?

Slightly stranger, but still perfectly legal, is to simple declare a new block in the middle of a function, as follows:
int main() {
    int a;
    int b;
    {
        int x;
        int a;
        ...
        a = 10;
    }
    ...
}

This new scope within main() gives you two new variables, x and a, and allows you to use them until you reach the end of the compound statement and fall out of the block of code. Once again, note that it is OK for each scope to reuse a previously defined name (in this case, a); the most recently defined (and still active) scope is the one that is used. For example, when setting a to 10 (Line 8) refers to a defined on Line 6 not on Line 2.

16.5 Global Variables

Thus far, we’ve only allowed for local variables, allocated upon the stack, with the scoping rules as defined above. However, sometimes it is useful to define a variable for use throughout all (or many) of the functions in your code. We call such variables global variables (or, sometimes, external variables), and we define them outside of a function. Here is a simple example:

```
int debug = 1;

int foo() {
    if (debug) { printf(" started foo...
\n"); }  
    ...
}

int main() {
    if (debug) { printf(" started main...
\n"); }  
    if (debug) { printf(" ended main...
\n"); }  
    return 0;
}
```

In this example, we define a global variable debug which is set to either non-zero or zero to indicate whether debug messages should be printed. Because debugging is useful throughout all the functions, it is more convenient to simply define a global variable that
can be accessed within all functions, rather than passing the value of `debug` to each function and thus polluting the **call signature** (i.e., the parameters that are passed to a function) in the description of the API.

Note that a local variable of the same name can be defined in any function; if so, the local variable of that name will be used instead of the global variable. For example, we could define `foo()` from the previous example as follows:

```c
int foo() {
    int debug = 0;
    if (debug) { printf(" started foo...\n"); }
    ...
}
```

In this case, a new local variable `debug` is on the stack for the duration of the call to `foo()`, and thus is used instead of the global definition. In this way, local definitions override global ones.

### 16.6 Access To Globals In Other Files With `extern`

Although we won’t deal with this aspect of programming for a little while, in this section and the next we’ll introduce two keywords that are commonly used in C programs to either gain or restrict access to global variables. The first is `extern`, which enables C programs to access global variables not defined in the current file.

For this to make sense to you, however, you have to know that C programs are often compiled from many different source-code files and libraries – a topic we have not yet discussed. For now, just assume that sometimes in the code you are writing, you’ll need to access such a variable, and thus you’ll need some way to **declare** it in the current source code file you are writing.

For example, assume this is some variable (an integer) named `error_type` which stores some information you wish to access. You know that `error_type` is **defined** elsewhere (that is, space has been declared for it as a global variable) and you just wish to access it within the file you are editing. To do so, all you do is add this declaration somewhere in your code:

```c
extern int error_type;
```

```c
void foo() {
    if (error_type == ERROR_IO) {
        ...
    }
}
```
As you can see, to the rest of the code you write, `error_type` is just an integer, but instead of defining it at the top of the file, you just indicate its type and name and let C handle the rest.

### 16.7 Some Privacy For Globals With `static`

In some cases, you may wish to define a global variable but not make it widely available to every function within your program. C provides a limited way to do this, via the `static` keyword.

For example, let’s say we have a group of functions that all use a global variable `debug` (as above). However, we don’t want functions in other parts of the program (namely, in other files) to be able to access this `debug` variable. Therefore, all we do is add `static` before the type of the definition as follows:

```c
static int debug = 0;
```

Declaring a variable `static` in this manner allows the functions within the file you are editing to access the variable as a global, but prevents functions from other files from doing so.

Thus, for any global variable you use, you should think about whether to add a `static` to it; is it needed outside of the file you are editing? If not, definitely add `static` and avoid polluting the namespace of your program.

Interestingly, `static` can also be applied to functions, thus limiting how such functions can be called (again, only from within one source file, instead of from elsewhere). Commonly, when you define some kind of helper function that is only needed to implement an API but should not be called by others, prepending it with `static` is good practice when defining it:

```c
static int helper_function() {
    ...
}
```

We’ll see more examples of using `static` in this manner later in the book.

---

2The namespace, informally, consists of all the variables and functions you define; you should be particularly wary of adding global variables to the general namespace, as it might lead to naming collisions across parts of a large program. Use `static` to avoid such problems.
16.8 Common Mistakes

Before closing, we highlight a couple of common mistakes people make. The first is using too many global variables. When a young programmer discovers that it is possible to declare global variables and access them with reckless abandon from within any function, sometimes the temptation is too much to overcome! And suddenly, the program is filled with one global after the other, and the code becomes what professionals call “a big pile of stuff.” While there is no general rule to follow, trying to minimize the use of global variables is probably a good idea.

Humorously, another common mistake is not using global variables enough. In some cases (as with the debug flag above) it is simply much easier to have a global variable accessible in every function, instead of changing every function and passing in said variable to it.

The guideline you should use, as always: what makes the code cleaner, more readable, and more maintainable? If adding a global does this, then feel free; if adding a global starts to make the code a mess, don’t do it. Experience, as always, helps.

One last mistake, of sorts: please do always initialize all global variables. While the language may do you a favor and set them to a zero value by default, it is always clearer in the code to set the value at first to something sensible if at all possible when defining the variable.

16.9 Summary

We have seen the introduction of the notion of scope. Stack allocated variables have local scope and thus can be referred to directly only within the text of the given function; statically-defined global variables have global scope and can be referenced from within any function. One new keyword, extern, helps provide access to global variables defined in other files or libraries; the other, static, limits how much global variables (or functions) can be accessed from other files. Use globals, but not too often; while there is no simple rule,
you’ll find that the more you write C code, the more you’ll get a sense of where the use of them is appropriate, and where it’s just plain offensive. Who knew that something as simple as a global variable definition could offend other programmers?
References

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau
Arpaci-Dusseau Books, 2008-2017
Available: http://www.ostep.org
More gratuitous self promotion. Can you really complain?

[C13] “Who invented liquid soap, and why?”
Answered by Diana Cretu
February 2013
According to Ms. Cretu, self-proclaimed “photo spammer”, liquid soap was invented in the early 1800's and then patented in 1865 by William Shepphard (“Improved Liquid Soap”, U.S. patent number 49561). If you’re going to include Wikipedia as a source, why not Quora too? Should you even trust Quora? See www.quora.com/Can-you-trust-the-information-on-Quora to find out.

[F14] “Mending Wall”
Robert Frost
In “North of Boston”, 1914 (published by David Nutt)
www.poetryfoundation.org/poems-and-poets/poems/detail/44266
Who are we to analyze poetry? So let’s not. But here is a snippet from this fine poem:
My apple trees will never get across
And eat the cones under his pines, I tell him.
He only says, ‘’Good fences make good neighbors’’.
Makes you wonder about the barriers we build between one another, no?

[GD02] “MapReduce: Simplified Data Processing on Large Clusters”
Sanjay Ghemawat and Jeff Dean
OSDI ’04, San Francisco, CA, October 2004
This is not really the proper citation for how distributed systems can tolerate crashing machines – an idea almost as old as computers themselves. However, it is a recent reference to a popular system, MapReduce, which tolerates machine crashes and is generally interesting to learn about.