

# C Compiler, Part 10: Global Variables

Feb 18, 2019

*This is the tenth post in a series. Read part 1 [here](#).*

We're back! I said I was going to do a non-compiler post next, but that turned out to be a lie. Instead, we're going to implement global variables. This isn't too complicated, but it lets us learn about some new sections of object files and program memory.

As always, tests are [here](#).

**Note for macOS Users:** since the last post, Apple started phasing out support for 32-bit programs on macOS. What that means for us is that if you're using the default C compiler on macOS Mojave, you'll get an error if you try to compile for a 32-bit backend<sup>1</sup>:

```
$ gcc -m32 example.c
ld: warning: The i386 architecture is deprecated for macOS (remove from th
ld: warning: ignoring file /Applications/Xcode.app/Contents/Developer/Plat
ld: dynamic main executables must link with libSystem.dylib for architectu
clang: error: linker command failed with exit code 1 (use -v to see invoca
ld: warning: The i386 architecture is deprecated for macOS (remove from th
```

But never fear! The Homebrew version of GCC works just fine, although it still emits a warning:

```
$ gcc-8 -m32 static.c
ld: warning: The i386 architecture is deprecated for macOS (remove from th
```

I'm pretty sure there's a way to get the default compiler to build 32-bit programs as well but I don't know what it is.

When you run a 32-bit program (like the ones produced by *your* compiler), you might also get a warning that it isn't optimized for your computer. This is also due to Apple's efforts to phase out 32-bit programs, but you don't need to do anything about it.

The bigger issue, of course, is that the next version of macOS won't run 32-bit programs at all. I plan to update all my posts before that happens to cover 64-bit compilation too. And yes, I do regret targeting a 32-bit architecture to begin with, thank you for asking. Luckily, apart from calling conventions all the differences so far are pretty minor.

With that out of the way, let's move on to...

## Part 10: Global Variables

We can already handle local variables declared inside functions. Now we'll add support for global variables, which any function can access.

```
int foo;

int fun1() {
    foo = 3;
    return 0;
}

int fun2() {
    return foo;
}

int main() {
    fun1();
    return fun2();
}
```

Note that global variables can be shadowed by local variables of the same name:

```
int foo = 3;

int main() {
    int foo = 4; // shadows global 'foo'
    return foo; // returns 4
}
```

Global variables are similar to functions in that they can be declared many times, but defined (i.e. initialized) only once:

```
int foo; // declaration
```

```
int main() {  
    return foo; // returns 3  
}  
  
int foo = 3; // definition
```

And, like functions, global variables must be declared (but not necessarily defined) before they're used:

```
int main() {  
    return foo; // ERROR: not declared!  
}  
  
int foo;
```

Declaring a function and a global variable with the same name is an error:

```
int foo() {  
    return 3;  
}  
  
int foo = 4; // ERROR
```

Unlike local variables, global variables don't need to be explicitly initialized. If a local variable isn't initialized, its value is undefined, but if a global variable isn't initialized its value is 0.

```
int main() {  
    int foo;  
    return foo; // This could be literally anything  
}
```

```
int foo;  
  
int main() {  
    return foo; // This will definitely be 0  
}
```

Note that we're using the terms "declaration" and "definition" the same way we did for functions. This is a global variable declaration<sup>2</sup>:

```
int foo;
```

This is both a declaration and a definition:

```
int foo = 1;
```

The `static` and `extern` keywords would add some extra complications, but we won't support those yet.

Now let's move on to...

## Lexing

No new tokens this week, so we don't have to touch the lexer.

## Parsing

Previously, a program was a list of function declarations. Now it's a list of top-level declarations, each of which is either a function declaration or a variable declaration.

So our top-level AST definitions now look like this:

```
toplevel_item = Function(function_declaration)
               | Variable(declaration)
toplevel = Program(toplevel_item list)
```

And we need a corresponding change to the top-level grammar rule:

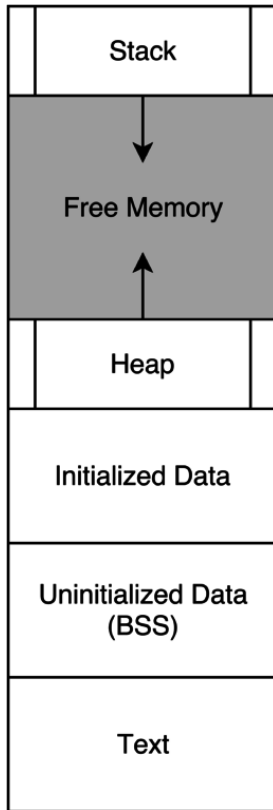
```
<program> ::= { <function> | <declaration> }
```

### ☑ Task:

Update the parsing pass to support global variables. The parsing stage should now succeed on all valid examples in stages 1-10.

## Code Generation

Global variables need to live somewhere in memory. They can't live on the stack, because they need to be accessible from every stack frame. Instead, they live in a different chunk of memory, the data section. We've already seen what a running program's stack looks like; now let's step back and see how all of its memory is laid out<sup>3</sup>:



The x86 instructions we've been dealing with so far all live in the text section. Our global variables will live in the data section, which we can further subdivide into initialized and uninitialized data—the uninitialized data section is usually called BSS<sup>4</sup>.

So far we've only generated assembly for the text section, which contains actual program instructions; let's see what the assembly to describe a variable in the data section looks like:

```
.globl _my_var ; make this symbol visible to the linker
.data          ; what's next describes the data section
.align 2       ; this data should be aligned on 4-byte intervals (i.e. it
_my_var:
.long 1337     ; allocate a long integer with value 1337
```

A couple things to note here:

- The `.data` directive tells the assembler we're in the data section. We'll also need a `.text` directive to indicate when we switch back to the text section.
- A label like `_my_var` labels a memory address. The assembler and linker don't care whether that address refers to an instruction in the text section or a variable in the data section; they're going to treat it the same way.

- On macOS, `.align n` means “align the next thing to a multiple of  $2^n$  bytes”. So `.align 2` means we’re using a 4-byte alignment. On Linux, `.align n` means “align the next thing to a multiple of  $n$  bytes”, so you’d want `.align 4` to get the same result.

Once you’ve allocated a variable, you can refer to its label directly in assembly:

```
movl %eax, _my_var ; move the value in %eax to the memory address of _
```

So the basic gist here is:

1. When you encounter a *declaration* for a global variable, add it to the variable map. The variable map entry will be its label instead of a stack index:

```
var_map = var_map.put("my_var", "_my_var")
```

Note that this new variable map entry must be visible when we generate later top-level items; this isn’t true of entries we add while processing function definitions.

2. When you encounter a *definition* for a global variable, with an initializer, emit assembly to allocate it in the data section. Then emit a `.text` directive before you go back to generating function definitions.
3. When you encounter a *reference* to a variable, handle it the same way you did before. If its entry in the variable map is a label instead of a stack index, of course, you should use it directly instead of as an offset from `%ebp`. If it doesn’t have an entry, that’s an error.

But there are a few wrinkles.

## Uninitialized Variables

If, by the end of the program, we have any variables left that have been declared but not defined, we need to declare them in a special section for uninitialized data. On Linux, all uninitialized data lives in the BSS section, which also includes any variables initialized to 0. On macOS it’s a little more complicated: uninitialized static variables go in BSS, and uninitialized global variables go in the common section, which indicates to the linker that they may be initialized in a different object file. We don’t support static variables yet, so on macOS we don’t need to store anything in BSS. Of course, we also don’t have any tests with multiple source files, so if you just use BSS instead of common, effectively making all global variables static, the tests will still pass.

The data section consists of the actual values of our data; we can load it directly into memory and use it as-is. The BSS and common sections, on the other hand, don’t contain all of our

uninitialized values, because they would just be big blocks of zeros. Storing a big block of zeros on disk would be a waste of space. Instead, we just store the size of BSS and common in our binary, and allocate that much memory for them when we load the program. So keeping initialized and uninitialized variables separate is just a trick to reduce the size of binaries.

On macOS, we can allocate space in the common section using the `.comm` directive:

```
.text
.comm _my_var,4,2 ; allocate 4 bytes for symbol _my_var, with 4-byte a
```

Allocating space in BSS, on the other hand, looks almost exactly the same as allocating a non-zero variable, but we'll use `.zero 4` to allocate 4 bytes of zeros instead of `.long n` to allocate a long integer with value `n`:

```
.globl _my_var ; make this symbol visible to the linker
.bss          ; what's next describes the BSS section
.align 4      ; this data should aligned on 4-byte intervals (Linux a
_my_var:
.zero 4       ; allocate 4 bytes of zeros
```

Note that in assembly, unlike in C, it's perfectly fine to reference a label like `_my_var` before that label is defined. That's why we can wait until the end of the program to allocate any uninitialized variables.

## Non-Constant Initializers

Global variables are loaded into memory before the program starts, which means we can't execute any instructions to calculate their initial values. Therefore their initializers need to be constants. For example, this isn't valid:

```
int foo = 5;
int bar = foo + 1; // NOT A CONSTANT!
int main() {
    return bar;
}
```

Most compilers permit global variables to be initialized with constant expressions, like:

```
int foo = 2 + 3 * 5;
```

This requires you to compute `2 + 3 * 5` at compile time. You can support this if you want, but you don't have to; the test suite doesn't check for it.

## Validation

To recap, here's what we need to validate:

- Variables, including global variables, are declared before they are defined.
- No global variable is defined more than once.
- No global variable is initialized with a non-constant value.
- No symbol is declared as both a function and a variable.

It's easy to validate the first bullet point during code generation; we're doing that for local variables anyway. The remaining points can be validated either during code generation, or in a separate validation pass. I'd recommend handling them wherever you validate function definitions and calls.

### ☑ Task:

Update the code generation pass (and your validation pass, if you have one) to fail with an error for all invalid stage 10 examples, and succeed on all valid stage 10 examples.

## PIE 🍷

If you compile a program with global variables using a real compiler, the assembly will look quite different from what we described above. You may also notice, if you're on macOS, that the linker will warn you about the assembly your compiler produces:

```
$ ./my_compiler global.c
ld: warning: The i386 architecture is deprecated for macOS (remove from th
ld: warning: PIE disabled. Absolute addressing (perhaps -mdynamic-no-pic)
```

PIE stands for "position-independent executable", which means an executable consisting entirely of position-independent code. This section briefly explains what position-independent code is and why you might need it, but doesn't explain how to implement it. Feel free to skip it if you're not interested.

Position-independent code is code that can run no matter where it's loaded in memory, because it never refers to absolute memory addresses. The code our compiler produces is not position-independent, because it has instructions like:

```
movl $3, _my_var
```



In order for this instruction to run, the linker needs to replace `_my_var` with an absolute memory address. This works if we know the absolute address of the data and BSS sections in advance.

Position-independent code, on the other hand, never refers to the address of symbols like `_my_var` directly; instead, those addresses are calculated relative to the current instruction pointer. In case I didn't have enough of a reason to regret targeting a 32-bit architecture, position-independent assembly is much simpler with a 64-bit instruction set:

```
movl $3, _my_var(%rip) ; use _my_var as offset from instruction pointer
```

To get the same result with a 32-bit architecture you need something like this:

```
call    ___x86.get_pc_thunk.ax
L1$pb:
leal    _my_var-L1$pb(%eax), %eax
movl    (%eax), %eax
```

I won't walk through exactly what this code is doing; if you're curious, [this article](#) gives a good overview of position-independent code for x86.

There are two reasons you might want to generate position-independent code:

1. You're compiling a shared library. Maybe this is a really widely used library, like libc. Maybe all or most processes on a system will want a copy of this library. It seems like a waste to have a separate copy for every process, eating up all your RAM. Instead, we can load the library into physical memory just once, then map it into the virtual memory of every process that needs it. But we can't guarantee a library the same starting address in every process that loads it. So sharing one library between several processes only works if the library works no matter what memory address it's at—which is to say, it needs to be position-independent. However, we're compiling an executable, not a library, so this doesn't apply to us.
2. You have address space layout randomization (ASLR) enabled. ASLR is a security feature that makes some memory corruption attacks harder to carry out. Many of these attacks involve forcing program execution to jump to the instructions an attacker would like to execute. With ASLR enabled, memory segments are loaded at random locations<sup>5</sup>, which makes it harder for attackers to figure out what address to jump to. Code needs to be position independent in order to run correctly when loaded to a random memory address. Since Apple really wants all macOS applications to support ASLR<sup>6</sup>, the linker will try to build a position-independent executable by default, and complain if it can't.

The fact that your compiler can't generate position-independent executables is just one of many, many reasons you shouldn't use it to build real software. I don't have that much faith in these blog posts, and neither should you!


If you want to learn more about ASLR, I found [these slides](#) helpful. Of course, there's also [Wikipedia](#).

## Up Next

So far, I've been implementing a compiler and writing posts as I go. This system worked really well for a while, but now it's starting to work less well; I realized that some decisions I made in earlier stages made this stage harder to complete, so I had to go back and change them. I think I'm likely to run into more problems like that in later posts. So I'm going to take a break, finish building the compiler (whatever I decide "finished" means), and then come back and write the rest of this series. I probably won't post another update for six months. So basically...I'm going to keep posting at about the same rate I have been.


When I come back, I'll have a plan for what to cover in the rest of the series. See you then!

*If you have any questions, corrections, or other feedback, you can [email me](#) or [open an issue](#).*


<sup>1</sup> The compiler that ships with the XCode Command Line Tools—the one that was giving me this error—is actually *not* GCC. It's [Clang](#), another open-source compiler that's developed mostly by Apple. XCode installs Clang at `/usr/bin/gcc`, no doubt for very sound and legitimate reasons, although I don't know what they are. 


<sup>2</sup> The standard actually considers this a *tentative definition* (section 6.9.2):


*A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier static, constitutes a tentative definition.*

Basically, if we can't find a real definition anywhere else in the file, we can treat a declaration like a definition with an initial value of 0. We're still going to call it a declaration, though. 

<sup>3</sup> [Typical computer data memory arrangement](#) by Majenko is licensed under [CC BY-SA 4.0](#).

This diagram is an oversimplification; it doesn't show every memory segment we might find in a running program. Also, sometimes memory segments are laid out in a different order—we'll talk about that later. The point is that we have a dedicated chunk of memory for global variables. 

<sup>4</sup> BSS stands for "Block Started by Symbol," which is a relic of an assembler written in the 1950s(!). You can read more [here](#) if you want to go down a bit of a Wikipedia rabbit hole. 

<sup>5</sup> Exactly which memory segments are randomized, and how random their base addresses actually are, varies between systems. 

<sup>6</sup> Source. 

---

Want to become a better programmer? [Join the Recurse Center!](#)

© 2022 Nora Sandler.