Learning Objectives

- Be able to name the four principal steps of the C build process.
- Be able to identify which C language elements will produce labels and symbols.
- Recognize the difference between an object file's symbol table and its relocation table.
- Understand that types are a feature of the C language that disappear upon compilation.
- Be able to recognize when globals clash, even if the compiler and/or linker cannot tell.

Getting Started

To get set up for today's activity, run these commands on a shark machine:

```
$ mkdir linking
$ cd linking
$ wget http://www.cs.cmu.edu/~213/activities/linking
$ chmod +x linking
$ ./linking
```

Then follow the instructions on your screen, filling in the discussion questions below when you are prompted to do so. As you complete each part of the exercise, you'll reinvoke the `linking` executable repeatedly in the same manner.[1]

3 Phases of Compilation

Problem 1. In the first step, where did all the extra code coming from? What do you think the lines beginning with '#' and a number mean?

The extra code came from `stdio.h` and other headers included by that header.

The lines beginning with '#' and a number tell the second step of compilation which lines of code came from which files.

[1] In case you get lost or want to see a past set of instructions again, you can seek directly to any part of the activity. Each invocation of `linking` outputs a "page number" in the upper-right corner; if passed to `linking` as a command-line argument, this replays that part. You can also provide the section numbers from this sheet.

Problem 2. The `gcc -S main.c` step produces a file called `main.s`. What type of file is this? Examining its contents, you should notice labels corresponding to the global variable and both functions. Given *only* a label's name, can you tell what its C type is?

`main.s` contains assembly language. You can't tell what C type a label is; that information has been discarded. You *may* be able to reconstruct some of the type by looking at the code or data stored at each label and how it is used, but it is not possible to do this perfectly.

4 The Symbol Table

Problem 3. Looking at the addresses in the leftmost column, do you notice anything suspicious about the locations of `global` and `set_global`?

Both `global` and `set_global` appear to be at address 0! You can't have two different things at the same address, and you can't have *anything* at address 0.

5 Object File Sections

Problem 4. Which section contains `set_global`? How about `global`?

`set_global` is in the `.text` section, and `global` is in the `.data` section.

Problem 5. The output also contains flags describing the properties of each section. Thinking back to attack lab, describe one limitation that these flags (or the lack thereof) impose on each of the sections from your previous answer.

Only the `.text` section has the `CODE` flag. This is how the operating system knows to set things up so that only the data in the `.text` section can be executed as machine code.

Most of the sections (except `.data` and `.bss`) have the `READONLY` flag. This is how the operating system knows to set things up so that the data in those sections cannot be modified by the running programe.

Problem 6. The sections' offsets within the object file differ, but what do you notice about their memory addresses (`VMA` and `LMA`)?

All of the memory addresses are zero.

6 Relocations

Problem 7. Try disassembling the object file using `objdump -d`. At what address(es) does the code seem to expect to find `global`? How about the `printf()` function?

The code seems to expect to find `global` at offset 0 from register `%rip`—which doesn't make a whole lot of sense. This must be another value that still needs to be filled in, like the section addresses. Similarly, the `call` instructions that will call `printf` are currently making a call to the very next instruction—you may remember that this is the same as "offset 0 from register `%rip`."

Problem 8. The object file also includes what's known as a "relocation table." Examine this with `objdump -r`. What locations does it record (the leftmost column), and do you have a guess as to why this will be useful?

The locations in the leftmost column identify all of the places in the machine code where a value still needs to be filled in. The next step will use these "relocation records" to update all of the machine instructions with the correct addresses for `global`, `printf`, etc.

7 The BSS

Problem 9. `global` has moved to a different section: which one? Can you guess why the compiler treats zero-initialized variables specially?

`global` is now in the `.bss` section. Maybe, if all the variables that will be zero-initialized are gathered into this section, they can be handled more efficiently, somehow?

Problem 10. Look at the `Size` column. How large will the `.bss` section be in the loaded process memory image? Now look at the entries in the `File off` column. How large is the `.bss` section is the executable file? Can you infer how the `.bss` section is treated differently from the other sections in an ELF executable?

The `.bss` section's size in memory will be 4 (just big enough for one `int`) but its offset in the file is the same as the offset of the *next* section in the file. In other words, it doesn't take up any space in the file.

10 Clashing Symbols

Problem 11. Take a quick look at both `main_zero.c` and `helper.c`. What do you think will happen when we try to link these modules together?

Both of them define `global`, so the link operation will fail.

13 Missing Declarations

Problem 12. Will building this program (linking against `helper.o`) work? If so, why? If not, at what step of the build (preprocessing, compilation, assembly, or linking) will it fail?

This will fail at the compilation stage, because the compiler does not have any type information for `global`.

(For historical reasons, a missing declaration of a *function*, like `set_global`, causes the compiler to *guess* what its type is, rather than issuing an error. It still complains about this, but, when you actually compiled `main_scary.c`, did you notice that it printed "*error*: '`global`' undeclared" but "*warning*: implicit declaration of function '`set_global`'"? Warnings don't stop compilation.)

15 Mismatched Types

Problem 13. What's wrong with the program now?

`global` is declared as a `float` here, even though `helper.c` defines it as an `int`. And `set_global` is declared to take a `float` argument, even though `helper.c` defines it to take an `int` argument.

Problem 14. Will building this program work? If so, why? If not, at what step will it fail?

This program will compile and link successfully, even though it's wrong! The compiler doesn't look at `helper.c` at all while compiling `main_scary.c`, so it does not notice the mismatch. And the *symbols* `global` and `set_global`, in the object files, don't have types, so the linker does not notice either.

16 *(Advanced)* Silent Failure

Problem 15. Did the build fail as early as you expected?

If you wrote anything but "this program will compile and link successfully, even though it's wrong" as your answer to the previous question, you got a surprise!

18 *(Advanced)* Mutability

Problem 16. What is inconsistent now? How do you expect the program to behave?

Now the inconsistency is that `global` is defined with the `const` modifier in `helper.c`, but it's declared without that modifier in `main_scary.c`, and furthermore `main` tries to modify the value. This program will still compile and link, but will crash with a "Segmentation fault" message when it tries to modify the value. If you look at the symbol table for `helper.o` again, you'll see that `global` is now in the `.rodata` section, which, like `.text`, is protected from modification.