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?

[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?

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`?

5 Object File Sections

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

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.

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

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?

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?

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?

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?

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?

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?

15 Mismatched Types

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

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

16 *(Advanced)* Silent Failure

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

18 *(Advanced)* Mutability

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