15213 Lecture 15: Linking

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 a copy of today's activity, log into a shark machine and do the following from a new, empty folder:

1. `$ wget http://www.cs.cmu.edu/~213/activities/lec15`
2. `$ chmod +x lec15`
3. `$ ./lec15`

Then follow the instructions on your screen, filling in the below discussion questions only as prompted to do so. As you complete each part of the exercise, you'll simply reinvoke the `lec15` executable repeatedly in the same manner[1].

3 Phases of Compilation

1. In the `cpp` step, where is all the extra code coming from?

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 and both functions. Given *only* a label's name, can you tell its type?

4 The Symbol Table

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

[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 outputs a "page number" in the upper-right corner; if passed to `lec15` as a command-line argument, this replays that part. You can also provide the section numbers from this sheet.

5 Object File Sections

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

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.

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

6 Relocations

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?

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

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

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

9 Clashing Symbols

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?

12 Missing Declarations

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?

14 Mismatched Types

13. What's wrong with the program now?

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

15 *(Advanced)* Silent Failure

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

17 *(Advanced)* Mutability

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