CSE 374
Programming Concepts & Tools

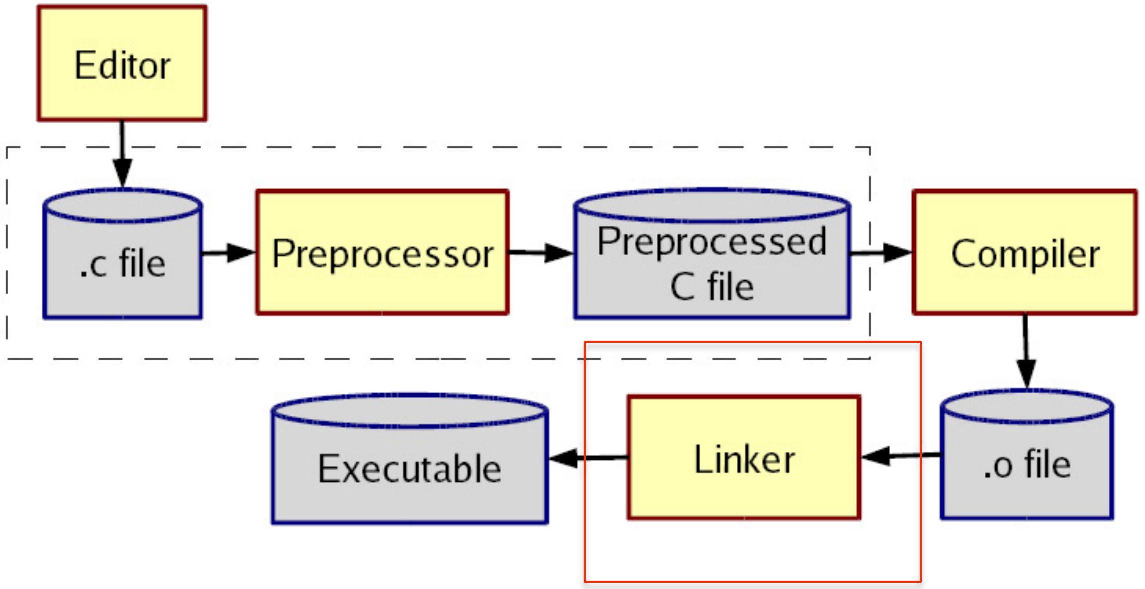
Brandon Myers

Winter 2015

Lecture 20 – Linking and Libraries

(Thanks to Hal Perkins)

The compilation picture (revisited)



Old story, but new details…

Intro to linking

- Linking is just one example of “using stuff in other files”...
- In compiling and running code, one constantly needs other files and programs that find them
- Examples:
  - C preprocessor #include
  - C libraries (where is the code for printf, malloc?)
- Usually you’re happy with programs “automatically finding what you need” so the complicated rules can be hidden
- Today we will demystify and make generalizations

Common questions

TODO: did we answe

1. What you are looking for?
2. When are you looking for it?
3. Where are you looking?
4. What problems do cycles cause?
5. How do you change the answers?

#include files – what really happens?

cpp (invoked implicitly by gcc or g++ on files ending in .c, .cc, .cpp, etc.).

- What: files named “foo” when encountering **`#include <foo>`** or **`#include "foo"`** (note: **`.h`** is just a convention)
- When: When the preprocessor is run (making x.i from x.c, although usually you don’t see this)
- Where: “include path”: current-directory, directories chosen when cpp is installed (e.g., /usr/include), directories listed in INCLUDE shell variable, directories listed via `gcc -I` flags, ...

more #include…

- The rules on “what overrides what” exist, but tough to remember
- Can look at result of cpp to see “what really happened”
  - gcc -E foo.c >foo.i   OR cpp foo.c >foo.i
- Example: for nested #include, the original current-directory or the header file’s current-directory?
- Example: Why shouldn’t you run cpp on one machine and compile the results on another?

What about cycles?

- File a.c calls functions in file b.c which calls functions in file a.c which …
- Not a problem – put *declarations* in header files and include each header file as needed. Actual function *definitions* aren’t circular

Compiled code – .o files

- So far we have talked about finding source code to create compiled code (.o files for C)
- These files are not whole applications, so we have the same questions for “finding the other code”
  - printf, malloc, getmem (called from main), …
- A .o file is not “runnable” – you have to actually link it with the other code to make an executable
- Linking (ld, or called via gcc or g++) is a step between compiling and executing

Linking

- If a C file uses but does not *define* a function (or global variable) foo, then the .o has “unresolved references”. *Declarations* don’t count; only *definitions*.
- The linker takes multiple .o files and “patches them” to include the references. (It literally moves code and changes instructions like function calls.)
- The final executable must have no unresolved references (you have seen this error message)
- What: Definitions of functions/variables
- When: The linker creates an executable
- Where: .o files given as arguments to (and much more...)

Object files (detail)

- every .o file has:
  - code (text) segment where the function definitions go
  - data segment where the global variables go
  - symbol table: each function definition, global variable with a relative memory address
  - relocation table: list of lines of code that need to be fixed (i.e., that have unresolved references)
  - (see these with `nm foo.o`)
    - “T”: definition in the text segment
    - “D”: definition in the data segment
    - “U”: unresolved reference

(static) Linking

- 1. put the code and data segments from all the .o files together into a new file.
- 2. for each unresolved reference in the relocation table find it in one of the symbol tables and fix up the addresses (need to use the relative memory address with the offset of the code/data in the final executable)

More about where

- The linker and O/S don't know anything about main or the C library
- That's why gcc "secretly" links in other things
- We can do it ourselves, but we would need to know a lot about how the C library is organized. Get gcc to tell us:
  - gcc -v -static hello.c
  - Should be largely understandable
  - -static (puts *all* the code you need into a.out during linking)
  - the secret *.o files: (they do the stuff before main gets called, which is why gcc gives errors about main not being defined when you have no main)

Archives

- An archive is roughly a tar file, but with extra header information about the .o files in it
- Create with `ar` program (lots of features, but fundamentally take .o files and put them in, but order matters)
- The semantics of passing ld an argument like -lfoo is complicated and often not what you want:
  - Look for what: file libfoo.a (ignoring shared libraries for now), when: at link-time, where: defaults, environment variables, and the -L flags (analogous to -I)
  - Go through the .o files in libfoo.a in order
    - If a .o defines a needed reference, include the .o
    - Including a .o may add more needed references
    - Continue

The rules for linking, with archives

- A call to ld (or gcc for linking) has .o files and -lfoo options in left-to-right order
- State: “Set of needed functions not defined” initially empty
- Action for .o file:
  - Include code in result
  - Remove from set any functions defined
  - Add to set any functions used and not yet defined
- Action for .a file: For each .o in the archive, in order
  - If it defines one or more functions in set, do all 3 things we do for a .o file
  - Else do nothing
- At end, if set is empty create executable, else error
- some linkers are smart about symbols, but if the linker processes in left-to-right order then you need to put an archive before the .o file that needs it

Library gotchas

1. Position of -lfoo on command-line matters (for many linkers)
   - Only resolves references for “things to the left”
   - So -lfoo typically put “on the right”
2. Cycles
   - If two .o files in a .a need each other, you’ll have to link the library in (at least) twice!
   - If two .a files need each other, you might do -lfoo –lbar -lfoo -lbar -lfoo ...
   - (There are command-line options to do this for you, but they are turned off by default.)
3. If you include math.h, then you’ll need -lm

Another gotcha

4. No repeated function names
   - Two .o files in an executable can't have (public) functions of the same name
     - Can have *static* functions with the same name! ("static" on a function means not externally visible to other .o files)
   - Can get burned by library functions you do not know exist, but only if you need another function from the same .o file
     - (Solution? 1 public function per file?!)
     - (Solution in C++: namespaces; can emulate them in C by prefixing your function names)

Dynamic Linking

- The basic static linking model has disadvantages:
  - Uses lots of disk space (copy library functions for every application)
  - More memory when programs are running (what if the O/S could have different processes magically share code?)
- So we can link later:
  - Shared libraries (link when program starts executing). Saves disk space. O/S can share actual memory behind your back (if/because code is immutable).
  - Dynamically linked/loaded libraries. Even later (while program is running). Devil is in the details.
- “DLL hell” – if the version of a library on a machine is not the one the program was tested with…

Summary

- Things like “standard libraries” “header files” “linkers” etc. are not magic
- But since you rarely need fine-grained control, you easily forget how to control typically-implicit things. (You don’t need to know any of this until you need to know it ☺)
- There’s a huge difference between source code and compiled code (a header file and an archive are quite different)
- The preprocessor includes header files
- The linker includes files from archives of .o files