CSE 303: Concepts and Tools for Software Development

Dan Grossman

Winter 2006

Lecture 21— Linkers, Libraries, Archives, ...

Today

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` and `malloc`)
- Java source files (referring to other source code)
- Java class files (referring to other compiled code)

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

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?

our old friends: files, function names, paths, environment variables, command-line flags scripts, configuration files, ...

Previous example

cpp (invoked implicitly by gcc on files ending in .c).

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).

Where: "include path" current-directory, directories chosen when cpp is installed (e.g., /usr/include), directories listed in INCLUDE shell variable, directories listed via -I flags, ...

The rules on "what overrides what" exist, but tough to remember. Can look at result to see "what really happened".

Example: for nested #include, the original current-directory or the header file's current-directory?

Example: Why shouldn't you run cpp on 1 machine and compile the results on another?

`javac` is similar

If `A.java` defines class `A` to have a field of type `B`, how "does the compiler know what `B` is"?

What: a file named `B.class` (probably the result of compiling `B.java`).

When: When compiling a source file that uses the class `B`.

Where: "class path" current-directory, directories chosen when `javac` was installed, directories listed in `CLASSPATH` shell variables, directories listed via `-classpath` flags, ... (Note: Packages correspond to subdirectories)

The rules on "what overrides what" exist, but tough to remember.

Source code cycles

What if two source files refer to each other?

- C: Can't but don't need to: Put *declarations* in header files and include each header file at most once.
- Java: If `B.class` is not found, but `B.java` is, (implicitly) compile `B.java` (potentially with information the compiler already has about `A`).

IDEs

Fancier development environments provide help with “packages”, “projects”, etc.

Fundamentally, the questions are the same and their are settings and menu items for controlling your development process.

Compiled code

So far we have talked about finding *source code* to **create** *compiled code* (either `.o` files for C or `.class` files for Java).

These files are *not* whole applications, so we have the same questions for "finding the other code".

The Java story is a bit simpler, so we will do it first.

Java class-loading and execution

Recall `java A` *`args`* runs class `A`'s static `main` method with args.

`java` is just a program that finds `A.class` and knows what to do (*interpretation* and/or *just-in-time compilation*).

But it will probably have to find lots of other classfiles too.

Simple (untrue but doable) version: Recursively find all the class files you need before starting execution:

- What: class files referred to
- When: start of execution
- Where: classpath, etc.

Disadvantages?

Java class-loading continued

Actually, the JVM is much *lazier* (technical word) about class-loading; waiting until a class is actually used (technical definition) during execution.

That is, the *when* is "later" and "more complicated".

So is the *where*:

- `jar` files (lots of classes in one file, retrieved together)
- remote class files (applets with code over the web, etc.)
- different *security* settings for classes found different places

Why use a `jar` ("Java archive") file:

- Keep classes that need each other together
- Faster/simpler remote retrieval

Object code is different

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`) is a "when" between compiling and executing.

Again, `gcc` hides this from you (just `-c` or not `-c`), but it helps to know what is going on.

First discuss *static linking*, which is mostly like the untrue version of Java we sketched.

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.)

An executable must have no unresolved references (you have seen this error message).

What: Definitions of functions/variables

When: The linker creates an executable

Where: Other `.o` files on the command-line (and much more...)

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 by end of today.
- `-static` (stick with the simple "get all the code you need into `a.out` story)
- the secret `*.o` files: (they do the stuff before `main` gets called, which is why `gcc` gives errors about `main` not being defined).
- `-lc`: complicated story about finding the *library* (a.k.a. "archive") `libc.a` and including any *files* that provide still-unresolved references.

Archives

An archive is the "`.o` equivalent of a `.jar` file" (though history is the other way around).

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 (`LIBPATH` ?) 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.