CSE 303 Concepts and Tools for Software Development

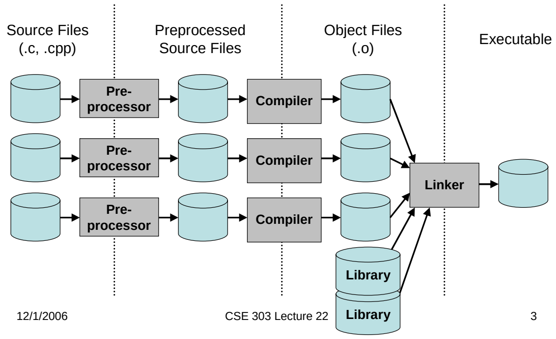
Richard C. Davis
UW CSE – 12/1/2006

Lecture 22 – Linkers

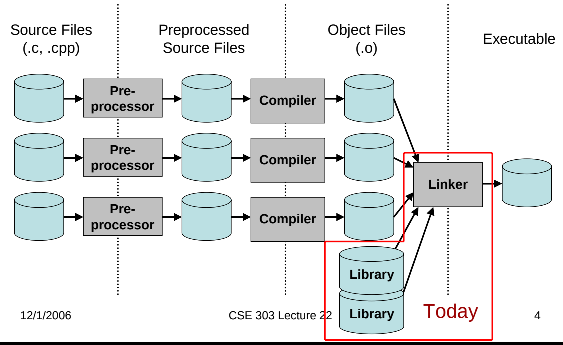
Administravia

- Any questions on HW7?

The Build Process



The Build Process

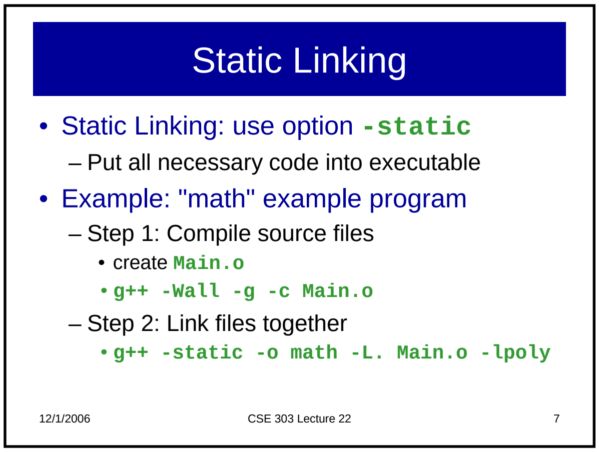


The Goal of the Linker

- Compiled code (`.o` file) is not "runnable"
- Link with other code to make executable
  - Where is the code for `printf` and `new`?
  - We only included the header files…
  - Need to find that code and put in executable
- Normally `gcc`/`g++` hides this from you
- Use `-c` option to stop right before linking
  - We use this to produce `.o` files

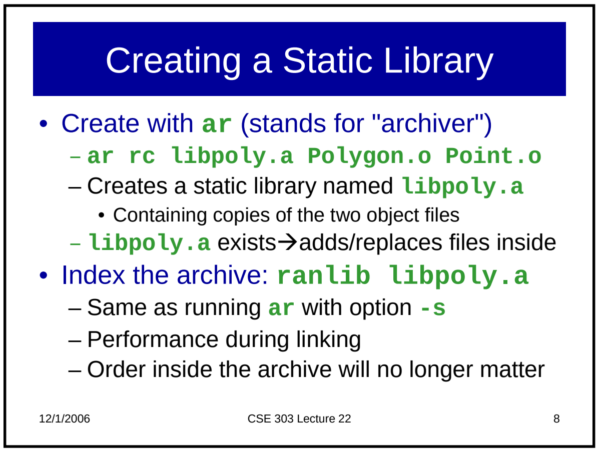
Linking Overview

- C/C++ file uses undefined func./global var.
  - The `.o` file has "undefined references"
  - Note: declarations don't count, only definitions
- Linker "patches" `.o` files to resolve refs.
- Executable has no unresolved refs.
- Ways to invoke linker
  - `ld` command
  - Implicitly through `gcc`/`g++` (we'll do this)



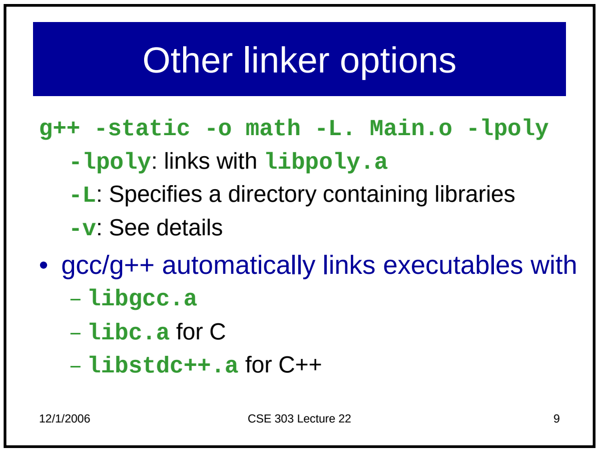
Static Linking

- Static Linking: use option `-static`
  - Put all necessary code into executable
- Example: "math" example program
  - Step 1: Compile source files
    - create `Main.o`
    - `g++ -Wall -g -c Main.o`
  - Step 2: Link files together
    - `g++ -static -o math -L. Main.o -lpoly`



Creating a Static Library

- Create with `ar` (stands for "archiver")
  - `ar rc libpoly.a Polygon.o Point.o`
  - Creates a static library named `libpoly.a`
    - Containing copies of the two object files
  - `libpoly.a` exists→adds/replaces files inside
- Index the archive: `ranlib libpoly.a`
  - Same as running `ar` with option `-s`
  - Performance during linking
  - Order inside the archive will no longer matter

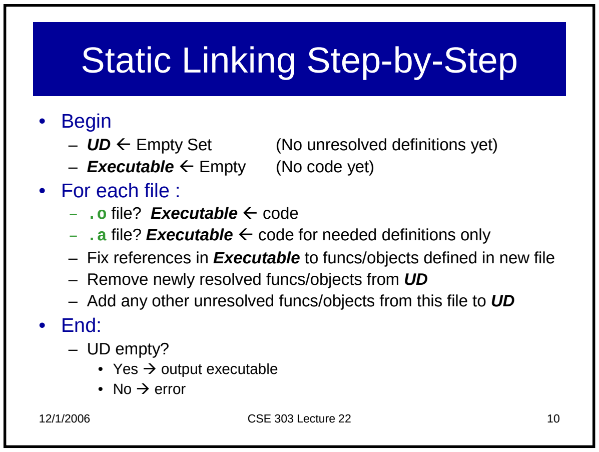


Other linker options

`g++ -static -o math -L. Main.o -lpoly`

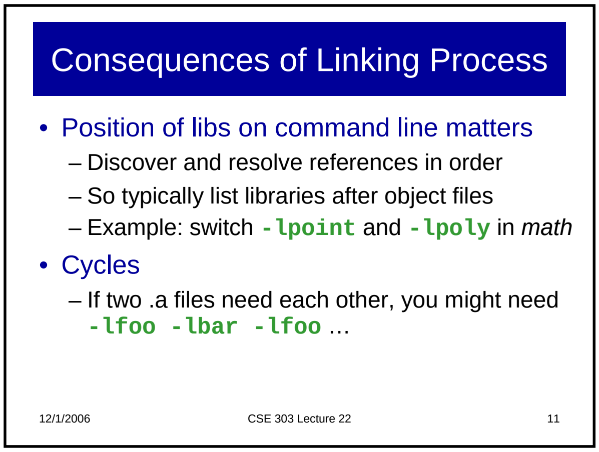
- `-lpoly`: links with `libpoly.a`
- `-L`: Specifies a directory containing libraries
- `-v`: See details

- gcc/g++ automatically links executables with
  - `libgcc.a`
  - `libc.a` for C
  - `libstdc++.a` for C++



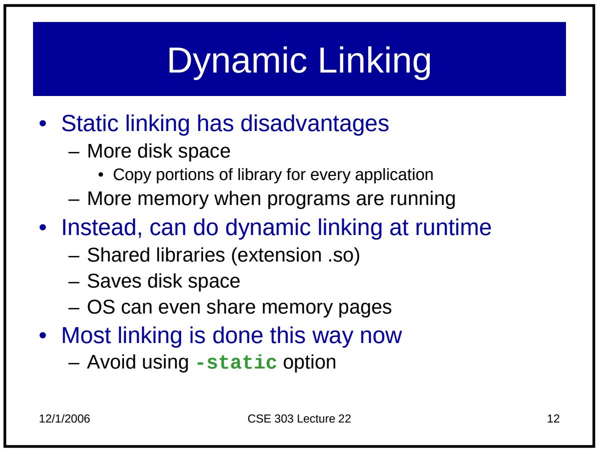
Static Linking Step-by-Step

- Begin
  - ***UD*** ← Empty Set (No unresolved definitions yet)
  - ***Executable*** ← Empty (No code yet)
- For each file :
  - `.o` file? ***Executable*** ← code
  - `.a` file? ***Executable*** ← code for needed definitions only
  - Fix references in ***Executable*** to funcs/objects defined in new file
  - Remove newly resolved funcs/objects from ***UD***
  - Add any other unresolved funcs/objects from this file to ***UD***
- End:
  - UD empty?
    - Yes → output executable
    - No → error



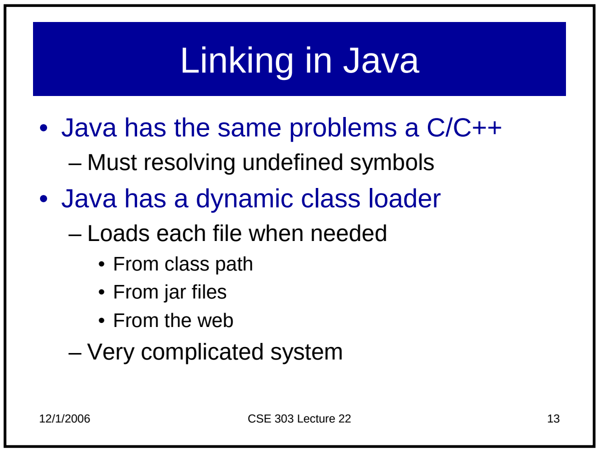
Consequences of Linking Process

- Position of libs on command line matters
  - Discover and resolve references in order
  - So typically list libraries after object files
  - Example: switch `-lpoint` and `-lpoly` in *math*
- Cycles
  - If two .a files need each other, you might need `-lfoo -lbar -lfoo` …



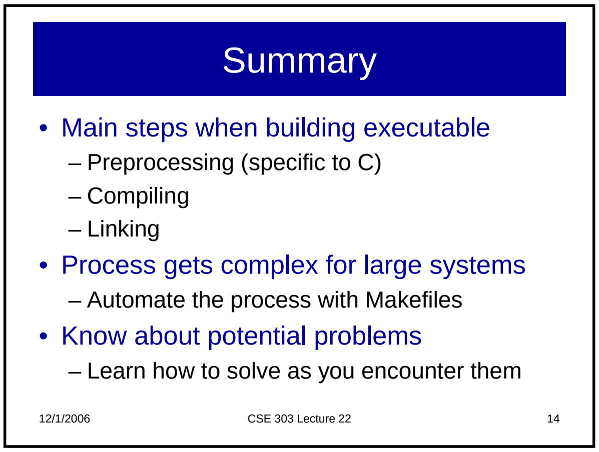
Dynamic Linking

- Static linking has disadvantages
  - More disk space
    - Copy portions of library for every application
  - More memory when programs are running
- Instead, can do dynamic linking at runtime
  - Shared libraries (extension .so)
  - Saves disk space
  - OS can even share memory pages
- Most linking is done this way now
  - Avoid using `-static` option



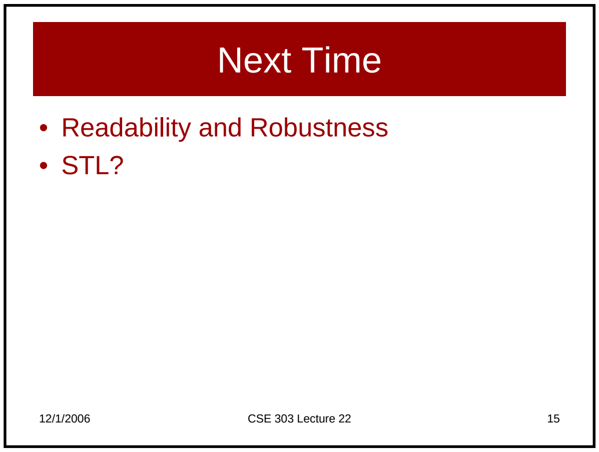
Linking in Java

- Java has the same problems a C/C++
  - Must resolving undefined symbols
- Java has a dynamic class loader
  - Loads each file when needed
    - From class path
    - From jar files
    - From the web
  - Very complicated system



Summary

- Main steps when building executable
  - Preprocessing (specific to C)
  - Compiling
  - Linking
- Process gets complex for large systems
  - Automate the process with Makefiles
- Know about potential problems
  - Learn how to solve as you encounter them



Next Time

- Readability and Robustness
- STL?