Topics for this Lecture

- Build systems

- Static analysis

Topic 1: Build Systems

- In a simple world, compiling and running a computer program is simple:
  - `> gcc –o myexe myprogram.c`
  - `> ./myexe`

- The world is not that simple most of the time, as you may notice if you’ve tried compiling any open source programs

How to Compile a Program

- Most larger programs require many complex commands with many arguments:

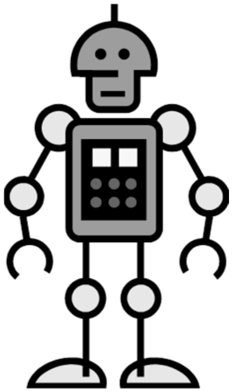
```
> gcc –g –c lib1.c –DARCH_X86 –DLINUX –DDEBUG –D –ftest-coverage –fprofile-arcs –O0
> gcc –g –c lib2.c –DARCH_X86 –DLINUX –DDEBUG –D –ftest-coverage –fprofile-arcs
> gcc –o mainexec m.c lib1.o lib2.o  –DARCH_X86 –DLINUX –DDEBUG –D -ftest-coverage –fprofile-arcs –lm –DNO_X –O3
```

- Compiling all the components of a modern software system may take *a long time*
    - Building the software for Curiosity at JPL was a half hour process or more, from scratch, with hundreds of commands run

Not Just a Shell Script

- Simply bundling all the compilation into a script doesn’t solve the problem
  - If you only change one file, which other files have to be recompiled? Do you start over?
  - A script is a series of commands; if you want to take advantage of multiple machines, you have to design the parallelism yourself

Build Systems



"Let the robot do the boring stuff!"

- Again, automation comes to our rescue
- Build systems:
  - Given a description including at least:
    - Components of a system (files)
    - How they depend on each other
    - How to produce the ones that are not provided by humans
  - A build system:
    - Uses information on which files have been modified and which files don't exist yet to produce the final products – for example, executables – for a system

Build Systems

- Lots of different build systems
    - Some are very simple, don’t do much beyond what I just described
    - Others are very complex, attempt to determine dependencies for you, automatically parallelize compilation, etc.
        - Sometimes integrated with figuring out local configuration (hardware, operating system, available tools)
        - Sometimes integrated with source control or testing

- In this class, we’ll use a very simple system, *make*

Simple Structure of a Makefile

- See dominion/Makefile in the class repository

- Structure is like this:

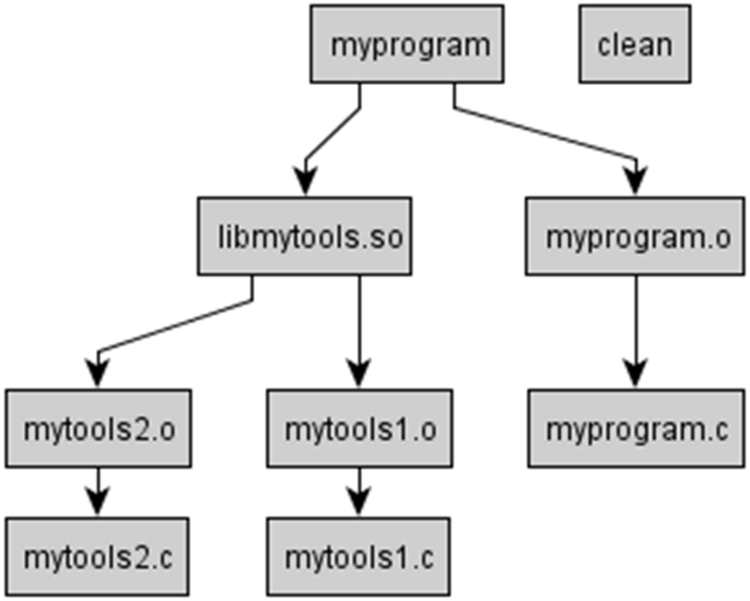
```
<targetfile1>:  <dependfile1> <dependfile2>
   <command to create targetfile1>
   <command to create targetfile1>

<targetfile2>: <dependfile3> <dependfile4>
   <command to create targetfile2>…
```

Simple Structure of a Makefile

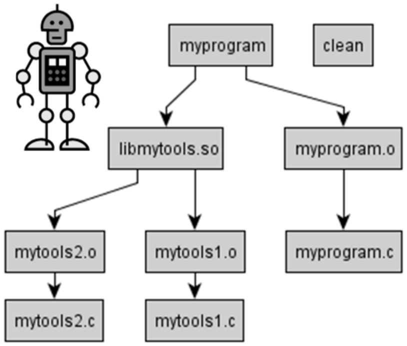
- Textual representation of a graph:

```
myprogram: libmytools.so myprogram.o
        …
libmytools.so: mytools2.o mytools1.o
        …
mytools1.o: mytools1.c
        …
mytools2.o: mytools2.c
        …
myprogram.o: myprogram.c
        …
Clean:
        rm –rf *.o *.so myprogram
```



Simple Structure of a Makefile

- Typing
  - `> make <targetfile>`
  - Tries to create <targetfile>
    - In particular, it first checks all the things <targetfile> depends on
      - If any don’t exist, they are created
      - If any are older than things they depend on, they are re-generated



Topic 2: Static Analysis

- Before we get to testing (our first big main topic) want to discuss another method for finding bugs
    - Analyze the source code for bad “patterns”
    - Happens to some extent every time you build a program
        - Your compiler has to analyze the code to compile and optimize it
        - `gcc –Wall` will warn you about some problems that might show up in testing

What is Static Analysis?

- Called “static” analysis because it analyzes your program without running it
  - Analysis that runs the program is called “dynamic” analysis (testing is the most common dynamic analysis)
- Differs in a few key ways:
  - Static analysis can catch bugs without a test case – just by structure of code
  - Static analysis can give “false positives” – warn you about a problem that can’t actually show up when the program runs

Static Analysis: Not Just Compilers

- While the compiler does some limited “bug hunting” during compilation, that’s not its main job
  - There are dedicated tools for analyzing source code for bugs
  - A few such tools include:
    - Uno (open source, available on the web)
    - Coverity (paid software, quite pricey but very powerful, used by NASA and others)
    - Klocwork
    - CodeSonar
- Won’t say much about these in this class, because they are typically fairly easy to use, just run them on your code

Static Analysis: Not Just Compilers

- Testing, on the other hand, requires more work from the programmer/test engineer
- So why not prefer static analysis in general?
  - Static analysis is generally limited to simple properties – don’t reference null pointers, don’t go outside array bounds
  - Also good for some security properties
  - But very hard/impossible to check things like “this sort routine really sorts things”