CSE 391
Lecture 8

Large Program Management: Make; Ant

slides created by Marty Stepp, modified by Jessica Miller and Ruth Anderson
http://www.cs.washington.ed0u/391/

Motivation

- single-file programs do not work well when code gets large
  - compilation can be slow
  - hard to collaborate between multiple programmers
  - more cumbersome to edit

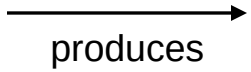
- larger programs are split into multiple files
  - each file represents a partial program or *module*
  - modules can be compiled separately or together
  - a module can be shared between multiple programs

- but now we have to deal with all these files just to build our program…

Compiling: Java

- What happens when you **compile** a Java program?

```
$ javac Example.java
```



`Example.class`

Answer: It produces a .class file.

- `Example.java` is compiled to create `Example.class`

- How do you **run** this Java program?

```
$ java Example
```

Compiling: C

command	description
`gcc`	GNU C compiler

- To **compile** a C program called ***`source`*`.c`**, type:

  `gcc -o` ***`target source`*`.c`** → produces → ***`target`***

  (where ***target*** is the name of the executable program to build)

  - the compiler builds an actual *executable file* (not a `.class` like Java)
  - Example: `gcc -o hi hello.c`

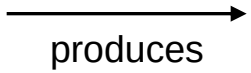
    Compiles the file `hello.c` into an executable called "`hi`"

- To **run** your program, just execute that file:
  - Example: `./hi`

Object files (.o)

- A .c file can also be **compiled** into an *object (.o) file* with **`-c`** :

```
$ gcc -c part1.c
$ ls
part1.c   part1.o   part2.c
```

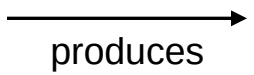


`part1.o`

  - a `.o` file is a binary "blob" of compiled C code that cannot be directly executed, but can be directly **linked** into a larger *executable* later

- You can **compile** and **link** a mixture of `.c` and `.o` files:

```
$ gcc -o myProgram part1.o part2.c
```



`myProgram`

Avoids recompilation of unchanged partial program files (e.g. **`part1.o`**)

Header files (.h)

- **header** : A C file whose only purpose is to be #included
  (#include is like java import statement)
  - generally a filename with the **.**h extension
  - holds shared variables, types, and function declarations
  - similar to a java interface: **contains function *declarations* but *not implementations***

- key ideas:
  - every ***name*.**c intended to be a module (not a stand alone program) has a ***name*.**h
  - ***name*.**h declares all global functions/data of the module
  - other **.**c files that want to *use* the module will #include ***name*.**h

Compiling large programs

- Compiling *multi-file* programs repeatedly is cumbersome:

```
$ gcc -o myprogram file1.c file2.c file3.c
```

- Retyping the above command is wasteful:
  - for the developer (so much typing)
  - for the compiler (may not need to recompile all; save them as `.o`)

- Improvements:
  - use up-arrow or history to re-type compilation command for you
  - use an alias or shell script to recompile everything
  - use a system for compilation/build management, such as `make`

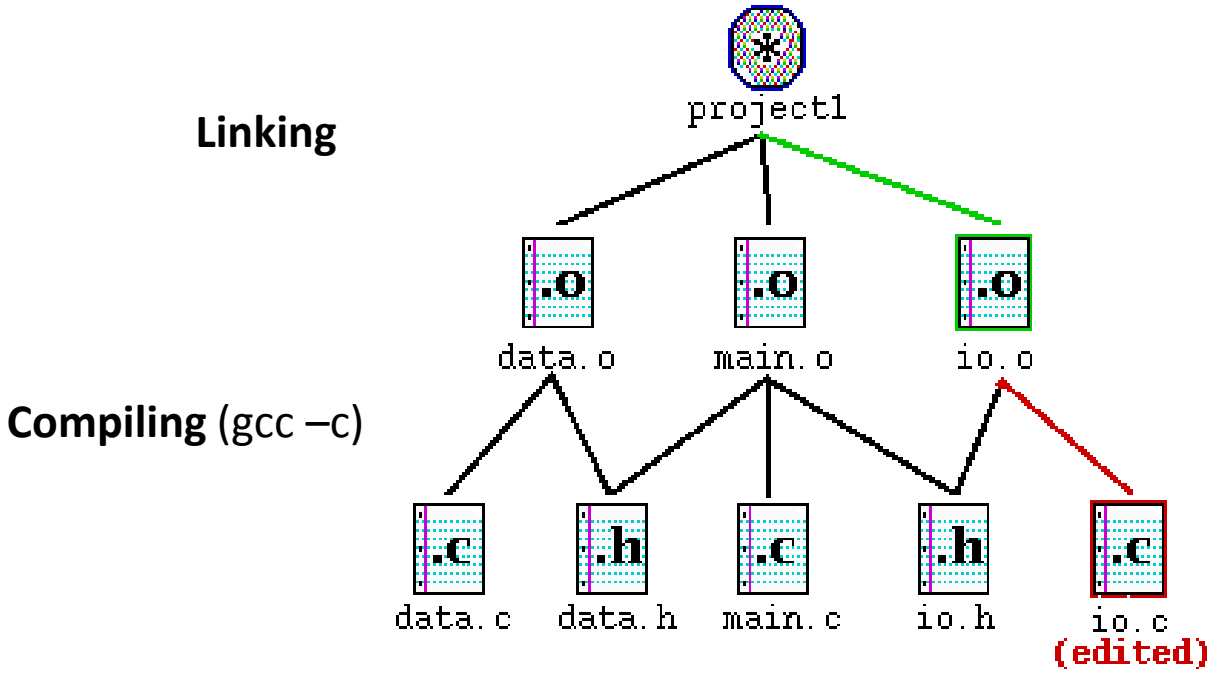
`make`

- **`make`** : A utility for automatically compiling ("building") executables and libraries from source code.
  - a very basic compilation manager
  - often used for C programs, but not language-specific
  - primitive, but still widely used due to familiarity, simplicity
  - similar programs: `ant`, `maven`, IDEs (Eclipse), ...

- **Makefile** : A script file that defines rules for what must be compiled and how to compile it.
  - Makefiles describe which files depend on which others, and how to create / compile / build / update each file in the system as needed.

Dependencies

- **dependency** : When a file relies on the contents of another.
  - can be displayed as a *dependency graph*
  - to build `main.o`, we need `data.h`, `main.c`, and `io.h`
  - if any of those files is updated, we must rebuild `main.o`
  - if `main.o` is updated, we must update `project1`



Makefile rule syntax

```
target : source1 source2 ... sourceN
        command
        command
        ...
```

- ***source1*** through ***sourceN*** are *dependencies* for building ***target***
- A ***source*** is a file that is used as input to create the target.
  (Sources are sometimes called *prerequisites*.)
- A ***target*** often depends on several ***sources***.
- Make will execute the ***command***s in order
  (Newer GNU documentation calls the commands a *recipe*.)

Example:

```
myprogram : file1.c file2.c file3.c
        gcc -o myprogram file1.c file2.c file3.c
```

this is a tab THIS IS NOT spaces!!

- The ***command*** line must be indented by a single tab
  - not by spaces; ***NOT BY SPACES!*** SPACES WILL NOT WORK!

Running `make`

`$ make` ***`target`***

- uses the file named `Makefile` in current directory
- Finds a rule in Makefile for building ***target*** and follows it
  - if the ***target*** file does not exist, or if it is older than any of its *sources*, its *commands* will be executed

- variations:

`$ make`

- builds the ***first*** target in the Makefile by default

`$ make -f` ***`makefilename`***
`$ make -f` ***`makefilename target`***

- uses a makefile other than `Makefile`

How does make process rules?

```
myprogram : file1.c file2.c file3.c
        gcc -o myprogram file1.c file2.c file3.c
```

When you type "***make***":

- if `myprogram` **does not exist**, then it will execute the command:
  `gcc -o myprogram file1.c file2.c file3.c`
  (Which will create a file called "myprogram")

- if `myprogram` **does exist** , then its timestamp will be compared to the timestamps on `file1.c, file2.c,` and `file3.c`
  If any of the sources have been modified since `myprogram` was created, the gcc command will be run to update `myprogram.`

- If any of the sources were the target of another rule in the makefile (e.g. file1.o), then the timestamps on ***its*** dependencies would be checked recursively and rebuilt before rebuilding `myprogram`

Another example

```
aprogram : foo.o bar.o
        gcc -o aprogram foo.o bar.o

foo.o: foo.c
        gcc -c foo.c

bar.o: bar.c
        gcc -c bar.c
```

When you type "***make***":

- `foo.o` and `bar.o` will be updated if needed (their rules will be consulted and the timestamps on `foo.c` and `bar.c` will be checked).
- if `aprogram` **does exist**, then its timestamp will be compared to the timestamps on `foo.o` and `bar.o`
  If `foo.o` and `bar.o` have been modified since `aprogram` was created, the gcc command will be run to update `aprogram`.

Making a Makefile

- **Exercise**: Create a basic Makefile to build project1
  - Basic works, but is wasteful. What happens if we change io.c?
    - everything is recompiled. On a large project, this could be a huge waste

Making a Makefile

- **Exercise**: Create a basic Makefile to build project1
  - Basic works, but is wasteful. What happens if we change io.c?
    - everything is recompiled.
    - On a large project, this could be a huge waste
  - Augment the makefile to make use of precompiled object (.o) files and dependencies
    - by adding additional targets, we can avoid unnecessary re-compilation

Rules that don't create their target

```
# A rule that creates its target
myprog: file1.o file2.o file3.o
	gcc -o myprog file1.o file2.o file3.o

# This rule does not create a file named "clean"
clean:
	rm file1.o file2.o file3.o myprog
```

- make assumes that a rule's command will build/create its target
  - *If the target is not present, make will execute the commands!*
  - *Otherwise it compares the timestamps of the target to its dependences, recursively checking timestamps on the dependencies dependencies etc.*
  - *if your rule does not actually create its target, the target will never exist*, so the rule will always execute its commands (e.g. `clean` above)
  - `make clean` is a common convention for removing all generated files
  - Other things you may see in makefiles:

http://www.gnu.org/software/make/manual/html_node/Phony-Targets.html#Phony-Targets

http://www.gnu.org/software/make/manual/html_node/Force-Targets.html#Force-Targets

Rules with no commands

```
# This rule has no commands
all: myprog myprog2

myprog: file1.o file2.o file3.o
	gcc -o myprog file1.o file2.o file3.o

myprog2: file4.c
	gcc -o myprog2 file4.c
...
```

- The `all` rule has no commands, but depends on `myprog` & `myprog2`
  - typing `make all` will ensure that `myprog`, `myprog2` are up to date because their dependencies will be checked recursively and updated as needed
  - Having an `all` rule is also a common convention, and is often put as the first rule in a file, so that just typing `make` without giving a target will build "everything" (the dependencies of `all`)

Variables

NAME = ***value*** (declare)
$(***NAME***) (use)

Example Makefile:

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog

$(PROGRAM): $(OBJFILES)
        gcc -o $(PROGRAM) $(OBJFILES)

clean:
        rm $(OBJFILES) $(PROGRAM)
```

- variables make it easier to change one option throughout the file
  - also makes the makefile more reusable for another project

More variables

Example Makefile:

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
CC = gcc
CFLAGS = -g –Wall

$(PROGRAM): $(OBJFILES)
        $(CC) $(CFLAGS) -o $(PROGRAM) $(OBJFILES)
```

- many makefiles create variables for the compiler, flags, etc.
  - this can be overkill, but you will see it "out there"

Special variables

$@	the current target file
$^	all sources listed for the current target
$<	the first (left-most) source for the current target
	(there are other special variables*)

Example Makefile:

```
myprog: file1.o file2.o file3.o
        gcc $(CFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
        gcc $(CFLAGS) -c $<
```

- **Exercise**: change our hello Makefile to use variables for the object files and the name of the program

*http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables

Pattern Rules

- Rather than specifying individually how to convert every `.c` file into its corresponding `.o` file, you can make use of pattern rules:

```
# conversion from .c to .o
%.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

  - This means: "To create *filename*.o from *filename*.c, run gcc -c -g -Wall *filename*.c -o *filename*.o "
  - **The rule listed above is actually pre-defined in make**, so you do not need to include it in your makefile. You can add additional dependencies by adding rules with no commands:

```
main.o: io.h data.h
data.o: data.h

# Note, we can omit data.c as a source for data.o as long as
  we do not list commands for creating data.o
```

https://www.gnu.org/software/make/manual/html_node/make-Deduces.html#make-Deduces

Old-Fashioned Suffix Rules

- In older Makefiles, you may see rules like the one below, which have a similar effect:

```
.c.o:
        $(CC) –c $(CFLAGS) -c $< -o $@
```

- This means: “To create *filename*.o from *filename*.c, run gcc -c -g -Wall -c *filename*.c -o *filename*.o “

Aside: make Exercise

- **figlet** : program for displaying large ASCII text (like `banner`).
  - http://freecode.com/projects/figlet

- Download a piece of software and compile it with make:
  - download `.tar.gz` file
  - un-`tar` it
  - look at `README` file to see how to compile it
  - (sometimes) run `./configure`
    - for cross-platform programs; sets up `make` for our operating system
  - run **`make`** to compile the program
  - execute the program

What about Java?

- Create Example.java that uses a class MyValue in MyValue.java
  - Compile Example.java and run it
    - javac automatically found and compiled MyValue.java
  - Now, alter MyValue.java
    - Re-compile Example.java… does the change we made to MyValue propagate?
    - Yep! javac follows similar timestamping rules as the makefile dependencies. If it can find both a .java and a .class file, and the .java is newer than the .class, it will automatically recompile
    - But be careful about the depth of the search...
- But, this is still a simplistic feature. Ant is a commonly used build tool for Java programs giving many more build options.

Ant

- Similar idea to Make
- Ant uses a **`build.xml`** file instead of a Makefile

```
<project>
     <target name=“name”>
          tasks
     </target>

     <target name=“name”>
          tasks
     </target>
</project>
```

- Tasks can be things like:
  - `<javac  … />`
  - `<mkdir … />`
  - `<delete … />`
  - A whole lot more…http://ant.apache.org/manual/tasksoverview.html

Ant Example

- Create an Ant file to compile our Example.java program

- To run ant (assuming `build.xml` is in the current directory):

```
$ ant targetname
```

- For example, if you have targets called clean and compile:

```
$ ant clean
$ ant compile
```

Refer to: http://ant.apache.org/manual/tasksoverview.html
for more information on Ant tasks and their attributes.

Example build.xml file

```
<!-- Example build.xml file -->
<!-- Homer Simpson, cse391 -->
<project>
   <target name="clean">
      <delete dir="build"/>
   </target>

   <target name="compile">
      <mkdir dir="build/classes"/>
      <javac srcdir="src" destdir="build/classes"/>
   </target>
</project>
```

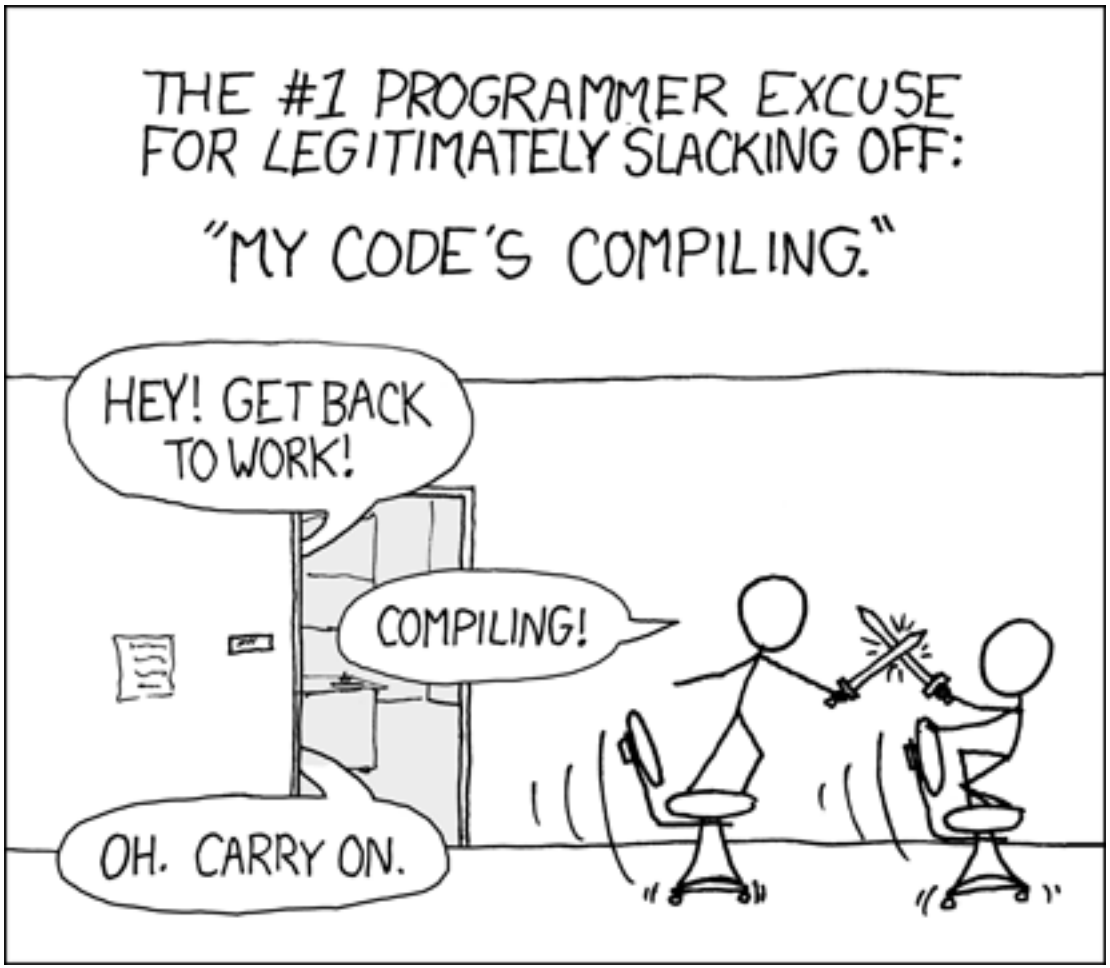
Some useful Ant tasks

- An overview:
  - http://ant.apache.org/manual/tasksoverview.html
- Compiling java programs:
  - http://ant.apache.org/manual/Tasks/javac.html
- Running java programs:
  - http://ant.apache.org/manual/Tasks/java.html
- Creating directories:
  - http://ant.apache.org/manual/Tasks/mkdir.html
- Deleting files:
  - http://ant.apache.org/manual/Tasks/delete.html

Automated Build Systems

- Fairly essential for any large programming project
  - Why? Shell scripts instead? What are these tools aiming to do?
  - Is timestamping the right approach for determining “recompile”?
  - What about dependency determination?
  - What features would you want from an automated build tool?
  - Should “building” your program also involve non-syntactic checking?
    - Ant can run JUnit tests…

Making a Makefile



courtesy XKCD