Makefiles

- May have to deal with projects with many .h, .cpp files, multiple different executables, and complex include heirarchies
- Want to ensure that, after editing something, all the affected files are updated correctly
- Don’t want to needlessly recompile files that aren’t affected
- Need a mechanism to define which files need to be recompiled based on what has changed, and how to recompile them

Makefile concept

- Each item that can be rebuilt is called a target (e.g. each executable and each .o file)
- For each target, we provide a list of file dependencies, if any of these files have changed then the file needs to be rebuilt
- For each target, we provide a rule specifying exactly how to rebuild it
- Makefile data is put in a file named makefile, by default one makefile per directory (with targets for everything in the directory)
- To rebuild a target from the command line, we type

  make *targetname*

Sample makefile

- Suppose we have stack.h and stack.cpp to define and implement a stack data type, and a program myprog.cpp with header file myprog.h, which utilize the stack code

```
# sample makefile (note: the # is used for comments)
myprogx: myprog.o stack.o
	g++ myprog.o stack.o -o myprogx

myprog.o: myprog.cpp myprog.h stack.h
	g++ -c myprog.cpp -o myprog.o

stack.o: stack.cpp stack.h
	g++ -c stack.cpp -o stack.o
```

Makefiles and tab syntax

- Note that the name of the target begins a line, and the first character on the next line (the line with the rebuild rule) ***MUST*** be a tab ... anything else (even spaces instead of a tab) will result in make yelling at you when it runs
- If your editor automatically substitutes spaces for tabs then you’ll need to turn that off while editing makefiles

Using the makefile

- We could manually specify what to rebuild from the command line, e.g. “make stack.o”, but usually we specify the executable and let make figure out the details, e.g. “make myprogx”
- If we simply type “make” then it assumes you mean the top target in the makefile (i.e. myprogx in this case)

How the makefile works

- For current target, make goes throughlist of dependencies:
  - if any have their own rules in the makefile it treats them as intermediate targets, processing them then coming back
- After processing all of target’s dependencies, if any of the dependency files have changed more recently than the target then it rebuilds target using the associated rule
- Make uses the file modification date to determine what is most recent (compare target date/time to dependency date/time)
- If a target doesn’t exist (e.g. if this is first time compiling) then it automatically builds it

Using our stack/myprog example

- Suppose we had compiled everything for myprogx, then we edit myprog.cpp and run “make progx” to rebuild
- Make looks at progx dependencies, myprog.o and stack.o, sees both have rules, goes off to check them
- Checks stack.o rule, looks at its dependencies, stack.cpp and stack.h – neither of them have changed
- Checks myprog.o rule, looks at its dependencies, myprog.cpp and myprog.h, myprog.cpp has changed, runs g++ -c myprog.cpp -o myprog.o
- Now back at myprogx, sees a dependency has changed, so runs g++ myprog.o stack.o -o myprogx

Echo and @

- Can generate output statements inside a makefile using echo, e.g.

```
myprog.o: myprog.cpp myprog.h stack.h
        echo “myprog.o needs updating”
        g++ -c myprog.cpp -o myprog.o
```

- While make runs, it prints each statement before executing it, so if the above target ran we’d see

```
echo “myprog.o needs updating”
Myprog.o needs updating
g++ -c myprog.cpp -o myprog.o
```

- We can “silence” a command by putting @ in front of it (e.g. @echo) in which case make doesn’t display the command before running it

Phony targets

- If we have actions we want to include in a makefile that don’t involve actually building anything, we can use a “phony” target to run them
- Suppose I want to use “make clean” to remove a set of .o files and executables, I could do the following:

```
.PHONY: clean
clean:
    rm -f stack.o myprog.o myprogx
```

“All” as a first target in file

- Suppose we have multiple executables in our directory, so our makefile has rules for each
- We’d like to be able to bring them all up to date by simply typing “make”
- We could add a target at the top of the file that has each of them as a dependency, e.g. all: progx testerx whateverx
- Since make uses top target by default, and checks rules for each dependency, this does exactly what we want

“touch” and testing makefiles

- Of course, it’s entirely possible we make mistakes when creating our sets of rules and dependencies for a makefile, so it’s a good idea to test our makefiles
- We need to figure out which rules should run after editing a file, edit the file, run make, and see if it works
- Of course, editing a file when we don’t actually want to change it is risky, so we use “touch” instead
- From the command prompt, “touch filename” simply changes the modification date of the file to now

Variables in makefiles

- We can create and use variables in makefiles, for much the same reasons as in other forms of coding
- e.g. suppose we want to use g++ options “-Wall -Wextra -O” in all of our g++ commands – we could type them in for every rule, but that’s error prone and difficult to update everywhere if we change our option list later
- Instead, we put them in a variable and use that, e.g.

```
Options=-Wall -Wextra -O
....
g++ ${Options} -c foo.cpp -o foo.o
```

Overriding variable values

- When running make from the command line, we can specify a different value to use for a variable, e.g. suppose one time we want to compile with -g instead of -Wall -Wextra -O
- To do so, we specify the variable value as an argument for make, e.g.

```
make Options=-g
```

Special variables in make

- There are a number of special variables in make, here we’ll just consider two of them
- $@ is a variable referring to the name of the target
- $< refers to the first dependency in the list

```
Prog.o: prog.cpp prog.h
    g++ -c prog.cpp -o prog.o
```

- Equivalent to:

```
Prog.o: prog.cpp prog.h
    g++ -c $< -o $@
```

We’ve only scratched the surface

- There is much much much more functionality with make, and ways we can create far more flexible rules
- Other features of make, and other programs, exist to generate makefiles automatically from the heirarchy of #includes
- IDEs essentially do all this “behind the scenes” for you in order to automate compilation