Makefiles

- a program can be made up of many different .h and .cpp files
- we want a way to ensure that, after editing something, all the affected files are updated correctly
- we don’t want to needlessly recompile files that aren’t affected
- makefiles give us a way to identify what needs to be recompiled based on which files have been edited, and to specify 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`

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

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