```
CS 214 / 2021-02-15
===================


making make make with makefiles
-------------------------------

Make is a tool that we can use to automate running programs
-> primarily to simplify compiling programs

You provide make a set of rules in a file called a "make file"
-> tells make how to create certain files / do certain tasks
-> traditionally a file named "Makefile"
        -> capital M means it appears earlier in the alphabetical listing
        -> make has a list of "default" file names that it will look for
        -> you can explicitly tell make which file to use as a make file

using make

        $ make

                <- tells make to look at Makefile and update the first target

        $ make some_target
        $ make target1 target2 target3...

                <- tell make to update specified target(s)

        $ make -B [targets...]

                <- tells make to update targets, ignoring modification dates

        $ make -j N [targets...]

                <- tells make to run on N processors simultaneously (if possible)
                <- not necessarily useful for us


rules

[target name]: [zero or more dependencies]
        [recipe: shell commands]

NOTE: recipe must be indented using a single tab (\t) character


Example:

        program: program.c
                gcc program.c -o program

This describes a rule to make "program"
        - "program" depends on "program.c"
                - ie., if program.c changes, then program is out of date
        - recipe says how to create program from program.c

        - to run:
                $ make program

How make treats this rule
        - when we tell make to make program, it will check whether it needs to
                update the file "program"

        Process:
        1. Recursively update all dependencies of the target
        2. Check whether the target exists
        3. If so, check whether the target is older than its dependencies
        4. If target does not exist or is older than its dependencies (or we used -B)
                make performs the recipe


        Idea: only perform the recipe if we need to
                - but the recursive check ensures that files get rebuilt as needed

Example:
        demo: demo.o arraylist.o
                ...
        demo.o: demo.c arraylist.h
                ...
        arraylist.o: arraylist.c arraylist.h
                ...

        After we build demo, we will have
                arraylist.h arraylist.c demo.c
        older than
                arraylist.o demo.o
        older than
                demo

        I modify demo.c, so now it is newer than demo

                demo.c < demo < demo.o

        When I do "make demo", make will update demo
                Recursively update demo.o
                        demo.o is older than demo.c, so we rebuild demo.o
                Recursively update arraylist.o
                        arraylist.o is fresher than arraylist.c and arraylist.h, so do nothing
                Now, demo.o is fresher than demo, so rebuild demo


        Note that demo.o and arraylist.o both depend on arraylist.h
                Why?

                        presumably both demo.c and arraylist.c include arraylist.h
                        changes to arraylist.h may require recompiling

                -> make does not track dependencies automatically; you must specify
                        them yourself
                        (or use a program to track them)

----
Why might changing a header require recompilation?

Let's say I have
        typedef int foo_t;

If I change this to
        typedef unsigned int foo_t;

Any code doing math with a foo_t value may need to change
        e.g., signed vs unsigned division instructions



---

Special cases:

Rules do not require dependencies

For example,

        clean:
                rm -f list of files....


        <- doesn't depend on anything, so no need for recursive check
        <- doesn't create a file named "clean", so it will always execute
                "pseudo-rule"

        <- note that we can perform any command, not just compiling
                <- we use -f to prevent rm complaining if we try to delete something that
                   doesn't exist

Other non-compiling uses

        test: demo
                ./demo some_input

        <- re/compile demo if needed (update target demo using rule)
        <- call demo with specified argument(s)

Note:
        make checks the exit status of the commands it performs
                if a program does not return EXIT_SUCCESS, the rule fails

        So if we "make test" and GCC reports an error when compiling "demo",
                make will stop (i.e., not try to run "demo")

        -> this is one of the reasons we have exit statuses in our programs
                tell whoever ran the program whether the program succeeded

Make variables

VARIABLE_NAME = some text

        <- declares a variable
        <- variable names do not have to be all-caps, but this is common

Make will substitute $(VARIABLE_NAME) with the value of the variable

        CC = gcc
        CFLAGS = -g -Wall

        demo: demo.o arraylist.o
                $(CC) $(CFLAGS) -o $@ $^


        as though we had written
                "gcc -g -Wall -o demo demo.o arraylist.o"

Make has a bunch of special variables
        $@  -  the name of the target of that rule
        $^  -  the dependency list for that rule
        $<  -  the first dependency for that rule

Why use these?
        - save typing
        - define rules in a more general way
        - make changes in a centralized way (e.g., adjust CFLAGS)
        - override variable declarations when we call make

                make demo CC=clang

                        <- will compile using clang instead of gcc

        - other fancy stuff when using multiple directories

-> but these are all optional; you are never required to use variables

Rule schemes
        - fast way to define a bunch of rules at once

        %.o: %.c
                gcc -c $<


        This says: to make any file ending with .o, run this command;
                it says that FILE.o depends on FILE.c
                the command is "gcc -c FILE.c"


        We can override this for specific files by writing an explicit rule

        # general rule
        %.o: %.c
                $(CC) -c $(CFLAGS) $<

        # specific rule for one file
        special.o: special.c
                $(CC) -c $(CFLAGS) -other-flag $<


Note: make has a default rule for %.o files that it will use if you don't
        specify one

Why use $< instead of $^ ?

        Recall that .o files can depend on .h files as well
        We don't want to include the .h files in the argument to gcc

How can we specify .h files if we are using general rules?

        We can declare additional dependencies with rules that don't have a recipe


        %.o: %.c
                $(CC) -c $(CFLAGS) $<

        demo.o: arraylist.h
        arraylist.o: arraylist.h

        -> now demo.o will depend on demo.c and arraylist.h


Make is not specific to any language

We could do something like this

        %.class: %.java
                javac $<

        # additional options may be required (I haven't used javac in a long time)


Note: target and dependencies can be paths

        lib/foo.o: src/foo.c
                ....


Try stuff out!


input and output
----------------

https://www.gnu.org/software/libc/manual/html_node/I_002fO-Overview.html


How do we communicate with the outside world from our program?
        print to screen
        read from keyboard
        read/write files
        send/receive network messages
        communicate with other processes

In Unix/C the answer to all of these are "files"
        in general, a file is a stream of bytes that we can read from and/or write to

        -> files may be actual files stored on a disk, but we reuse the abstract
                inferface for all I/O

This is not how older languages/operating systems worked
        Older file systems were based on "records"
                text files were weird/hard to work with
        different functions/system calls for different kinds of communication

In C/Unix, we use the same general model for all communication
        possibly with extra features for certain types
                e.g., we can "rewind" files or jump to specific places


We have two sets of functions for working with files in most C environments

- "streaming" functions specified by C standard
        obtained from <stdio.h>
        use FILE * (file pointer)
        function names usually start with f
                fopen, fclose, fread, fwrite, ...

- low-level system calls specified by Posix
        obtained from various places <unistd.h>, <fcntl.h>, and others
        use file descriptors (just an integer)
        open, close, read, write, ...


In this class, we will focus on the low-level system calls

In general, you will want to use the streaming functions most of the time
        - more portable (C standard library)
        - potentially fewer system calls

Next time:
        using file descriptors
        opening files
        reading and writing
        converting between file descriptors and file pointers
```