CMSC 216
Introduction to Computer Systems
Lecture 9
Make

How to run make

- Just executing the command "**`make`**" will build the first target in the current directory's makefile
- Makefiles should be named either "**`makefile`**" or "**`Makefile`**"
- The command "**`make` *`target-name`***" will build the target ***`target-name`*** instead of the first target in the makefile

Modification times

- Targets are only built if their dependencies have been updated, using modification times
- **`ls`** can display file modification times:
  - **`ls -l`** list files showing their modification time, size, etc.
  - **`ls -lt`** list files in order of mod. time (most recent first)
  - **`ls -lrt`** invert the sort order of a listing
- How are file modification times changed?
  - Changing contents of a file in any way, via a text editor or any other means, adjusts the file's mod. time
  - UNIX "**`touch`**" command can be used to set file mod. time if you need to do it manually
    - default is to use current time, but can use any date/time
    - be careful with it – can cause unnecessary compilations, or prevent necessary ones, and usually it's not needed

Make's operation

- Make constructs a dependency tree based on the makefile
- A target is out of date when any one of its (direct or indirect) dependencies is newer than itself
- If a target is out of date, make recreates it by performing the action specified in that target's action
  - this process is performed bottom to top in the dependency tree
- Make ensures minimum compilation, if your makefile and dependencies are correct
- Will the following rule make **`publicX`**?

```
publicX: publicX.c puzzles.c puzzles.h
        gcc -o publicX publicX.c puzzles.c
```

- Yes, but is it desirable? We rebuild the entire program no matter which source file is changed with this rule.

Makefile macros

- A macro is a makefile variable
- A macro is defined by: *variable* = *value*

```
CC = gcc
CFLAGS = -ansi -pedantic-errors -Wall -Werror

file.o: file.c file.h
        $(CC) $(CFLAGS) -c file.c
```

  – Last rule has the effect as if it were written:

```
file.o: file.c file.h
        gcc -ansi -pedantic-errors -Wall -Werror -c file.c
```

- Note that the above options for **`CFLAGS`** only affect the way that code is compiled, not linking (so they should be omitted for rules that just do linking)

Additional features of make

- Implicit rules
  - Make understands conventions
    - **`xxx.o`** is built from **`xxx.c`** using the action "**`$(CC) $(CFLAGS) -c xxx.c`**"
  - Has lots of rules it knows about
    - With a simple, single-file program (e.g., **`helloworld.c`**), you can build your program with a command like this:

      `make helloworld`
  - These are useful, but there's a lot to know about using them correctly

More make features

- Actions can be any shell commands, even multiple commands, not just compilation/linking

```
clean:
        @echo "Removing object, executable, and core files."
        rm -f *.o *~ core core.*
```

- Preceding an action line with an at sign (`@`) keeps make from echoing the command when make is run
- A long line in a makefile can be continued to the next line by ending it with a backslash

A more complex makefile

```
# comments start with a pound sign and last until the end of the line

CC = gcc
CFLAGS = -ansi -pedantic-errors -Wall -Werror
PROGS = int_count table

all: $(PROGS)

int_count: int_count.o
        $(CC) -o int_count int_count.o

table: table.o main.o hash_table.o
        $(CC) -o table table.o main.o hash_table.o

table.o: table.c table.h
hash_table.o: table.h
main.o: table.h
int_count.o: table.h

clean:
        rm -f *.o $(PROGS)
```

macro definitions

builds the two top-level programs

note no `$(CFLAGS)` in these

use implicit rules

not part of the main tree - built by typing "make clean"

More about header files

- Note that header files should contain only declarations, with no executable code
- Header files are not directly compiled (for example, there were no rules to compile any `.h` files in the previous example); they just get included in source files, which are compiled
- Header files should contain declarations that the compiler needs to see to be able to compile executable code
  - examples include function prototypes, typedefs, and **`extern`** global variables

Chapter 11, Reek

Dynamic Memory Allocation

Stack vs. Heap Memory

- Stack memory is allocated and deallocated in a specific order
  - you can only remove from and add to the top of the stack
  - any item in the stack is always freed before the item below it
  - useful for local variables, since functions work in similar manner
- Heap memory
  - can be allocated and deallocated in any order while the program is running

Dynamically allocated memory

- Why use it?
  - to eliminate compile-time limits:
    - Often, the size of a data structure isn't known until runtime
    - Allocating a huge amount of space to fit every possible scenario can be error-prone at worst and a waste of space at best
- User-managed heap vs. Garbage collection
  - garbage collection:
    - frees programmer from burden of keeping track of objects (convenience)
    - slows down execution due to collection needing to be run periodically; collection is also expensive
  - user-managed heap
    - programmer can control the precise time to deallocate objects
    - requires a lot more care - errors can occur if the wrong thing is freed, or the right thing is freed at the wrong time

Memory management functions

- Allocating memory:

  `void *malloc(size_t amount);`

  - allocates **`amount`** bytes of memory (if available) from the heap and returns a pointer to the beginning of it
  - no initialization of the space

  `void *calloc(size_t count, size_t obj_size);`

  - allocates **`count`** objects of size **`obj_size`** each (if memory is available), returns a pointer to beginning of it
  - initializes all the space to 0

  – both return **`NULL`** if the allocation fails

  – both require **`#include <stdlib.h>`** since prototypes are contained there

- Note that both these functions return a **`void *`**

Memory management functions, cont.

- Deallocating memory

```
void free(void *ptr);
```

  - returns the memory pointed to by **`ptr`** to the free pool in the heap
  - memory MUST have been previously allocated from the heap
  - does not change **`ptr`** (why not?), so you may want to set **`ptr`** to **`NULL`** after calling **`free()`** if you might accidentally use **`ptr`** again
  - **`free(NULL);`** does nothing - it's perfectly OK to do

More about `free()`

- Once you've freed memory, it goes back to the heap, meaning you can't trust its value to remain the same (or even stay valid!)
- It's up to you to ensure you don't dereference a freed pointer - otherwise, weird things can happen ...