Make

Class 20

Modularity

- we have seen the advantages of modularity
- but with modularity comes an explosion of files
- tens or hundreds of .c and .h files
- depending on each other in a complex web
- even for just 5 files, up-arrowing gets tedious really quickly

Dependencies

- more importantly, there is a web of dependencies

File	Depends On
mycrypt.h	nothing
mycrypt.c	mycrypt.h
options.h	nothing
options.c	options.h, mycrypt.h
main.c	options.h, mycrypt.h

- if one file is changed, what happens?
- e.g., the options module is changed, what needs to be re-compiled?

Make

- universally, the process of managing dependencies and selectively recompiling is automated
- some languages make the process easy
- Java's absolute rule of one class per file, class name the same as the file name makes dependency tracking easy

- but C (like most languages) has no such convention, making the job harder

Make

- Unix has a powerful, flexible tool called make
- make depends on a specification file that the programmer builds
- this file is named Makefile (uppercase first letter, no extension)

- every time a new file is added to the project the Makefile is updated to state how the new file fits into the project's dependencies

- make is controlled by three things
  1. the rules listed in the Makefile
  2. does a needed file exist?
  3. the relative timestamps of a file that depends on another file

Rules

- the fundamental entry type in a Makefile is a rule

```
target: dependent list
<TAB>command
<TAB>command
<TAB> ...
```

- the tab character is absolutely essential (unlike C code)
- every code editor knows about the tab character at the beginning of a makefile command line

Rules

```
target: dependency-list
<TAB>command
<TAB>command
<TAB> ...
```

- target is interpreted as a filename
- dependency-list is a space-separated list
- each is a filename or the name of another target
- the target depends on all of things in dependency-list

Rules

```
target: dependency-list
<TAB>command
<TAB>command
<TAB> ...
```

- the command lines are run when the rule is triggered

- if the target has a timestamp older than any item in dependency-list
- or if the target does not exist
- then the rule is triggered and the commands are executed

A Rule

```
mycrypt: main.o options.o mycrypt.o
	clang -std=c89 -Weverything -pedantic-errors \
	  -o mycrypt main.o options.o mycrypt.o
```

- this rule says the file mycrypt depends on main.o, options.o, and mycrypt.o
  1. if any of these three files has a newer timestamp than mycrypt
  2. or if mycrypt does not exist
- then run the clang command (otherwise do not run it)
- note that in front of "clang" is a tab character, while in front of "-o" there is no tab character, only spaces for readability

A Rule

```
mycrypt: main.o options.o mycrypt.o
  clang -std=c89 -Weverything -pedantic-errors \
    -o mycrypt main.o options.o mycrypt.o
```

- mycrypt depends on main.o, options.o, and mycrypt.o
- what if one of them (say, options.o) does not exist?
- mycrypt depends on it!

- in this case, before make can trigger this rule, it must attempt to create the file options.o
- make searches the Makefile for a rule whose target is options.o
- make triggers that rule first, then returns to the current rule

A Complete Makefile

```
mycrypt: main.o options.o mycrypt.o
        clang -std=c89 -Weverything -pedantic-errors \
          -o mycrypt main.o options.o mycrypt.o

main.o: main.c mycrypt.h options.h
        clang -std=c89 -Weverything -Wno-padded \
           -pedantic-errors -c main.c

mycrypt.o: mycrypt.h mycrypt.c
        clang -std=c89 -Weverything -pedantic-errors \
          -c mycrypt.c

options.o: options.c options.h mycrypt.h
        clang -std=c89 -Weverything -Wno-padded \
          -pedantic-errors -c options.c
```

A Makefile

- this makefile has four rules
- the first rule is the default rule

- the command:
  ```
  $ make
  ```
  with no arguments invokes the first (default) rule in the file named “Makefile”
- to specifically invoke a different rule, it can be named on the command line:

  ```
  $ make options.o
  ```

  tries to find the rule named “options.o” and invokes it

Macros

```
mycrypt: main.o options.o mycrypt.o
        clang -std=c89 -Weverything -pedantic-errors \
          -o mycrypt main.o options.o mycrypt.o

main.o: main.c mycrypt.h options.h
        clang -std=c89 -Weverything -Wno-padded \
           -pedantic-errors -c main.c

mycrypt.o: mycrypt.h mycrypt.c
        clang -std=c89 -Weverything -pedantic-errors \
          -c mycrypt.c

options.o: options.c options.h mycrypt.h
        clang -std=c89 -Weverything -Wno-padded \
          -pedantic-errors -c options.c
```

- in the makefile there is a lot of repeated code
- we can simplify the makefile and make it more robust by defining some macros
- macros are defined at the top of the makefile

Makefile v2 with Macros

```
CC=clang
CFLAGS=-std=c89 -Weverything -pedantic-errors
OBJECTS=main.o options.o mycrypt.o

mycrypt: $(OBJECTS)
        $(CC) $(CFLAGS) -o mycrypt $(OBJECTS)

main.o: main.c mycrypt.h options.h
        $(CC) $(CFLAGS) -Wno-padded -c main.c

mycrypt.o: mycrypt.h mycrypt.c
        $(CC) $(CFLAGS) -c mycrypt.c

options.o: options.c options.h mycrypt.h
        $(CC) $(CFLAGS) -Wno-padded -c options.c
```

- make macros are similar to BASH variables

Dummy Targets

- targets can be “files” that don’t exist
- files that will never exist
- these allow you to force command execution
- using convenient names allows easy execution of common tasks
- a common dummy target is “clean”

```
clean:
        rm -f *.o *~ mycrypt
```

- clean has no dependencies, and since no file name “clean” exists, the commands are run unconditionally
- this causes all object files, all editor backup files, and the executable, if they exist, to be deleted, so nothing is left except the “clean” source code
- note that this will not work if a file named “clean” actually does exist

Another Dummy Target

- another extremely common traditional dummy target is “all”
- this is mainly to keep you from having to remember the exact name of the real “do it all” target

```
all: mycrypt
```

- “all” is a dummy target that depends on “mycrypt”
- it has no commands, so it just goes to “mycrypt” and executes those rules

Dummy Target: Test

- a final common dummy target is “test”
- this traditionally invokes multiple tests to see if your program is working correctly

```
test: mycrypt
        @echo Testing missing first switch
        ./mycrypt -5 roses.txt
        @echo Testing missing second switch
        ./mycrypt -e roses.txt
        @echo Testing good run
        ./mycrypt -e -120 roses.txt | ./mycrypt -d -120
```

Final Makefile

```
CC=clang
CFLAGS=-std=c89 -Weverything -pedantic-errors
OBJECTS=main.o options.o mycrypt.o

all: mycrypt

mycrypt: $(OBJECTS)
	$(CC) $(CFLAGS) -o mycrypt $(OBJECTS)

main.o: main.c mycrypt.h options.h
	$(CC) $(CFLAGS) -Wno-padded -c main.c

mycrypt.o: mycrypt.h mycrypt.c
	$(CC) $(CFLAGS) -c mycrypt.c

options.o: options.c options.h mycrypt.h
	$(CC) $(CFLAGS) -Wno-padded -c options.c

clean:
	rm -f *.o *~ mycrypt

test: mycrypt
	@echo Running tests
	./runtests.sh
```