CSE 303
Lecture 17

Makefiles

reading: *Programming in C* Ch. 15

slides created by Marty Stepp
http://www.cs.washington.edu/303/

The compilation process

- What happens when you compile a Java program?

```
$ javac Example.java
```

  - `Example.java` is compiled to create `Example.class`

- But...
  - what if you compile it again?
  - what if `Example.java` uses `Point` objects from `Point.java`?
  - what if `Point.java` is changed but *not* recompiled, and then we try to recompile `Example.java`?

Compiling large programs

- compiling multi-file programs repeatedly is cumbersome:

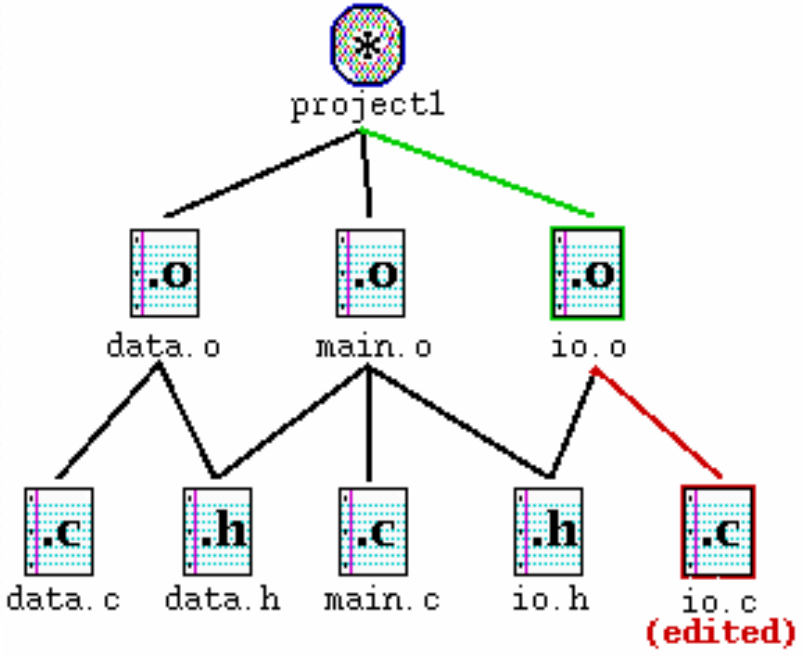
```
$ gcc -g -Wall -o myprogram file1.c file2.c file3.c
```

- retyping the above command is wasteful:
  - for the developer (so much typing)
  - for the compiler (may not need to recompile all; save them as `.o`)

- improvements:
  - use up-arrow or history to re-type compilation command for you
  - use an alias or shell script to recompile everything
  - use a system for compilation/build management, such as `make`

Dependencies

- **dependency** : When a file relies on the contents of another.
  - can be displayed as a *dependency graph*
  - to build `main.o`, we need `data.h`, `main.c`, and `io.h`
  - if any of those files is updated, we must rebuild `main.o`
  - if `main.o` is updated, we must update `project1`



make

- **make** : A utility for automatically compiling ("building") executables and libraries from source code.
  - a very basic compilation manager
  - often used for C programs, but not language-specific
  - primitive, but still widely used due to familiarity, simplicity
  - similar programs: ant, maven, IDEs (Eclipse), ...

- **Makefile** : A script file that defines rules for what must be compiled and how to compile it.
  - Makefiles describe which files depend on which others, and how to create / compile / build / update each file in the system as needed.

`make` demo

- **figlet** : program for displaying large ASCII text (like `banner`).
  - http://sourceforge.net/projects/freshmeat_figlet/

- Let's download a piece of software and compile it with make:
  - download `.tar.gz` file
  - un-`tar` it
  - (optional) look at `README` file to see how to compile it
  - (sometimes) run `./configure`
    - for cross-platform programs; sets up `make` for our operating system
  - run `make` to compile the program
  - execute the program

Makefile rule syntax

```
target : source1 source2 ... sourceN
        command
        command
        ...
```

- Example:

```
myprogram : file1.c file2.c file3.c
        gcc -o myprogram file1.c file2.c file3.c
```

- The ***command*** line must be indented by a single tab
  - not by spaces; ***NOT BY SPACES!*** SPACES WILL NOT WORK!

Running `make`

`$ make` ***`target`***

- uses the file named `Makefile` in current directory
- finds rule in Makefile for building ***target*** and follows it
  - if the ***target*** file does not exist, or if it is older than any of its *sources*, its *commands* will be executed

- variations:

`$ make`

- builds the *first* target in the Makefile

`$ make -f` ***`makefilename`***
`$ make -f` ***`makefilename target`***

- uses a makefile other than `Makefile`

Rules with no sources

```
myprog: file1.o file2.o file3.o
	gcc -g -Wall -o myprog file1.o file2.o file3.o

clean:
	rm file1.o file2.o file3.o myprog
```

- make assumes that a rule's command will build/create its target
  - but if your rule does not actually create its target, the target will still not exist the next time, so the rule will always execute (`clean` above)
  - `make clean` is a convention for removing all compiled files

Rules with no commands

```
all: myprog myprog2

myprog: file1.o file2.o file3.o
	gcc -g -Wall -o myprog file1.o file2.o file3.o

myprog2: file4.c
	gcc -g -Wall -o myprog2 file4.c
...
```

- `all` rule has no commands, but depends on `myprog` and `myprog2`
  - typing `make all` will ensure that `myprog`, `myprog2` are up to date
  - `all` rule often put first, so that typing `make` will build everything

Variables

```
NAME = value          (declare)
$(NAME)               (use)
```

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog

$(PROGRAM): $(OBJFILES)
        gcc -g -Wall -o $(PROGRAM) $(OBJFILES)

clean:
        rm $(OBJFILES) $(PROGRAM)
```

- variables make it easier to change one option throughout the file
  - also makes the makefile more reusable for another project

More variables

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
ifdef WINDIR          # assume it's a Windows box
        PROGRAM = myprog.exe
endif
CC = gcc
CCFLAGS = -g -Wall

$(PROGRAM): $(OBJFILES)
        $(CC) $(CCFLAGS) -o $(PROGRAM) $(OBJFILES)
```

- variables can be conditional (`ifdef` above)
- many makefiles create variables for the compiler, flags, etc.
  - this can be overkill, but you will see it "out there"

Special variables

$@	the current target file
$^	all sources listed for the current target
$<	the first (left-most) source for the current target
	(there are other special variables)

```
myprog: file1.o file2.o file3.o
        gcc $(CCFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
        gcc $(CCFLAGS) -c $<
```

Auto-conversions

- rather than specifying individually how to convert every `.c` file into its corresponding `.o` file, you can set up an *implicit* target:

```
# conversion from .c to .o
.c.o:
        gcc $(CCFLAGS) -c $<
```

  - "To create *filename*.o from *filename*.c, run gcc -g -Wall -c *filename*.c"

- for making an executable (no extension), simply write `.c` :

```
.c:
        gcc $(CCFLAGS) -o $@ $<
```

  - related rule: **`.SUFFIXES`** (what extensions can be used)

Dependency generation

- You can make `gcc` figure out dependencies for you:

  `$ gcc -M` ***`filename`***

  - instead of compiling, outputs a list of dependencies for the given file

  `$ gcc -MM` ***`filename`***

  - similar to -M, but omits any internal system libraries (preferred)

- Example:

```
$ gcc -MM linkedlist.c
linkedlist.o: linkedlist.c linkedlist.h util.h
```

  - related command: `makedepend`