introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

Introduction to make

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

introduction

make

- Automates certain tasks
  - Usually simple command-line stuff
  - Compiling multi-file programs
  - Archiving/extracting
  - Software installation
- Often used to manage builds
  - Compiles only as necessary
    - Uses file modification times to decide when it is necessary

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

basic make

Make rules

- A basic makefile consists of *rules*

```
target : dependencies
TAB[command1]
TAB[command2]
...
```

- The tab character precedes the rule
- The target is (usually) a file to be created

Make example

- E.g.,

```
program : main.c
   gcc main.c -o program
```

- `main.c` should already exist
  - Or, there's another target that creates it
- `main.c` will only be compiled if:
  1. `program` doesn't exist, or
  2. `main.c` is newer than `program`

Dependency recursion

- Dependencies are checked recursively down the tree:

```
program : main.o
   gcc main.o -o program

main.o : main.c
   gcc -c main.c
```

- Nothing happens if `program` is newer than `main.o`, and `main.o` is newer than `main.c`
- If `main.o` doesn't exist, or is older than `main.c`, it will be rebuilt, then `program` will be rebuilt
- If `program` doesn't exist, or is older than `main.o`, it will be rebuilt

Slightly more involved example


```
program : main.o service.o
   gcc main.o service.o -o program

service.o : service.c service.h
   gcc -c service.c

main.o : main.c service.h
   gcc -c main.c
```

- If `main.c` is updated, then `main.o` and `program` are rebuilt
- If `service.c` is updated, then `service.o` and `program` are rebuilt
- If `service.h` is updated, everybody is updated

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

continuing lines

Continuing lines

- Use \ to continue a dependency list or a command
  `program` are rebuilt

```
program : main.o curses.o utils.o keyboard.o \
          deck.o suits.o
  gcc -o program main.o curses.o utils.o keyboard.o \
       deck.o suits.o

...
```

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

command prefixes

Command prefixes

Introduction to make

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

- Turn off make echo by preceding line with a `@`

```
blah :
   @echo "Don't say this line twice"
```

- If any command returns an unsuccessful status, `make` reports the error and exits
- Precede a line with a `-` to have `make` ignore the status

```
clean :
   -rm program # fails if program doesn't exist
   -rm *.o     # We want this to happen, regardless
```

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

invocation

Specifying input file

- Specify a makefile using the option `-f` option to `make`:

```
$ make -f someMakeFile
```

- If not specified, `make` looks in the current directory for:
  1. `makefile`
  2. `Makefile`

Specifying a target

- Make allows you to specify target(s)

  `make [options] [target]`

- If no target is specified, `make` builds the first target it finds
- `-n` (dry run) is another handy option
  - Just print commands that would execute, w/out executing them

Phony targets (gnu only)

- Some targets exist for convenience
- We don't actually want to produce a file
- Commands won't run if a file of the same name exists
- We can declare targets as phony:

```
.PHONY : clean

clean :
   -rm program # fails if program doesn't exist
   -rm *.o     # We want this to happen, regardless
```

- No time stamps are compared, commands run every time

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

macros

Defining macros

Macros can be defined in a makefile:

```
OBJS = main.o curses.o utils.o keyboard.o \
           deck.o suits.o
cc = gcc
CFLAGS = -O

program : $(OBJS)
   $(cc) $(CFLAGS) $(OBJS) -o program

main.o : main.c
   $(cc) -c $(CFLAGS) main.c

$(OBJS) : sysdefs.h

...
```

Macro substitution

Evaluates the macro, after some substitutions.

```
SOURCE = main.c curses.c utils.c keyboard.c \
         deck.c suits.c

OBJS = ${SOURCE:.c=.o}

cc = gcc
CFLAGS =

...
```

Defined macros

$@ Name of current target

$< Name of first prerequisite

$^ All prerequisites

$? All prerequisites newer than target

```
program : main.c service.h
   $(cc) $(CFLAGS) $< -o $@
...
```

Choosing a different shell

- If you want to use a different shell, say, `bash`, to interpret the commands
- Set the `SHELL` variable at the top to modify all commands:

```
SHELL := /bin/bash

...
```

- You can do this for individual targets:

```
program : SHELL:=/bin/bash
program : main.c service.h
   $(cc) $(CFLAGS) $< -o $@
...
```

Suffix rules

- Some rules are easy to generalize
- If target has the same name as a dependency, but different suffix
- E.g., compile C files into object code

```
big.o : big.c this.h that.h other.h

%.o  :  %.c
   $(cc) -c $(CFLAGS) $<
```

- Other dependencies can be named
- Can also be specified this way:

```
.c.o :
   $(cc) -c $(CFLAGS) $<
```

Suffix rules for java: compile java source files into bytecode


```
JFLAGS = -g
JC = javac
RM = rm -f

# define new file suffixes
.SUFFIXES: .java .class

.java.class:
   $(JC) $(JFLAGS) $<

CLASSES = \
   Main.java \
   FancyClass.java \
   SpecialLibrary.java

all: classes

classes : ${CLASSES:.java=.class}

clean:
   $(RM) *.class
```

Suffix rules for LaTex: compile LaTex source files into pdf files


```
.SUFFIXES: .tex .pdf

.tex.pdf :
        pdflatex $<
        pdflatex $<

PDFS = unix.pdf

all: pdfs

pdfs : ${PDFS:.tex=.pdf}

.PHONY : clean

clean :
        -\rm *.aux
        -\rm *.log
        -\rm *.nav
        -\rm *.out
        -\rm *.snm
        -\rm *.toc
        -\rm *.vrb
```

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

recap

Recap

- Make files can do anything you do at the command line
- Care has to be taken to make them portable
- We've looked at fairly simply makefiles
  - Still useful
  - Makefile might call other makefiles
  - Macros can be defined in a separate file, used by several makefiles

introduction
basic make
dependencies
continuing lines
command prefixes
invocation
macros
recap
credits

credits

Thanks

The contents of these slides were created by Kurt Schmidt and modified by other faculty of the Drexel University CS Department including Geoffrey Mainland, Vera Zaychick, Jeremy Johnson, Spiros Mancoridis, and others.