The make Utility

Programming Tools and Environments
Winter 2006

make

- The make utility automates certain tasks (usually simple command-line stuff)
- Examples:
  - compiling multi-file programs
  - archiving/extracting
  - installation

make

- Typically used for program development:
  - runs the compiler only when necessary
  - uses file modify times to decide when it is necessary

Makefiles

- You have to give make a file containing targets.
- Targets are generally associated with:
  - a dependency list
  - command line(s) (that usually build the target)

Dependency List

- A list of files that the target depends upon
  - the target is only built if it doesn’t not exist, or if it exists, but is older than any of the dependencies.
  - dependency may also have targets in the same makefile; these are then intermediate targets
  - This relationship is checked recursively, down the dependency tree

Input Files

- Specify a makefile using the -f option to make
  - make -f myMakefile
- If not specified, make looks in the current directory for (in order):
  - makefile
  - Makefile
**Rules**

- `target : prerequisites`
- `command1`
- `command2`

These MUST be abs!  

- Target depends on the files listed after the colon.  
- Commands are Unix commands that (may) build a new target file.

**Simple Example**

- **This rule would tell make that the file linecount depends on the file foo.c.**
- **To build linecount the command “wc -l foo.c > linecount” should be run.**
- **If linecount doesn’t exist, or foo.c is newer (modified more recently than linecount), the wc command will be executed.**

**Simple Makefile for building a program.**

```
main: main.o subs.o
    gcc -o main main.o subs.o

subs.o: subs.c subs.h
    gcc -c subs.c

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

**Make command line (simplified)**

```
make [options] [targets]
```

- Options include:  
  - “-f filename” use filename as the Makefile
  - “-n” don’t do anything – just print what needs to be done.
  - targets are what should be built (if needed).

**Specifying a target**

- **Just specify the target after the make command:**
  
  `make subs.o`

- **If no target is specified, the first target listed in the file is created.**
Revisit Example

- A target doesn't need dependencies, nor does its command-line need to make create the target:
  - clean:
    - \rm *.o
    - \rm main
  - $ make clean will cause those files to be removed (assuming there is no file called clean in the directory. See next slide.)

Phony targets (GNU only)

- A target can be specified as phony, so that it is always executed:
  - .PHONY: clean
    - clean:
      - \rm *.o
      - \rm main
  - The dash in front of the commands tells make to ignore the return values; to continue executing, even if an error is returned.

Macros

- You can create macros (fancy variables)

  OBJJS = subs.o main.o
  main: $ (OBJJS)
  gcc -o main $(OBJJS)

Fancy Stuff

- Macro substitution – evaluates to the value of a macro after some substitutions:
  - SOURCE = main.c subs.c
  - OBJJS = $(SOURCE:.c=.o)
  - now OBJJS is main.o subs.o

Suffix Rules

- General rules for building files that end with some suffix from files with the same name but a different suffix.
  - For example, how to build a .o file from a .c file.
  - Special predefined macros help with suffix rules:
    - $< means current prereq.
    - $@ is the current target.

Example Suffix Rule

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

  This rule tells Make how to create a .o file any time is has a .c file (and needs the .o file to build a target).
A Complete Example

```bash
.c.o:
gcc -c <$

SOURCE = main.c subs.c
OBJJS = $(SOURCE:.c=.o)

main: $(OBJJS)
gcc -o main $(OBJJS)
```

Comments and other Makefile notes

- Comments begin with a `#`
- Can be placed at the beginning of a line or after a non-comment line
- Lines that are too long can be continued on the next line by placing a `\` at the end of the first line

Basic Makefile example

```bash
program : main.o iodat.o dorun.o
gcc -o program main.o iodat.o dorun.o
main.o : main.c
gcc -c main.c
iodat.o : iodat.c
gcc -c iodat.c
dorun.o : dorun.c
gcc -c dorun.c
```

Simplifying the example Makefile with macros

```bash
OBJJS = main.o iodat.o dorun.o
CC = /usr/bin/gcc
program : ${OBJJS}
${CC} -o $@ ${OBJJS}
main.o : main.c
${CC} -c $@
iodat.o : iodat.c
${CC} -c $@
dorun.o : dorun.c
${CC} -c $@
```

Simplifying the example Makefile again

```bash
OBJJS = main.o iodat.o dorun.o
CC = /usr/bin/gcc

program : ${OBJJS}
${CC} -o $@ ${OBJJS}
```

Other useful Makefile tips

- Include a way to clean up your mess
  ```bash
  clean:
  /bin/rm -f *.o core
  ```
- Include a target to build multiple programs
  ```bash
  all:
  program1 program2 program3
  ```