The *make* utility

(original presentation courtesy of Alark Joshi)

- *Make* is a utility that is included with Linux/Unix operating systems
- It is a command generator
- It is designed to help you compile large projects

What is *make*?

- *Make* is non-procedural
  - You tell make what you want (e.g. a particular class file or executable)
  - You provide a set of rules showing dependencies between files
  - *Make* uses the rules to get the job done

Why use *make*?

- For large programs, recompiling all the pieces of the program can be very resource intensive
- But if the program is complex, determining exactly what needs to be recompiled can be a time-consuming (and error-prone) task
- The *make* utility was written to assist with this process

What does *make* do?

- *Make* uses a file called ***Makefile*** (or makefile) to determine what needs to be recompiled
- Makefile contains a set of rules
- When you run *make*, it uses the rules in the Makefile to determine what needs to be done
- *Make* does the minimum amount of work needed to get the job done

Rules in a Makefile

- A typical rule has the form

```
target: dependency list
        command list
```

- target can be the name of a file that needs to be created (or a "phony" name that can be used to specify which command to execute)
- The dependency list is a space separated list of files that the target depends on in some way
- The command list is one or more linux commands needed to accomplish the task of creating the target

Rules in a Makefile

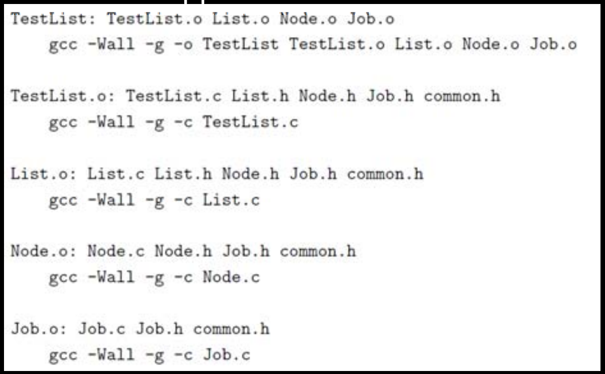
- Each command must be indented with a **tab**
- Both dependency lists and commands can be continued onto another line by putting a \ at the end of the first line
- A # is used to start a comment in a makefile
  - The comment consists of the remainder of the line

Example

- Dependencies for a "list" program:
- Main.c and TestList.c include List.h, Node.h, Job.h, and common.h
- List.c includes List.h, Node.h, Job.h, and common.h
- Node.c includes Node.h, Job.h, and common.h
- Job.c includes common.h

Rules for the "list" program

- Brute-force approach



```
TestList: TestList.o List.o Node.o Job.o
    gcc -Wall -g -o TestList TestList.o List.o Node.o Job.o

TestList.o: TestList.c List.h Node.h Job.h common.h
    gcc -Wall -g -c TestList.c

List.o: List.c List.h Node.h Job.h common.h
    gcc -Wall -g -c List.c

Node.o: Node.c Node.h Job.h common.h
    gcc -Wall -g -c Node.c

Job.o: Job.c Job.h common.h
    gcc -Wall -g -c Job.c
```

How *make* works

- When you type *make* without a target name will assume you mean to make the first real target in the makefile
- When you type *make target*, the make utility will look at the rule for target
- *make* will recursively search through the rules for all the dependencies to determine what has been modified

How *make* works

- If the current version of *target* is newer than all the files it depends on, make will do nothing.
- If a target file is *older* than any of the files that it depends on, the command following the rule will be executed

Macros

- Sometimes, you find yourself using the same stuff in lots of command actions---macros simplify things
- Define macro

```
CC = gcc
CFLAGS = -O –Wall –g
PROGS = list TestList
```

- Then use the macro by typing $(macroname)

```
$(CC) $(CFLAGS) –c List.c
```

```
# File "makefile" used to build "ola207" executable

CXX = g++ -std=c++11 –pedantic

ola207:  bakery.o bakeryPrint.o
	$(CXX) -g bakery.o bakeryPrint.o -o ola207

bakery.o:  bakery.cpp bakeryPrint.h bakeryConstants.h
	$(CXX) -g -c bakery.cpp

bakeryPrint.o:  bakeryPrint.cpp bakeryPrint.h bakeryConstants.h
	$(CXX) -g -c bakeryPrint.cpp
```

Phony Targets

- Phony targets are targets that do not correspond to a file

```
clean:
    rm –f *.o
```

Example

```
all: subdirs

subdirs:
	cd bad; make
	cd almost-generic; make
	cd generic-with-library; make
	cd generic; make

clean:
	cd bad; make clean
	cd almost-generic; make clean
	cd generic-with-library/; make clean
	cd generic; make clean
```

(Optional) Advanced Stuff follows:

Substitution Rules

- Often, you will find that your Makefile has many similar commands
- You can use patterns to define rules and commands for such cases
- For example, we could use the rule

```
%.o : %.c
   $(CC) $(CFLAGS) –c $<
```

- Which says that every .o file depends on the corresponding .c file and can be created from it with the command below the rule

Substitution Rules - Internal macros

- % - any name (the same in all occurrences)
- $@ - The name of the current target
- $< - The first dependency for the current target
- $^ - All the dependencies for the current target

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

hello: hello.o
   $(CC) $(CFLAGS) $< -o $@
```

Suffix Rules

- A suffix rule identifies suffixes that make should recognizes

```
            .SUFFIXES: .o .c
```

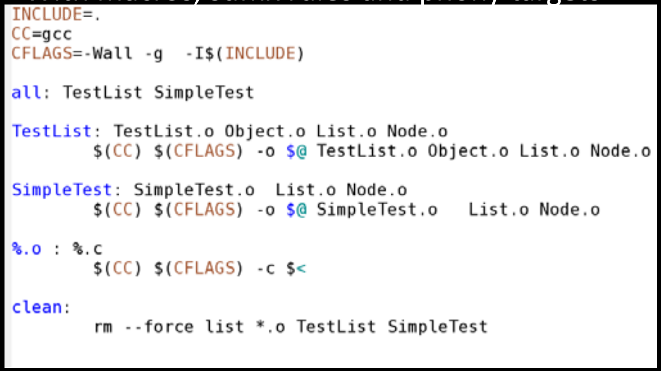
- Another rule shows how files with suffixes are related

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

- Think of this as saying the .o file is created from the corresponding .c file using the given command

A More Advanced Example

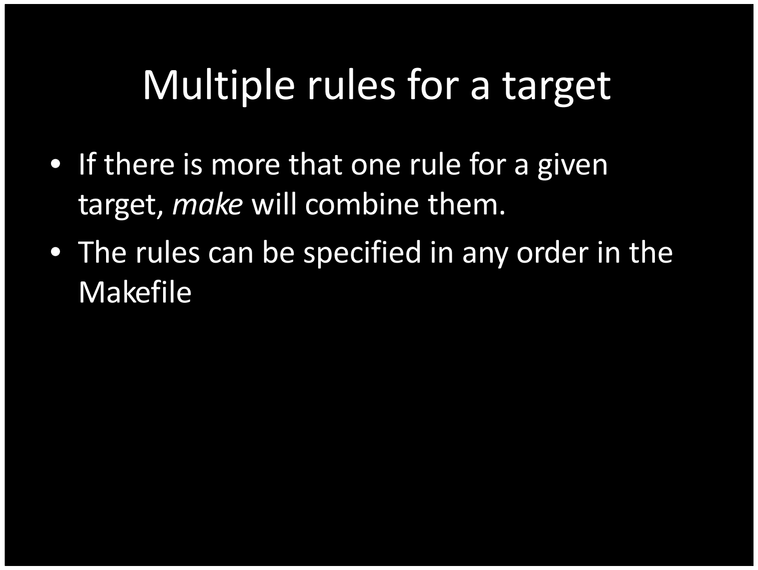
- With macros, suffix rules and phony targets



```
INCLUDE=.
CC=gcc
CFLAGS=-Wall -g  -I$(INCLUDE)

all: TestList SimpleTest

TestList: TestList.o Object.o List.o Node.o
	$(CC) $(CFLAGS) -o $@ TestList.o Object.o List.o Node.o

SimpleTest: SimpleTest.o  List.o Node.o
	$(CC) $(CFLAGS) -o $@ SimpleTest.o   List.o Node.o

%.o : %.c
	$(CC) $(CFLAGS) -c $<

clean:
	rm --force list *.o TestList SimpleTest
```

Taking the drudgery out of dependencies

- Dependencies for a .o file should include all the user written header files that it includes
- For a big project, getting all of these right can take some time
- The *gcc* command has an option -MMD which tells it to compute the dependencies.
- These are stored in a file with the suffix .d
- Include the .d file into the Makefile using `-include *.d`



Multiple rules for a target

- If there is more that one rule for a given target, *make* will combine them.
- The rules can be specified in any order in the Makefile