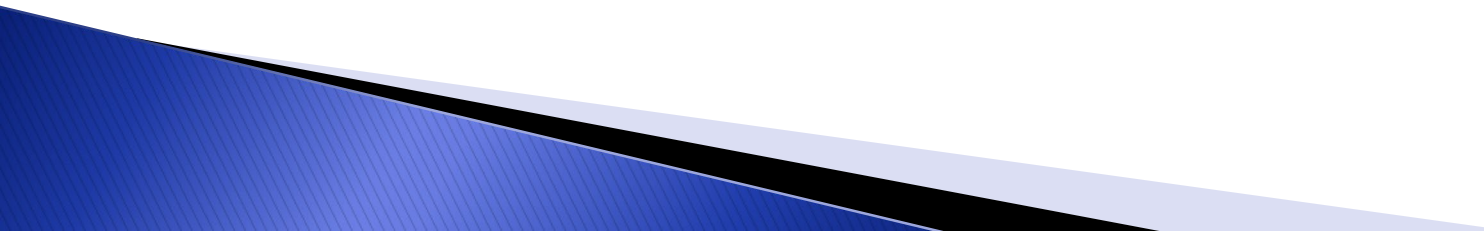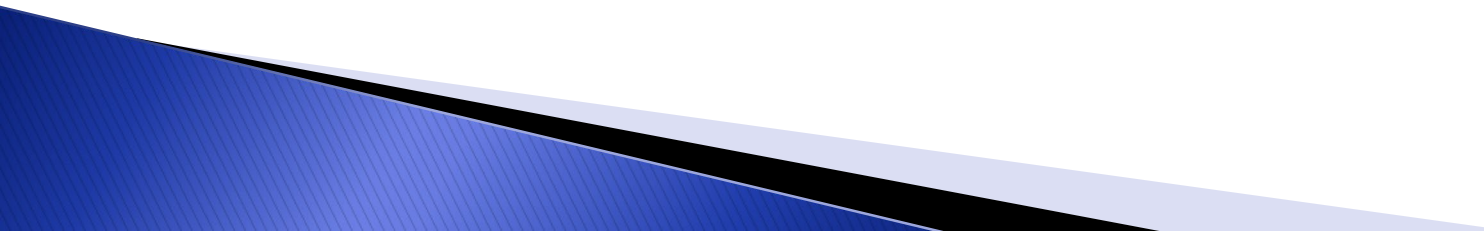# Makefiles

CSE 2421

Linux/Unix Make Command

- make is a utility for building and maintaining groups of programs (and other types of files) from source code
- The purpose of the make utility is to determine automatically which pieces of a large program need to be re-compiled and issue the commands necessary to recompile them.
- To use make, you must create a file call Makefile (or makefile) that describes the relationships among files in your program
- Make decides whether a target needs to be regenerated by comparing file modification times

Makefile Overview

▸ make is a Unix/Linux utility, which is used to manage building (compilation) of programs.

▸ It is very commonly used in industry in Unix/Linux environments.

▸ For a large program, it is more efficient for the developer and the machine if the program is written as a number of smaller files; recompiling each of these various files that make up the program is only done when something changes which requires recompilation.

▸ This also promotes reusability of code, because source code for functions can be kept in separate files, and the file can be called and compiled in any program for which it is useful.

▸ Makefiles contain file dependency lists and UNIX commands and will run them in a specified sequence.

▸ Anything that can be entered at the UNIX/LINUX command prompt can be in the makefile.

Writing and executing a makefile

▸ The name of your makefile *has to be*: makefile or Makefile

▸ The directory you put the makefile in matters!

▸ You can only have one makefile per directory.

▸ Each command must be **preceded by a TAB** and is **immediately followed by hitting the enter key.**

▸ MAKEFILES ARE UNFORGIVING WHEN IT COMES TO WHITESPACE!

▸ To run make… you must be in the directory where the makefile is.

▸ make is a Unix/Linux utility program.

▸ Command to run make on the makefile in the directory:

    $make tag-name

Note: tag-name is also called section name

▸ Let's look at a program with multiple source files that could be built with make. Let's suppose we want the executable to be called *mkprog*

Example makefile: Suppose program with the following files

/* mkfunc.h */

/* mkfact.c */

```
/*  The header file contains function
    prototypes for all functions in the
    program, except main() and library
    functions
*/
void print_hello();
int fact(int n);
```

```
#include "mkfunc.h"
/* note " ", not < > */

int fact(int n) {
    if (n != 1) {
        return (n * fact(n – 1));
    }
    else return 1;
}
```

.... and the following files

/* mkhello.c */

```
#include "mkfunc.h"
#include <stdio.h>

void print_hello() {
    printf("Hello World!\n");
}
```

/* mkmain.c */

```
#include "mkfunc.h"
#include <stdio.h>

int main() {
    print_hello();
    printf("5 factorial is %d", fact(5));
    return 0;
}
```

Makefile details

- Compiling our example would look like:

    gcc –std=c99 –pedantic –o mkprog mkmain.c mkhello.c mkfact.c

- The basic makefile is composed of lines:

    target: dependencies

    [tab] system command   [These are known as *rules*]

- *"all"* is the typical default target for makefiles

    $make all

    This executes:

    gcc –std=c99 –pedantic –o mkprog mkmain.c mkhello.c mkfact.c

- The *make* utility will execute the first target in the makefile by default, if no other one is specified, so the following command has the same effect with the makefile given in the next slide:

    $make   /* target "all" is implied if "all" is first */

Makefile for the above program

```
all: mkprog

mkprog: mkmain.o mkfact.o mkhello.o
        gcc mkmain.o mkfact.o mkhello.o –o mkprog

mkmain.o: mkmain.c  mkfunc.h
        gcc –std=c99 –pedantic –c mkmain.c

mkfact.o: mkfact.c mkfunc.h
        gcc –std=c99 –pedantic –c mkfact.c

mkhello.o: mkhello.c mkfunc.h
        gcc –std=c99 –pedantic –c mkhello.c

clean:
        rm –rf  *.o  mkprog
```

What if we added a new file?

```
all: mkprog

mkprog: mkmain.o mkfact.o mkhello.o mk_newfunc.o
        gcc mkmain.o mkfact.o mkhello.o mk_newfunc.o –o mkprog

mkmain.o: mkmain.c  mkfunc.h
        gcc –std=c99 –pedantic –c mkmain.c

mkfact.o: mkfact.c mkfunc.h
        gcc –std=c99 –pedantic –c mkfact.c

mkhello.o: mkhello.c mkfunc.h
        gcc –std=c99 –pedantic –c mkhello.c

mk_newfunc.o: mk_newfunc.c mkfunc.h
        gcc –std=c99 –pedantic –c mk_newfunc.c

clean:
        rm –rf  *.o  mkprog
```
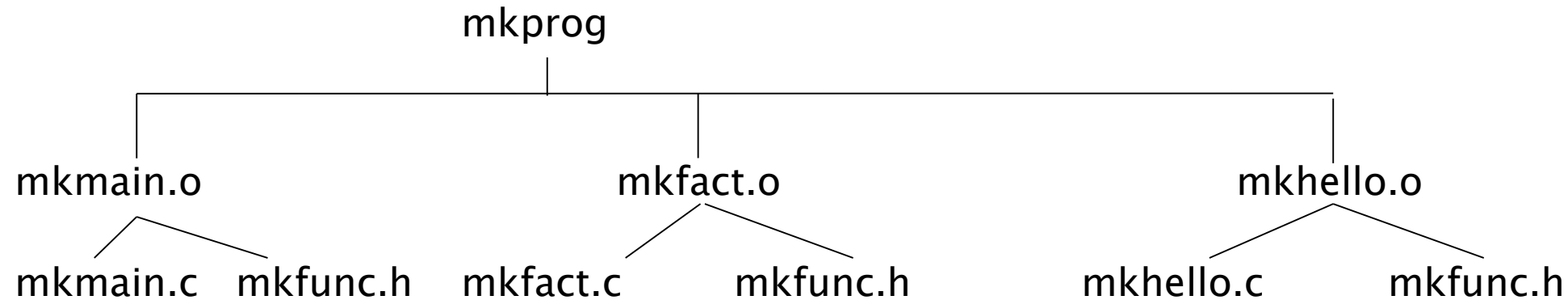
Dependency Table

▸ Based on the makefile above, make will create *a dependency table*:

```
                           mkprog
        ┌────────────────────┼────────────────────────┐
     mkmain.o             mkfact.o                  mkhello.o
      ┌───┴────┐          ┌───┴────┐                ┌───┴────┐
  mkmain.c  mkfunc.h   mkfact.c  mkfunc.h        mkhello.c  mkfunc.h
```

*Using this table, make can determine what to recompile based on changes in any of the files.

Examples from the command line

/home/4/jones.5684/cse2421
$make clean
rm –rf *.o mkprog
/home/4/jones.5684/cse2421
$make
gcc –std=c99 –pedantic –c mkmain.c
gcc –std=c99 –pedantic –c mkfact.c
gcc –std=c99 –pedantic –c mkhello.c
gcc mkmain.o mkfact.o mkhello.o –o mkprog
/home/4/jones.5684/cse2421
$mkprog
Hello World!
5 factorial is 120
/home/4/jones.5684/cse2421
$make
make: Nothing to be done for 'all'.

Use ls -lt

▸ To see what files have been modified recently, use ls -lt
  ◦ -l means long form (give details like times)
  ◦ -t means time sorted order (give the newest first)

▸ This will help you predict what will be compiled by make
▸ This will show you when make has nothing to do

Make can do more than compile C code

- Make can be used to issue *any* command in linux
- Make follows the rules to build the target specified on the command line
- It will run those commands if the target is older than any of the things it depends on

MANDATORY WARNINGS

▸ In this class, any compilation that turns a .c file into a .o file or into an executable file must have ALL of the following 4 warnings:
  ◦ –std=c99
  ◦ –pedantic
  ◦ –Wreturn–type
  ◦ –Wformat

▸ Forgetting any of these can cause a zero on your lab

I'm tired of writing –std=c99 –pedantic –Wformat –Wreturn-type

- With automatic variables…
  - $@ the target of the current rule
  - $^ the dependencies of the current rule
  - $< the first dependency of the current rule
  - $* the stem with which the pattern of the current rule matched
- And patterns…
  - % the Make wildcard % specifies a pattern

- We can create the following rule :

```
%.o : %.c
    gcc –std=c99 –pedantic –Wreturn-type –Wformat –g –c $< –o $@
```

Why bother?

%.o : %.c

   gcc –std=c99 –pedantic –Wreturn–type –Wformat –g –c $< –o $@

▸ Make will default to using **cc** (and not gcc) to build .o files and cc won't give warnings the way gcc with the flags we use gives warnings
▸ This doesn't deal with header file dependencies

But I have header files

%.o : %.c *.h

  gcc –std=c99 –pedantic –Wreturn–type –Wformat -g –c $< -o $@

- This rebuilds **all** .o when you touch **any** .h file
  ◦ This is almost perfect behavior and it let's us use a single, **very** useful rule
  ◦ There has to be at least one .h file in the directory for this to work right or it defaults to cc again.
- If you change one .c file it only rebuilds the one matching .o file
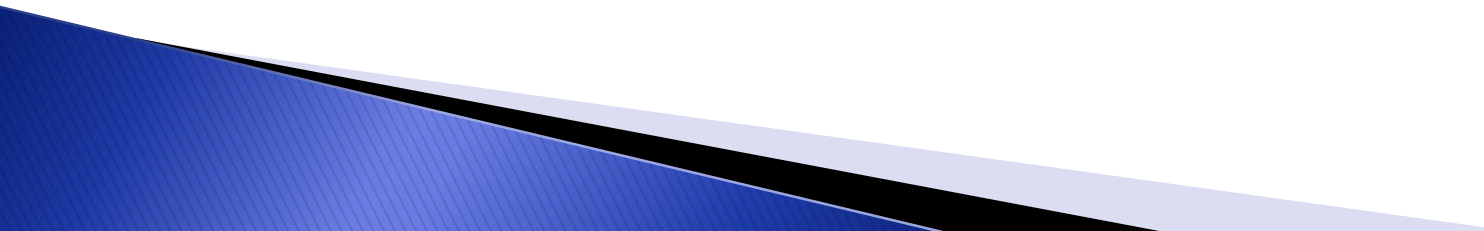
Simplify with variables

```
#put this at the top of the file
CFLAGS= –std=c99 –pedantic –Wreturn–type –Wformat

#it shortens the compile line
%.o : %.c *.h
    gcc $(CFLAGS) –g –c $< –o $@
```

A Rule for Executables

```
# all you have to do is change the target name and the dependency list
# (assuming that it only depends on .o files)
# the previous slides handle turning .c into .o
lab3: lab3.o table.o memory.o
    gcc –g –o $@ $^

# if you have libraries, put them at the end of the rule
lab3: lab3.o table.o memory.o
    gcc –g –o $@ $^ –lm –lncurses –L. –lcustom
```

Simplify with variables

```
#put this at the top of the file
LIB_FLAGS= -lm -lncurses -L. -lcustom

# if you have libraries, put them at the end of the rule
lab3: lab3.o table.o memory.o
     gcc -g -o $@ $^ $(LIB_FLAGS)
```

MANDATORY FLAG: make –r

- You can suppress the implicit rules that make has by using the –r flag:

make –r lab3

- Always use –r when you use make in order to force it to only build things exactly as you direct
- The graders will always build with make –r
- If your code fails to build with make –r you get a zero on the lab