5_UNIX

The *make* Tool/Command

The reasons for the ***make*** *utility*

In a large project, one doesn’t want to re-compile *everything*, every time a change is made.

make keeps track of dependencies,
of what needs re-compiling,
of what needs re-linking.

Syntax: make [-f makefile]

Most programmers use a standard output name.

Examples:

```
> make
> make –f Project23make
```

Often Used Options:

-f Tells **make** which file to use as its makefile
Without –f, it looks first for *makefile*
and then for *Makefile* by default.

-n Tells **make** to print out what it would have done without actually doing it.

-k Tells **make** to keep going when an error is found, rather than stopping as soon as the first problem is detected.

A simple code file & its make file

```
/*  power.c            */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void)
{
   float x, y;

   printf("\nThe program takes x and y from stdin and displays x^y.\n");

   printf("Enter integer x:  ");
   scanf ("%f", &x);
   printf("Enter integer y:  ");
   scanf ("%f", &y);

   printf("x^y is: %6.3f\n", pow((double)x,(double)y));
   return EXIT_SUCCESS;
}
```

/* The RUN is on the next slide */

```
[bielr@athena ~/csc60]30> gcc power.c -o power
/tmp/ccEzY1AX.o(.text+0x83): In function `main':
: undefined reference to `pow'
collect2: ld returned 1 exit status
[bielr@athena ~/csc60]31>
```

Forgot to add… –lm …to link to the math library.

```
[bielr@athena ~/csc60]38> gcc power.c -lm -o power
[bielr@athena ~/csc60]39> power

The program takes x and y from stdin and displays x^y.
Enter integer x:  9.82
Enter integer y:  2.3

x^y is: 191.362

[bielr@athena ~/csc60]40>
```

A makefile for the power program:

```
>vim makefile
```

Contents of the file named *makefile*

```
# Sample makefile for the power program
# Remember:  each command line starts with a TAB

power: power.c
	gcc power.c –o power –lm
```

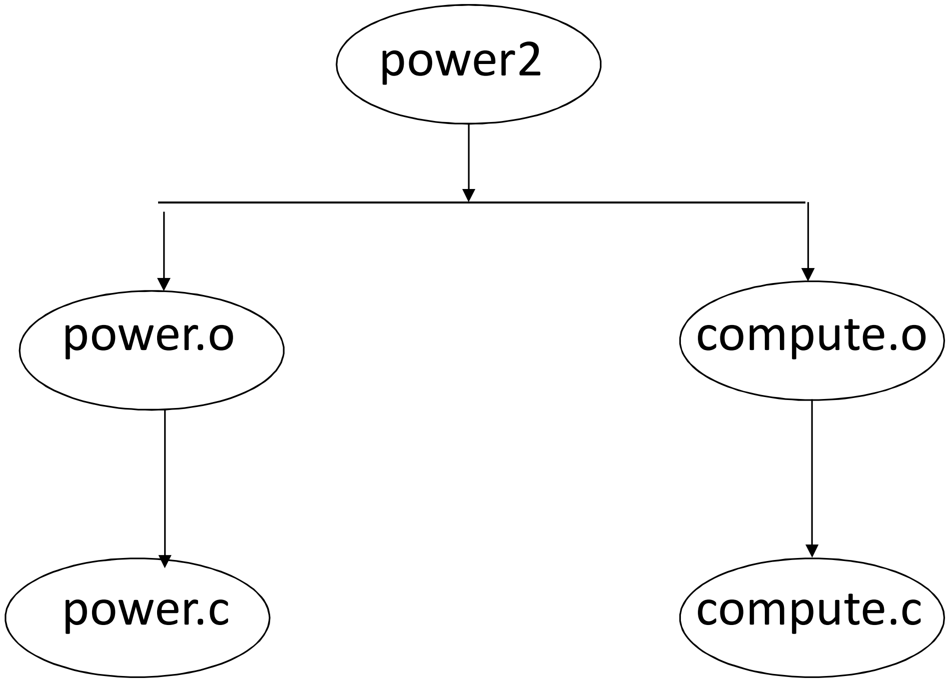
```
>
```

```
[bielr@athena ~/csc60]40> make
make: `power' is up to date.
[bielr@athena ~/csc60]41>
```

Use touch to alter the dates to force recompilation

```
[bielr@athena ~/csc60]44> touch power.c
[bielr@athena ~/csc60]45> make
gcc power.c -o power -lm
[bielr@athena ~/csc60]46>
```

Dependency Chart: *Alter **power** so it is in <u>two</u> functions, two files.*



/* **power2.c**   *Alter **power** so it is in <u>two</u> functions. */*
                                                  / page 1 of 2 */*

```
#include <stdio.h>
#include <stdlib.h>
double compute(double x, double y);
int main(void)
{
   float x, y;

   printf("\nThe program takes x and y from stdin and displays x^y.\n");

   printf("Enter integer x:  ");
   scanf ("%f", &x);

   printf("Enter integer y:  ");
   scanf ("%f", &y);

   printf("\nx^y is: %6.3f\n\n", compute(x,y));
   return EXIT_SUCCESS;
}
```

```
/* The new function    page 2 of 2 */
/* compute.c                        */

#include <math.h>
double compute(double x, double y)
{
    return (pow(x, y));
}
```

First pass at a makefile:

```
>cat makefile
power2: power.o compute.o
	gcc power.o compute.o  -o power2 -lm
>
```

```
[bielr@athena ~/csc60]60> make
make: *** No rule to make target `power.o', needed by `power'.  Stop.
[bielr@athena ~/csc60]61>
```

/   Second pass at a makefile: */*
/ Look at its contents. We have no* ***p2.h*** *but it is included in light italics to show where it would be placed.   */*

```
>cat makefile
power2: power2.o compute.o  p2.h
	gcc power2.o compute.o  -o power2 -lm
power2.o: power2.c  p2.h
	gcc -c power2.c
compute.o: compute.c  p2.h
	gcc -c compute.c
```

/* Run make using our new makefile */

```
[bielr@athena ~/csc60]68> make
gcc -c power2.c
gcc -c compute.c
gcc power2.o compute.o  -o power2 -lm
[bielr@athena ~/csc60]69>
```

```
#   Third and last pass at a makefile:
#      power2.h is not needed but included in light italics to
#     show where it would be located if it was needed
```

```
#     >cat makefile
power2: power2.o compute.o  power2.h
       gcc power2.o compute.o  -o power2 –lm

power2.o:  power2.h

compute.o:  power2.h
>
```

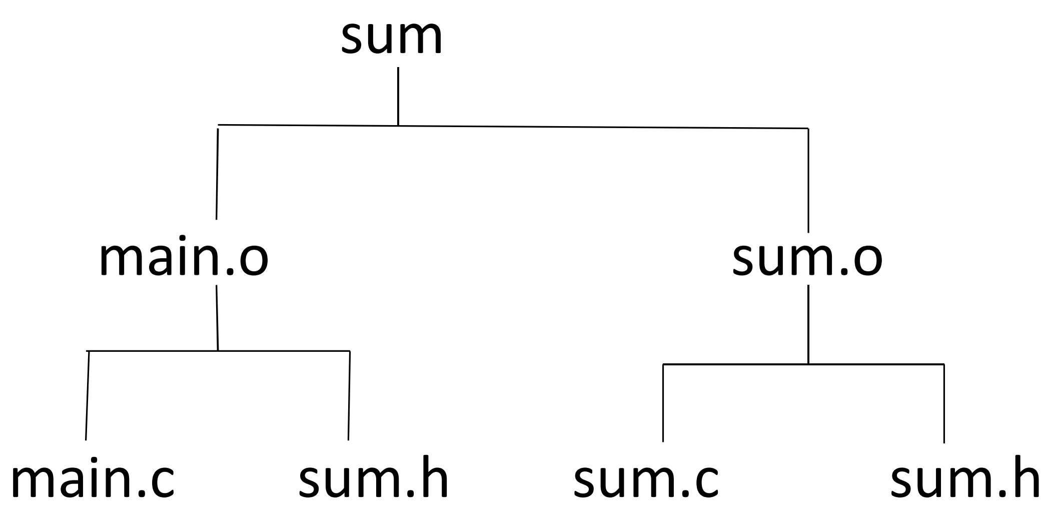
/ Helpful Comments */***

- Start by opening *vim*, and typing in the commands to a file named *makefile*.
  Close *vim* and then at the prompt, type: *make*
- When you enter **vim**, type: **:set list**

  This will show the non-printable characters:
  ^I = tab
  $ = end of line
- To reverse the setting, type: **:set list!**
- To create a tab on *athena*, you may have to hit the tab key **twice** in a row.

Another Example

*Example with two functions and a *.h file:*



Makefile Contents:

```
sum:  main.o  sum.o                  dependency
        gcc –o sum main.o sum.o      action

main.o: main.c sum.h                 dependency
        gcc –c main.c                action

sum.o: sum.c sum.h                   dependency
        gcc –c sum.c                 action
```

Dependency lines start in column 1.
Action lines <u>must</u> begin with a **tab**.

If anything on a dependency line has changed, then the associated action(s) take place.

A dependency and its actions together are called a **rule.**

An alternate example of this makefile:

```
sum:  main.o  sum.o                     dependency
         gcc –o sum main.o sum.o        action

main.o: sum.h                           dependency
         gcc –c main.c                  action

sum.o: sum.h                            dependency
         gcc –c sum.c                   action
```

A makefile with a macro:

```
sum:  main.o  sum.o                    dependency
        gcc –o sum main.o sum.o        action

main.o sum.o: sum.h                    dependency
        gcc –c $*.c                    action
```

The second rule states that the two .o files depend on sum.h.
If we edit sum.h, both main.o and sum.o must be remade.

The macro **$*.c** *expands first to main.c and then to sum.c*

Macros in a Makefile

The *make* utility supports simple macros that allow simple text substitution.

The macro must be defined before use, and is usually placed at start of the file.

Syntax: Macro_name = text

These are also called Macro Variables.

A more complicated example

The Compare_Sorts makefile in its entirety:
(Dissection of each line follows on the next slides.)

```
# Makefile for compare_sorts
#After excution, use prof to get a profile.

BASE     = /c/c/blufox
CC       = gcc
CFLAGS = -p
EFILE    = $(BASE)/bin/compare_sorts
INCLS    = -I$(BASE)/include
LIBS     = $(BASE)/lib/g_lib.a

OBJS     =  main.o chk_arrays.o compare.o \
            prn_arrays.o slow_sort.o

$(EFILE):  $(OBJS)
           @echo “linking. . . . .”
           @$(CC) $(CFLAGS) $(INCLS) –c $*.c

$(OBJS): compare_sorts.h
           $(CC)  $(CFLAGS)  $(INCLS)  -c $*.c
```

```
# Makefile for compare_sorts
#After execution, use prof to get a profile.
```

Comments start with # and go till end of line.

```
BASE  = /c/c/blufox
```

---

a macro definition.
Syntax: macro_name = replacement_string

BASE represents our base of operations on the local computer.
Doesn't have to be home directory.

```
CC        = gcc
```

The CC macro specifies the C compiler we are using.

CFLAGS = -p

The CFLAGS macro specifies the options, if any,
that will be used with the ***gcc*** *command.*

-p = Generate extra code to write profile information.
You must use this option when compiling the source files
you want data about, and you must also use it when linking.

What is profiling?

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

In very large projects, profiling can save your day by not only determining the parts in your program which are slower in execution than expected but also can help you find many other statistics through which many potential bugs can be spotted and sorted out.

```
EFILE     = $(BASE)/bin/compare_sorts
```

Specifies the executable file.

```
INCLS    = -I$(BASE)/include
```

---

*Specifies a directory for include files proceeded by the –**I** option.*

-I dir, --include-dir=dir

Specifies a directory dir to search for included makefiles. If several -I options are used to specify several directories, the directories are searched in the order specified. Unlike the arguments to other flags of make, directories given with -I flags may come directly after the flag: -Idir is allowed, as well as -I dir. This syntax is allowed for compatibility with the C preprocessor's -I flag.

```
LIBS        = $(BASE)/lib/g_lib.a
```

Tells the compiler where to find our programmer-constructed header file.

```
OBJS     =  main.o chk_arrays.o compare.o \
             prn_arrays.o slow_sort.o
```

In this macro definition the replacement string is the list of object files that occurs on the right side of the equal sign.

(Used backslash \ to continue the line.)

Order is unimportant.

```
$(EFILE):  $(OBJS)
        @echo "linking. . . . ."
        @$(CC) $(CFLAGS) $(INCLS) –c $*.c
```

The first line is a dependency line, and the other two specify the actions to be taken.

The @ symbol means that the action line itself is not to be echoed on the screen.

Macro invocation has form:

$(macro_name)

So $(EFILE) is replace by $(BASE)/bin/compare_sorts

which then becomes /c/c/blufox/bin/compare_sorts

```
$(OBJS): compare_sorts.h
        $(CC)  $(CFLAGS)  $(INCLS)  -c $*.c
```

$(OBJS) will be replace by the list of object files.

The second action line is expanded to:

```
@gcc –p –o /c/c/blufox/bin/compare_sorts main.o    \
    chk_arrays.o compare.o prn_array.o slow_sort.o \
    /c/c/blufox/lib/g_lib.a
```

Because of the backslash \, the line acts as one line.

-p causes the compiler to generate extra code suitable for the profiler.

More details about ***make*** (these expand just before use):

$?	List of prerequisites changed more recently than the current target
$@	Name of the current target
$<	Name of the current prerequisite
$*	Name of the current prerequisite, without any suffix
-	Tells **make** to ignore any errors.
@	Tells **make** not to print the command to standard output before executing it.

The ***touch*** command:

Syntax: touch [options] *files*

Changes two timestamps associated with a file:

- its *modification time* (when the file's data was last changed)
- its *access time* (when the file was last read)

If a given file doesn't exist, *touch* creates it as an empty file.

If we had done a *touch* on *compare_sorts.h,*

we would have forced all the other files to be recompiled.

Other Useful Tools:

diff	Prints the lines that differ in two files.
wc	Word Count. Counts lines, words, and characters in one or more files.
awk	A pattern scanning and processing language. On Linux, it is ***gawk,*** a superset of *awk.*
lex	Generates C code for lexical analysis. On Linux, it is ***flex ,*** a superset of *lex.*
sed	A stream editor that takes its commands from a file.
yacc	A parser generator.
bison	GNU Project parser generator (yacc replacement)

5_UNIX

The *make* Tool/Command

The End