make and makefile

CS 211

Steps of a C Program Compilation

- Step 1 – Run the Preprocessor
  - Deals with #include, #define, #ifndef, …
- Step 2 – Compile C code into Machine Code
  - Step 2a – Lexical Analysis
  - Step 2b – Parsing and Semanitic analysis
  - Step 2c – Creation of Machine Code in Object File
- Step 3 – Create the Executable Files
  - Linking the Program Object Files with Library Object Files

Steps of a C Program Compilation

- Step 1 & 2 – Create the Object Files
  - .o files
  - created using the –c flag in gcc (“compile only”)

- Step 3 – Create the Executable Files
  - By default, named: a.out
  - use the –o flag in gcc to call it something other than a.out

Multiple Source Code Programs

- Program is too large for a single .c file
- Used in team development for parallel development
- Used to create personal library code/files
- In the past was used to decrease compilation time

Multiple Source Code Programs

- Each .c file is first compiled into its own .o file
  - gcc -c myfile1.c
  - gcc -c myfile2.c
  - gcc -c myfile3.c
- All the .o files are linked together into one executable file
  - gcc myfile1.o myfile2.o myfile3.o -o myfile
- So 4 commands are needed to be typed in

Multiple Source Code Programs

- Makefiles allow us to store multiple commands so we don’t need to type them in even time
- Makefiles can check which source code files need to be recompiled since the last time the executable file was created.
- Helps keep track of the various files in use
- File is named: makefile

makefile Format

```
targetFile1:   dependencyFile1a   dependecyFile1b
        command1a
        command1b

targetFile2:   dependencyFile2a   dependecyFile2b
        command2a
        command2b
```

makefile Format: 2 Rules

```
targetFile1:   dependencyFile1a   dependecyFile1b
        command1a
        command1b

targetFile2:   dependencyFile2a   dependecyFile2b
        command2a
        command2b
```

makefile Format: separated by blank line

```
targetFile1:   dependencyFile1a   dependecyFile1b
        command1a
        command1b
---- this must be a blank line ----
targetFile2:   dependencyFile2a   dependecyFile2b
        command2a
        command2b
```

makefile Format: First line Targets before colon

```
targetFile1:   dependencyFile1a   dependecyFile1b
        command1a
        command1b

targetFile2:   dependencyFile2a   dependecyFile2b
        command2a
        command2b
```

makefile Format: Dependencies after colon

```
targetFile1:   dependencyFile1a   dependecyFile1b
        command1a
        command1b

targetFile2:   dependencyFile2a   dependecyFile2b
        command2a
        command2b
```

makefile Format: Commands after first line

```
targetFile1:   dependencyFile1a   dependecyFile1b
        command1a
        command1b

targetFile2:   dependencyFile2a   dependecyFile2b
        command2a
        command2b
```

makefile Format: Indented with a TAB

targetFile1: dependencyFile1a dependecyFile1b

<tab> command1a

<tab> command1b

targetFile2: dependencyFile2a dependecyFile2b

<tab> command2a

<tab> command2b

makefile Example for 2 Source File Program

```
max: max1.o max2.o
	gcc -o max max1.o max2.o

max1.o: max1.c max.h
	gcc -c max1.c

max2.o: max2.c max.h
	gcc -c max2.c
```

Source File 1 – max1.c

```
#include "max.h"

int main( )
{
        int x = 37;     int y = 52;

         int result = calcMax (x, y);

        printf ("Max is %d\n",  result );
}
```

Source File 2 – max2.c

```
#include "max.h"

int calcMax( int a, int b )
{
    if (a > b)
        return a;
    else
        return b;
}
```

The Header File – max.h

```
/* include all libraries here */
#include <stdio.h>

/* put any #define statements here */

/* put any structure declarations here */

/* list any needed function prototypes here */
int calcMax( int a, int b );
```

Global Variables are a Pain

- Need to declare in one (and only one) source code file
  - Allocate Memory for the variable on ONLY ONE PLACE!

```
int DEBUGMODE;
```

In Header File: Need to DEFINE (but do NOT ALLOCATE Memory)

```
extern int DEBUGMODE;
```

NOTE the Syntax for the #include statements

In the header file:

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

In the source code files:

```
#include “max.h”
```

< > ➔ Look in the system directory on your machine for the file

“ “ ➔ Look in the current directory for the file

What happens when we type: make

```
max: max1.o max2.o
	gcc -o max max1.o max2.o

max1.o: max1.c max.h
	gcc -c max1.c

max2.o: max2.c max.h
	gcc -c max2.c
```

- Top rule is checked
- Since dependencies are also targets
  - check “sub-targets” first
- So, rules are checked in this order
  - max1.o
  - max2.o
  - max

What happens when we type: make

```
max: max1.o max2.o
        gcc -o max max1.o max2.o

max1.o: max1.c max.h
        gcc -c max1.c

max2.o: max2.c max.h
        gcc -c max2.c
```

When max1.o rule is checked

- Compare time stamps of files
- If target file is older than any dependency file
  - EXECUTE THE COMMANDS!

- So, for max1.o rule:

  if max1.o is older than max1.c or max.h

  run the command:

  gcc –c max1.c

Why check Time Stamps?

- Source code time stamps change when edited
- Object files time stamps change at compilation
- So compilation needs to occur after the source code has been edited
- Executable files time stamps change at linkage
- So linkage needs to occur after the object files have been compiled