CMPSC 311
Scribe Notes
1/29/16 Lecture 8

Mohammed Jiban, Daniel Levine, Rahul Kalani, Jack Kulick

Introduction

Over the next few classes we will be finishing our discussion on basic C programming as well as other relevant utilities. A few other topics we will be covering include: System Calls, APIs, Process Creation and Execution, how to work on a UNIX machine, message sharing, shared memory, threads and synchronization, and asynchronous IO. Over the course of this chapter we will be touching on some of these topics in order to give a better understanding of what they are.

GCC

-o is used to specify name of executable

```
$ gcc -o hw hw.c
```

-Wall: helps to provide warnings that the compiler might have not had if you did not use this. Very useful for making program run cleaner.

```
$ gcc -Wall hw.c
```

-g: to enable debugging with gdb. The compiler embeds special information in the executable which the debugger will later need. Without this debugger won't be able to do its job.

```
$ gcc -g hw.c
```

-O: Compiler might have several degrees of optimization and you can specify that with this. Also can specify levels from less aggressive to more aggressive.

```
$ gcc -O hw.c
```

-c: to only create the object file

```
$ gcc -c hw.c
will create hw.o
```

C Program Files

header, source, and libraries files → program

Header Files (.h)
-header files contain types, function prototypes, constants. They are able to contain elements that are shared across modules and contain the interface exposed to external programs
E.g.: library header files
Default include path: `/usr/include`

Source files (.c)
-Source files are the core of the program modules and they Implement the functions of the program. It also contain static variables that are required only in that source file and include statements for header files to include global declarations and prototypes

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

Libraries
-These are object files created for us by the system and contain variety of utility functions and system calls.
-There are two types of files which are header and library itself
-Prototypes of functions called from program must be declared
- you need to include appropriate header files such as for string library you use `#include <string.h>`
-The standard C library which is called `libc` (all library names have prefix lib) is linked by default and other libraries have to be linked at compile time using "`-l<lib>`" flag example if you want to link library called `lib xyz` then you call it `-l<xyz>`

```
gcc –o hello hello.c –lm
```

-There are default locations where system will go looking for files
Default for headers: `/usr/include, /usr/local/include`
Default for libraries: `/lib, /usr/lib, /usr/local/lib`
When you are forced to keep your library in directory you want, you will have to specify the name of the directory. You do this by using -I is to specify non-standard paths for header files and –L is to specify non-standard paths for libraries

Dynamic Linked Libraries (.dll)
-Most commonly know as DLL
-DLL has a distinct advantage over static linked libraries because DLL doesn't get compiled with the main program

-DLL is dynamically linked with the program during execution, which saves random access memory (RAM) and DLL is only loaded if and when the DLL is needed

Makefiles

-A Makefile tells make how to compile and link a program
-Makefiles are used mostly in larger program projects to reduce the build time when only a few of the source files have changed
-When changes are made to a source file, you have to recompile each of the source files you changed, but a Makefile makes this a much more simple process
-The make command is implemented by comparing the timestamps of an object file with the similar source files. If the object file has a more recent timestamp it will be compiled. By doing this it ensures that files aren't compiled redundantly.
-A basic model of a Makefile is:

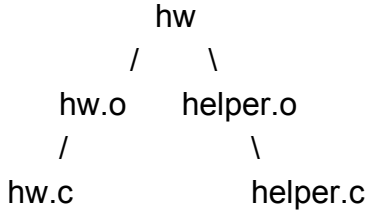
```
target: prerequisite
[tab] system command
```

-`target`: name of a file (executable or object) or action (`clean`)
-`prerequisite`: a file that serves as an input to create a target
-`system command`: an action that `make` carries out

-For example:

```
hw: hw.o helper.o
     gcc -o hw hw.o helper.o -lm
hw.o: hw.c
     gcc -o -wall -c hw.c
helper.o: helper.c
     gcc -o -wall -c helper.c
```

-This example builds a program by using a dependency tree



-The GNU version of make is `gmake`

Print Debugging

-`stdout`: Buffered; will flush the buffer when stated by the programming or convenient to the program

-`stderr`: Not Buffered; writes the message immediately

*The error message being displayed right away helps the debugging process

*Not buffered output doesn't affect the flow of the program data

-An example program that you can see `stdout` vs `stderr`

```
#include <stdio.h>
int main() {
     printf ("Hello World\n");
     fprintf (stderr, "Some error message\n");
     fprintf (stdout, "Goodbye!\n");
     return 0;
}
$./stderr > output.txt
```

Good Programming Practices

- Write code that is easily readable with much documentation to help identify code functionality. This documentation includes both comments and indentation.
    - E.g.: int *i; // pointer to an integer
- Have a good sense of coding style and use meaningful variable/function names.
    - E.g.: sort_list() vs. foo()
    - Don't use numbers to identify variables, use named constants.
        - E.g.: MAXNUM vs. 1738
    - Make sure functions are about a page long and any unrelated functions are in separate files.

Errorhandling

- It is possible for a function to return an error, which is why you should check for every possible error. To help reduce to number of errors yield to the following:
    - Don't unexpectedly exit from your functions
    - Exit from main()
- Use the system call: perror (). This makes library calls return -1, thus always check library calls for errors.
    - E.g.: if (close (fd)==-1)
      perror ("Failed to close the file");

Program Complication

- Preprocessor: program that processes the input data to produce an output which is used as an input to another program.
- Compiler: program that converts written source code into a computer language.
- Assembler: program that converts assembly code into machine code.
- Linker: program that takes one or many object files generated by the compiler and combines them into one single object file.