CS 2110 - Lab 11

Command Line & Intro to C

Wednesday, June 29, 2022

Lab Assignment: Command Line & C Quiz

1. Go to Quizzes on Canvas
2. Select Lab 11, password: **./cs2110**
3. Get 100% to get attendance!
   a) Unlimited attempts
   b) Collaboration is **allowed**!
   c) Ask your TAs for help :)

Homework 6

- Covers assembly subroutines and calling convention
- Released!
- **Due Thursday, July 7th at 11:59 PM**
- Files available on Canvas
- Submit to Gradescope (unlimited submissions)
- Will be demoed (logistics announced soon)

Homework 7

- Covers introductory C concepts
- Will be released on Friday, July 1st
- **Due Monday, July 11th at 11:59 PM**
- Files available on Canvas
- Submit to Gradescope (unlimited submissions)

Command Line

First, some terminology:

- **Shell**: a program that can interpret commands to interface with your system
  - e.g. `bash`, `sh`, `zsh`, `fish`, and many many more
- **Command line**: a shell, while running interactively (with user input)
- **Terminal**: the program you use to interact with the shell, handles text input/output

The Basics

Here's what a command line usually looks like:

the prompt

a command
(what you type)

```
nprin@nprin:~/host$ gcc -c main.c -o main.o
```

username

device name

current working directory

program name

zero or more arguments
(separated by spaces)

The shell will interpret each command you type, run it, and show you its output

Directories

- Like a file explorer, the command line has a *current working directory*
- **pwd** - "print working directory" to see where you are
- **cd** - "change directory" to move around
  - Absolute paths work from anywhere:
    - `cd /home/alice/classes/cs2110`
  - Relative paths work from your current directory:
    - `cd hw06`
- Special syntax:
  - `/` is the "root" directory of the file system, like `C:\` in Windows
  - `~` means the user's home directory, usually at `/home/<username>`
  - `.` means the current directory (why do we need this?)
  - `..` means the parent of the current directory

Which of the following commands would change the current working directory from /host/cs2110/homework7 to /host?

A. cd /host
B. cd ../..
C. cd host
D. cd ./host
E. cd ./.
F. cd /host/./../host

When reading paths in your shell, "~" indicates

Working With Directories/Files

- `pwd` – "print working directory"
- `ls` – "list" what's in a directory (the current one by default)
- `mv` – "move" a file/directory somewhere else, or rename a file
- `cp` – "copy" a file somewhere else
- `rm` – "remove" a file or directory
- `mkdir` – "make" an empty "directory"
- `cat` – view the contents of a file (short for "concatenate")

You don't need to memorize these, but practice using them!

Live Demo

Given that your current working directory is ~/host/cs2110, which of the following commands could you use to list the files in ~/host/cs2110/hw07?

A. ls hw07
B. cat hw07
C. dir hw07
D. mv hw07

Flags and Options

- Arguments that change the behavior of a command
- By convention, options will look one of two ways:
  - 2 hyphens followed by a long name, e.g. **`--recursive`**
  - 1 hyphen followed by a single letter, e.g. **`-r`**
    - These are usually abbreviations of the long form
- "Options" generally take arguments, while "flags" are usually an on/off switch
- Flag example: to view contents of a directory *including hidden files*, use `ls -a` or `ls --all`
- We'll be using `gcc` soon, which has a *lot* of different possible flags
- There's no standard around these, so different programs may have slightly different syntaxes

How To Get Help

- "How do I use `rm` to remove directories?"
- RTM: Read the manual!
- Most commands have **man pages**, which you can view with the **`man`** command
- `man rm`

Fig 2. Relevant xkcd

To look up the usage for any CLI (command line interface) command, you should look at what page?

Finding Commands

- Commands like `cat` are really just executable programs somewhere
- How does the shell know where to find them?
  - There's a special variable called the "PATH" that lists all the places to look
  - If you only type a program name without a path, it will look in these places
- What if we want to run our own program, say in the current directory?
  - We need to call it using its path, e.g. **`./hello`**
  - Just running **`hello`** isn't enough; it will check the PATH, but not the current directory
- What if the program we want to run isn't in our current directory?
  - Just give the full path to the program: **`/home/username/some-directory/hello`**

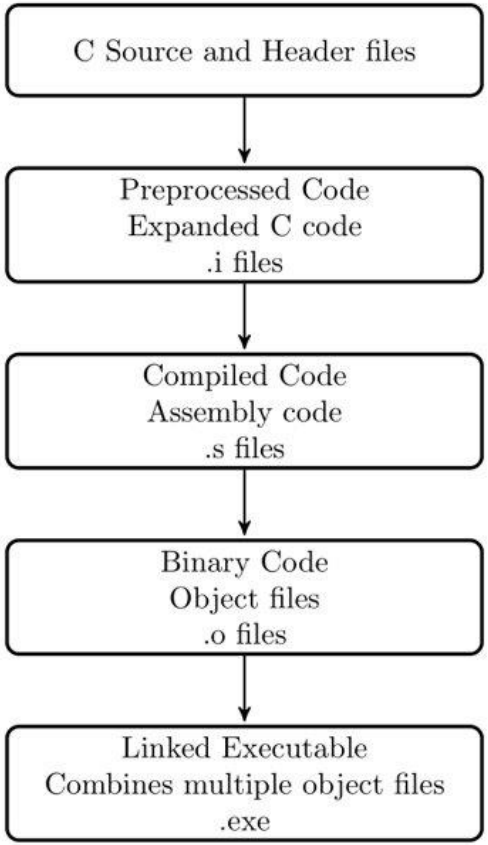
C

```c
#include <stdio.h>
#include "main.h"

int y = 0;

/* this is a comment */
int main (void) {
    int x = 3;
    y = ADD(x, 2);
    printf("%d\n", y);

    return 0;
}
```

- Developed in 1972 by Dennis Ritchie
- Like Java: compiled and statically-typed and has a similar syntax
- Manual memory management
- Low-level and procedural
    - Pretty close to assembly, but has convenient abstractions like functions
    - Not object-oriented; no classes
    - Used for systems programming, high-performance code, etc.
- Our textbook will be "The C Programming Language, 2nd edition" by K&R — available for free with your GT email! (See pinned piazza post #6)

Compilation Overview

Step 1. Preprocessor
Step 2. Compiler
Step 3. Assembler
Step 4. Linker



Macros

```
#define MACRO_NAME(ARGUMENTS) TEXT_REPLACEMENT
Ex:
#define MULT(A,B) ((A)*(B))
#define PI 3.141593
```

If preprocessor sees `MULT(5,7)` somewhere in the C file it will replace it with `((5)*(7))`

`PI` will be replaced with `3.141593`

Macro Perils

What would the following program print?

```
#include <stdio.h>

#define MULT(a, b) a*b

int main(void) {
    printf("%d\n", MULT(2 + 3, 3));
}
```

Macro Perils

Wrap every variable you use in parentheses:

```
#include <stdio.h>

#define MULT(a, b) (a)*(b)

int main(void) {
    printf("%d\n", MULT(2 + 3, 3));
}
```

Macro Perils

What would the following program print?

```
#include <stdio.h>

#define SUM(a, b) (a)+(b)

int main(void) {
    printf("%d\n", 4 * SUM(2, 3));
}
```

Macro Perils

Include outer parentheses when your macro produces an expression:

```
#include <stdio.h>

#define SUM(a, b) ((a)+(b))

int main(void) {
    printf("%d\n", 4 * SUM(2, 3));
}
```

Given the following macro definition, what is the result of the expression below?

```
#define DO_SOMETHING(A,B) (A+B*A)
DO_SOMETHING(4,2+3)
```

Header files

- In C, you can't use a function/variable before you declare it
- Header files contain function declarations and global variables
- They have a `.h` extension
- Like the "interface" of what a file exposes
- Shouldn't include function implementations

```
#ifndef MAIN_H
#define MAIN_H

#define ADD(x,y) ((x) + (y))

#endif
```

#include ...

- C does not have an "import" system
- Preprocessor copies all C code from *filename* and replaces the include statement with that code
  - `#include`-ing a header file is like copy-pasting the declarations you need
  - Generally, you should never `#include` a `.c` file—why?
- `#include <`*`filename`*`>` for system header files
- `#include "`*`filename`*`"` for header files you write

Include Guards

- What happens if I `#include` the same header file twice?
  - We'll get a compiler error due to multiple declarations
  - This can happen easily if a file includes two other files that share a dependency
- Solution: put this in our headers file to ensure each header file only gets included once:

```
#ifndef <HEADER_FILE_NAME>_H
#define <HEADER_FILE_NAME>_H
Contents of header file
#endif
```

Include Guards Example

```
#ifndef MAIN_H   /*if MAIN_H is already defined, skip to #endif*/
#define MAIN_H   /*define MAIN_H*/

// Declares that a function int square(int x) is defined somewhere
int square(int x);

#endif
```

This code ensures that `square` is only declared once even if there are multiple `#include main.h` statements

What is a good reason to use include guards?

C Data Types and Sizes

`char`	At least 8 bits (usually exactly 8)
`short`	At least 16 bits
`int`	At least 16 bits
`long`	At least 32 bits
`long long`	At least 64 bits
`float`	Unspecified, usually 32 bits
`double`	Unspecified, usually 64 bits
`long double`	Unspecified, usually 64, 96 or 128 bits

Unlike in Java, the exact sizes of number types is not specified in the C standard.

However, there are some minimums in the standard as well as some conventions. Regardless, the exact sizes are implementation-specific and may vary between compilers or systems.

One exception: POSIX (another standard) requires `char` to be 8 bits, so we can pretty much always assume it is.