CMSC 216
Introduction to Computer Systems

Akilesh Praveen
Dr. Ilchul Yoon • Spring 2020 • University of Maryland
http://cs.umd.edu/class/spring2020/cmsc216/

Last Revision: February 16, 2020

Table of Contents

---

1 Notes & Preface

This is a compilation of my notes for CMSC216 as a TA for the Spring 2020 offering of the course at the University of Maryland. All content covered in these notes was created by Dr. Ilchul Yoon and Dr. A.U. Shankar at the University of Maryland.

The actual content of this note repository is the content that I cover as a TA during my discussion section, combined with my personal insights for the course. I believe that together, these will serve as great **supplementary material** for CMSC216, but I would still highly recommend attending all of your lecture and discussion sections to achieve success in CMSC216.

The notes template in use is Alex Reustle's template, which can be found on his github at the following location: `https://github.com/Areustle/CMSC351SP2016FLN`

I maintain this repository and as such, take responsibility for any mistakes. Please send errors to `apraveen@cs.umd.edu`

2 Week 1 - Introduction to CMSC216

CMSC216 is where you learn how a computer works on a much lower level than you've experienced before. There are 3 main components that the course will explore.

Overview

- **UNIX** Threads, processes, and pipes as the building blocks of much bigger applications. We will be working with the UNIX operating system on the development environment at `grace.umd.edu`
- **C** is a high-performance language that works at a much lower level than Java. Things like memory management and advanced data structures are left up to the user. We'll cover concepts like memory management, pointers, and system calls.
- **Assembly** is even lower-level than C, and studying it will reveal how processors process instructions, store data, and maintain a stack and a heap. It's the lowest level you'll go in this class. For this semester's 216, you will be using MIPS assembly.

Grace

In this class, we will be using the `Grace` system to do all of our work. It's a little confusing to understand at first, so here's my way of thinking about it. In CMSC132, we did all of our work on our own computers. We pulled the skeleton code for the projects from the 132 website/repository, edited the code on our computers, and then uploaded our code to the submit server (via Eclipse) in order to test it.

In CMSC216, we have been given access to this big computer that UMD CS owns known as `Grace`. You, as a student, have been given a small chunk of that machine to call your own (for the semester). In this class, we will access your files on the `Grace` system using a program known as `ssh` (that's how MobaXTerm works) and do all of our editing + running code on `Grace` itself. In fact, we will also be submitting our projects from `Grace` to the UMD CS submit server.

Here are the relevant links for getting it all set up. You'll need to setup `Grace` and `gcc` (the C compiler that we'll be using within `Grace`).

- `http://www.cs.umd.edu/ nelson/classes/resources/GraceSystem.shtml`
- `http://www.cs.umd.edu/ nelson/classes/resources/setting_gcc_alias.shtml`

Useful UNIX Commands

Although the UNIX environment may seem confusing at first, learning it is essential to navigating the Grace environment. Below are some of the basic commands that you may find useful when getting started.

- **`ssh`** → If you are not using MobaXTerm, you will have to access grace using the `ssh` command. For the purpose of logging in for CMSC216, I recommend adding the `-y` flag in order to bypass the warning it will give you. E.g. `ssh -y yourdirectoryid@grace.umd.edu`
- **`ls`** → The `ls` command lists all the files in your current directory. You can use the `-l` flag to get more detailed information. E.g. `ls`, `ls -l`
- **`cd`** → The `cd` command changes the directory you're currently in, mainly to directories that you can see with `ls`. Typing `cd ..` will navigate one directory 'up' from your current directory, and `cd` without anything else will return you to your home directory. E.g. `cd 216public`
- **`pwd`** → This command displays your current directory. Useful for finding out where exactly you are in the UNIX file hierarchy. E.g. `pwd`
- **`cp`** → Copies files. If you use the `-r` flag, you're telling the command to recursively copy. If you want to use `cp` on directories, remember to use that flag.
- **`rm`** → This command stands for 'remove'. It can be used to remove singular files, or can alternatively be used with the `-r` flag to recursively remove directories. E.g. `rm hello.c`, `rm -r project1` (project1 would be a folder.
- **`.`**, **`..`**, **`~`**, and **`/`** → These abbreviations are pretty important. They can be used to navigate a filesystem in Unix and generate some clever commands. In order, they mean 'current directory', 'parent directory', 'user home folder', and 'root directory'. Below are some examples.
  - **`cp *.c ../`** → Copies all files that end with `.c` to the parent directory.
  - **`cd /`** → Changes directories to the root directory.
  - **`cp -r ~/216public/projects/project1 .`** → Recursively copies (this means that it copies directories as well as files) the project1 directory and everything in it into the current directory.

Lots of these UNIX commands are super useful once you get to know them, but it may be hard becoming acquainted with how they work from the outset. It's a far cry from the GUI you had in CMSC132, so here are a few tips.

- If you're just starting out and still need a graphical representation of the filesystem, I'd highly recommend setting up **MobaXTerm**. The program provides just a little more graphical representation than just a pure terminal, and allows you to navigate the Grace filesystem more freely. I like to think of it as training wheels as you get acquainted with Grace.
- I'd highly recommend getting used to making folders, deleting folders, deleting files, and navigating up and down through the filesystem with rapid sequences of `ls` and `cd`. As with all things, practice makes perfect, and pretty soon you'll be a command line wizard.

Machine

A computer is composed of several parts, but a great way to think about it is a few main components connected by a **bus**.

- **Memory** can just be thought of as a contiguous array of bytes. At the end of the day, this is the stuff that has to be written to/read from.
- **I/O Devices** are connected to the CPU via a bus, like mentioned above. By performing read/write operations to the right adaptor, the CPU is able to interface with different I/O devices.
- **CPU** is the central processing unit of the computer. It handles computational operations (arithmetic, logic, etc.) and interfaces with the memory and I/O devices via the bus. The CPU is also responsible for performing the **fetch-execute cycle**.

A bus is like one main connector that's responsible for making sure the CPU, memory, and I/O devices are all able to interface with each other.

Note that in this course, we won't be going too in-depth into hardware (that's more Computer Engineering), but it's great background knowledge to have as you approach this class, which is why I have included it here.

3 Week 2

The Math Library

We won't be using the math library much in C, but for the times that we do, just remember this one simple flag that we add to the gcc command. As an example, if you try to write some code that includes the math library like below, you'll find that it won't compile with a regular `gcc` command.

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

int main() {
   double value;

   printf("Enter a number: ");
   scanf("%lf", &value);     /* Notice the use of %lf */

   printf("sqrt %f: \n", sqrt(value));
   printf("power of 2: %f\n", pow(value, 2));
   printf("sin: %f\n", sin(value));

   return 0;
}
```

In this class we won't be using libraries that require this specific compilation option too much, but remember that the `-lm` flag essentially enables us to use the math library.

Using Emacs

Most of the instruction for this course will be done in `emacs`, a highly versatile text editor that you can use in GUI form or from the command line. It's always an option to use other text editors in this class, but I would recommend using `emacs`, as it's what all the in-class demos are in. There is a way to setup IDEs like Visual Studio Code to function with Grace, but I won't cover them here. I believe that although graphical IDEs have their advantages, you'll get plenty of experience with them in CMSC330 and CMSC4XX, so for now, develop your skills in a command line editor like `emacs` or `vi`.

For your benefit, here are some basic commands in `emacs` that I've found useful over the time that I took 216.

Note: When I indicate to type `M`, that means you need to press the 'meta' key. On most machines, the 'meta' key is the 'alt/option'. When I indicate to type `C`, I mean the 'control' key. The reason I'm using this notation is because it's the same notation that online guides use to describe `emacs` shortcuts.

- **`C-x C-s`** → Saves the file you're working on. Remember to do this frequently on Grace, as you can't guarantee that your connection to Grace will stay intact.
- **`C-x C-c`** → Closes the file that you're working on. If you haven't saved, it will prompt you to save.
- **`C-x u`** → Undo the previous command that you ran.
- **`C-s`** → Search forwards (this will search for text that'll be ahead of where your cursor is now.)
- **`C-r`** → Search backwards (this will search for text that'll be behind where your cursor is now.)

- **C-l** → This command will center the window around your cursor. A great technique when you have large C files that you're editing.
- **M-x column-number-mode** → Shows column numbers. Useful if you want to check if you're above the 80 character limit.

Debugging

There are three main debugging tools that we use in 216: Valgrind, GDB, and splint. For now, we won't focus too much on Valgrind, as it's more oriented towards helping programmers get rid of memory leaks and other memory-related issues. We will focus on GDB and Splint.

GDB

GDB Is the C equivalent of the Eclipse Debugger. It lets you do everything that the Eclipse Debugger allowed you to do in CMSC131 and CMSC132. The only real drawback here is that it's all done from the command line, so the graphic part of the interface is a little lacking. However, it's an essential tool that I'd highly recommend using to figure out errors in your code.

Online references will tell you that there are a lot of commands that you need to know to effectively use GDB, but here are some of the ones that I've found useful.

- **q** → exits gdb. Useful.
- **start** → starts running your code with a temporary breakpoint at the first line of main(). This allows you to set more breakpoints before the code actually starts executing.
- **l** → lists the code that you have.
- **b** → typing p with a number next to it sets a breakpoint at a line. E.g. `b 3`
- **n** → the equivalent of step over in the Eclipse debugger
- **s** → the equivalent of step into in the Eclipse debugger
- **c** → will continue running your code until the next breakpoint
- **p** → will print the value of an expression or a variable. E.g. `p valid_character('x')`.

In order to start GDB, you'll first need to compile your C code into an `a.out` file. Not only that, but I would recommend that you compile your code with the `-ggdb` flag, to ensure that GDB initializes your program correctly. In order to run GDB with your newly compiled program, remember to just type `gdb a.out`

4 Week 3

This week we go over a lot of general C-specific programming concepts in discussion, and that material is heavier than what we usually do in discussion. In that sense, I'll try and go over the more basic stuff that I think will be highly useful as you work on your projects.

Comma is an Operator

The comma in C is an operator. The best way to think about this in use is when you're declaring multiple variables at once, like when you say `int i, j = 2`.
Remember, commas are **also** used as separators in C. A great example would be if you're giving a function multiple parameters, like in `printf("%d and %d", i, j)`. When you consider the comma as an operator in C, it's always important to understand where it's an operator vs. where it's a separator.
Although we don't think about the comma operator quite a lot, one of the main reasons for understanding it would be initialization of multiple variables in a loop. Take a look at the following example from my notes (from a previous offering of the CMSC216 course).

```c
// Comma Operator Example by Nelson Padua-Perez

for (j=0, k=10; j<=limit; j++, k+= 10) {
  printf("j->%d, k->%d\n", j, k);
}
```

Notice how you initialize and increment multiple variables within a single for-loop.

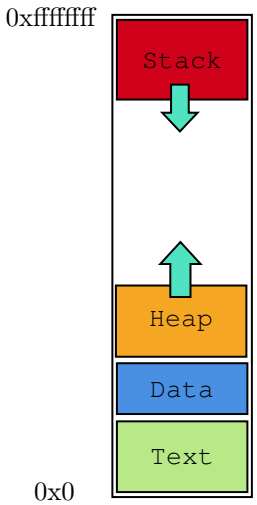
Identifier Scope

Scope exists in C in a similar way that it does in other languages. All you have to remember is that if you declare variables within code blocks, they won't be accessible outside those blocks. In that regard, this phenomenon is quite similar to how Java handles scopes.

C Program Memory Organization

As we delve deeper into systems-level programming, it's important to visualize how C actaully manages the memory that your programs use. The interesting part about this is that this diagram is an exact representation of system memory, so you're finally able to see 'under the hood' of your programs.

You can see that the lowest address is represented by `0x0` and the highest address is represented by `0xffffffff`. These addresses are actual locations in memory, represented in hexadecimal format (hence ffffffff being the highest address in the representation).



Your program is allocated a certain block of memory- within it are the following 4 components. Keep in mind that this too, is an abstraction. You can further explore how programs are represented in memory in classes like CMSC411, but this is just about as far as we'll go in 216.

- **Text** is where the code for your program goes. It's really not much more complex than that.
- **Data** is where global variables and variables that are static belong.
- **Heap** is where dynamically allocated memory lives. In Java, this stuff was managed for you. In C, you will have to manage it yourself, allocating memory and effectively increasing the size of the heap if you need more space while your program is running, and deallocating (freeing) memory to decrease the size of the heap. More on this when we discuss dynamically allocated memory.
- **Stack** is where local variables and function parameters live. It grows downwards (eventually meeting the heap and causing a stackoverflow) as functions are called. If you'll think back to 'stack frames' from recursion in CMSC132, this is the exact same concept.

Storage

There are two types of ways variables are stored in C- automatic and static. This basically goes hand-in-hand with block scopes and file scopes, but the important takeaways are these. First of all, in the example below, after the function `foo` is called, the variable `n` is thrown away.

```
int foo(int k) {
  int n = 216;
  return n;
}
```

In that regard, the variable `n` has automatic storage. If a variable has static storage, it basically exists throughout the duration of your program's running time. Such variables are initialized only once.

An important note: `static` in C does not mean the same thing as it does in Java. Here are the two main things that I think are worth remembering about static variables in C:

- Static variables need not necessarily be initialized. If you don't bother initializing a static variable (you still have to declare it- this is not Python, language of the heathen) it will automatically initialize to zero.
- Static variables retain their values between function invocations. In other words, they are not stored using automatic storage.

```
// example from Nelson Padua-Perez

void compute_static(int x) {
   static int value = 100; /* What would happen if we don't initialize it? */

   printf("(static) x: %d, value: %d, sum: %d \n", x, value, value + x);

   ++value;
}
```

In the example above, if you called `compute_static` twice, then your output would be `(static) x: 1, value: 100, sum: 101` and `(static) x: 1, value: 101, sum: 102`, as 'value' would retain its data between function calls.

Linkage

Linkage is essentially the science behind having C code spread across multiple files and making sure it all compiles and works properly.

We want to sometimes split code between multiple files for organizational purposes. Currently, the projects you're working on are small, but in order to make your programs versatile, modular and better organized, it's a great idea to split code between files.

When you attempt this, there may be issues that follow. For example, you may encounter a situation where you want to name a function `print_sum()` in two files. How would we deal with such a duplicate?

Problems of this sort can be solved by adjusting the **linkage** of these functions.

For actual code examples, please check the linkage-examples in the 216public directory. They're extremely thorough. My goal here is to provide a quick few tips on what I think are the most important parts.

Essentially, there are three types of linkage that you should remember to guide you through writing code in multiple C files.

- **`None`** → No linkage. This is how you usually declare your variables, and as you'd expect, doesn't do anything special in regards to linkage. Think of it this way: A variable with no linkage belongs to a single function, and cannot be shared. In other words, there is *only one copy per declaration.*
- **`Internal`** → Internal linkage is just a fancy way of saying you're using the **`static`** keyword. All declarations of a single identifier in file refer to the same thing. In other words, there is *only one copy per file.*
- **`External`** → External linkage is signified by the `extern` identifier, and it basically means that a name can only refer to a single entity in your entire program. In other words, there is *only one copy per program.*

5 Misc

This document will be updated frequently as we progress through CMSC216. Please send errors to `apraveen@cs.umd.edu`