CMSC 104 - Final Exam Topics

Number Systems

- Bits, Bytes, Words
  - A **bit** is a single 0 or 1
  - A **byte** consists of 8 bits
  - A **word** is 32 bits or 4 bytes. On more modern computers, a word may be 64 bits.
  - Sometimes you will here references to a **nibble**. This is just half a byte, or 4 bits.
- Number Systems - decimal, binary, and hex
  - Decimal numbers are base 10. We think of the columns (right to left) as the powers of 10 starting with 1 = 10^0.
  - Binary numbers are base 2. We think of the columns (right to left) as the powers of 2 starting with 1 = 2^0.
  - Hexadecimal is a short hand for writing binary.
    - Each hexadecimal character corresponds to a nibble (0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, 4 = 0100, 5 = 0101, 6 = 0110, 7 = 0111, 8 = 1000, 9 = 1001, A = 1010, B = 1011, C = 1100, D = 1101, E = 1110, F = 1111).
    - A byte is written as two hexadecimal characters, e.g. 3F = 0011 1111.
  - **You must know how to convert among the three number systems.**

Operating Systems and Linux

- The Operating System coordinates the CPU, memory, and I/O devices; it allows the user to communicate with the computer, controls access, and keeps track of running processes. Often just called an **OS**.
- Examples of an OS: Linux, Windows, Android, Mac OS, iOS.
- Linux provides both a Graphical User Interface (GUI) and a command-line interface (CLI).
- The Linux CLI prompt is something like "`linux1[1]%`". It can vary depending on the configuration of the computer.

- Linux CLI Overview
  - Files. A file is a sequence of bytes. Can be created in many different ways, for example using emacs. File names should not include spaces; stick with upper and lower case letters, numbers, underscore (_), and hyphen (-).
  - Directories. Directories are just special files that contain other files. They are organized in a hierarchy, like an upside-down tree. **You need to understand how to navigate the directory hierarchy**.
  - Special Directories. **Home directory** is where the OS looks for files when you first login. The **current directory** is where the OS is currently looking for files; the current directory is referenced by "." (dot). The **parent directory** is the directory one level up from the current directory; it is referenced by ".." (dot-dot). **Root** (/) is the top of the directory hierarchy.
  - Path names. Path names tell the OS where to find a file. **Absolute path names** indicate the location of a file starting at the root; they always start with /. **Relative path names** indicate the location of a file relative to the current directory.
  - Directory commands. `cd` is used to change directory; `pwd` prints the current directory; `mkdir` creates a directory; `rmdir` removes a directory; `ls` lists the contents of a directory. **Know how to use these**.
  - File commands. `rm` removes a file; `cp` copies a file; `mv` moves a file; `cat` displays the content of a file; `more` pages through the content of a file. **Know how to use these**.
  - Wildcards: in a file name, "?" matches any single character and "*" matches any number of characters. E.g. `hw?.txt` matches `hw1.txt`, `hw2.txt`, but not `hw10.txt`; `hw*.txt` matches `hw314159blargh.txt`.
  - I/O Redirection: You can use ">" to save the output of a command to a file. E.g. `ls > files.txt` creates a file called `files.txt` containing a listing of the current direcory.

Algorithms

- An **algorithm** is a finite set of unambiguous executable instructions that directs a terminating activity.
- Examples of algorithms: washing machine, Euclid's algorithm.

C Programming

- General C Stuff
  - The relationships among machine code, assembly language, and high-level languages such as C.
  - Structure of a program: program header comment, preprocessor directives, main(), other functions.
  - The **header comment** provides general documentation for the program - file name, date, author, description. Comments begin with /* and end with */.
  - **Preprocessor directives** begin with a # in column one. They help to connect your program with external programs (libraries). E.g. the `printf()` function is part of the stdio library, so code that uses `printf()` should have `#include <stdio.h>` in the preprocessor directives.
  - Every program has a **main function** called `main()` in the program. The main function is typically the first thing after the preprocessor directives. The function body is delimited by curly brackets ({ and }). The main function returns an integer, so it is declared in the function as `int main()` and the last statement in the function should be `return 0;`
  - Study the Hello, World! example. Be sure you understand the the structure of the program.
- Compiling a C program
  - The three phases:
    - Preprocessing - helps to interface your code to external code; also used to make code more portable.
    - Compilation - transforms your code into **object code**, which is machine code, but without all the parts needed for a runnable program.
    - Linking - the output of compilation is linked to external programs and the code needed to make it runnable. Produces a file called a.out (unless you specify a different name)
  - We are using the gcc compiler. To compile the program prog.c, creating an executable called prog, use the command
    - `gcc -Wall prog.c -o prog`

- The "-Wall" option generates warning messages, which are helpful for debugging.
- The resulting program (`prog`) can be run using `./prog`

- Variables
  - Know the naming rules: letters, digits, underscore; can't begin with a number; may not be a reserved word. Be able to identify illegal variable names.
  - Variable names are case sensitive.
  - Variable types: `int`, `float`, `double`, `char`
  - How variables are declared, e.g. `float pi;`
  - How variables are initialized (as part of declaration or later).
  - Using variables in simple formulas.
- Arithmetic Operators
  - +, -, *, /, % - especially keep in mind how these work for int vs. float.
  - Types and promotion - how are mixed-type expressions evaluated (int < float, short < long).
  - Operator Precedence
    - ()
    - * / % (left to right)
    - + - (left to right)
    - =
  - Using parentheses to affect order of computation or for clarity.
- Relational and Logical Operators
  - In C, false is 0, true is anything other than zero.
  - Relational operators: < , >, <=, >=, ==, !=
  - Arithmetic expressions are false or true as they are 0 or non-zero, respectively.
  - Be able to evaluate relational and arithmetic expressions.
  - The if, if/else, and nested if/else statements.
  - Watch out for = vs. ==

- Logical operators: && (AND), ll (OR), ! (NOT)
- Know the uses and truth tables for logical operators.
- Order of precedence
  - ()
  - * / % (left to right)
  - + - (left to right)
  - < <= > >= (left to right)
  - == != (left to right)
  - &&
  - ll
  - =
- Be able to evaluate logical expressions.

Loops

- while loop
  - Use for event controlled loops. For example, looping until a user enters a positive number: the program can't know how many attempts the user will need to enter something appropriate.

```
printf("Enter a positive number\n");
scanf("%d", &num);
while (num < 0) {
    printf("Enter a positive number\n");
    scanf("%d", &num);
}
```

  - Another example - reading an unknown number of grades

```
int score, num_scores = 0;
float sum = 0.0, avg;
printf("Enter a student score (-1 to end)\n");
scanf("%d", &score);
while (score >= 0) {
```

```
        sum += score;
        num_scores++;
        printf("Enter a student score (-1 to end)\n");
        scanf("%d\n", &score);
    }
    avg = sum / num_scores;
    printf("Average is %f\n", avg);
```

- Correct syntax, especially use of {}.

- do-while loop
  - Use for event controlled loops where you want the body of the loop to execute at least one time.

    In the "enter a positive number" example, we know we want to print the prompt and get input at least one time, so we can replace the while loop with a do-while loop:

```
do {
    printf("Enter a positive number\n");
    scanf("%d", &num);
} while (num < 0);
```

  - Could you replace the while loop in the "entering student grades" example with a do-while loop?
  - Correct syntax, especially use of {}.

- for loop
  - Use for counter-controlled loops, that is, loops that will repeat a known number of times (known to the program a the point where the loop occurs). For example, reading in the test scores for a known, fixed number of students.
  - Parts of a for loop: initialization, test, modification. For example

```
for (i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

    Prints the numbers 0, 1, ..., 9.

Initialization is "i=0", test is "i < 10", and modification is "i++".

- More complicated for loops. How many iterations execute?

```
for (i=0; i<100; i += 5) { ... }

for (i=1; i<=20; i++) { ... }

for (j=1; j<100; j *= 2) { ... }

etc.
```

Assignment Operators

- ++, --
- Difference between pre-increment (++x) and post-increment (x++); also pre-decrement (--x) and post-decrement (x--).
- +=, -=, *=, /=, and %=
- **Be sure to study examples / questions on slides!**

The char Data Type

- Used to store a single character (actually, the integer code representing that character)
- Use of '' to denote a character, e.g. `char c = 'X'`; assigns the code for the letter X to the variable `c`.
- Use the "`%c`" format string with `scanf()` and `printf()`, e.g. `printf("%c\n", c);`
- The `getchar()` function can be used to read individual characters from the keyboard, but remember that the return key counts as a character.

The Switch Statement

- Use of the **switch** statement and **case**s.
- Proper use of the **default** case
  - case matters (switch NOT Switch, case, NOT CASE, etc.)

- argument to switch *must* be an int variable (see below)
- "case" is followed by an integer or #define'd integer value

- Proper use of **break**
- Comparison with if and if-else (see slides)
- Example:

```
int day;
...
switch (day) {
     case 0:
          printf("Sunday\n");
          break;
     case 1:
          printf("Monday\n");
          break;
     case 2:
          printf("Tuesday\n");
          break;
     case 3:
          printf("Wednesday\n");
          break;
     case 4:
          printf("Thursday\n");
          break;
     case 5:
          printf("Friday\n");
          break;
     case 6:
          printf("Saturday\n");
          break;
     default:
          printf("Error - invalid day\n");
}
```

How would I modify this to print the message "Weekend!" if day is 0 or 6?

Functions

- Three aspects of defining and using a function:
  - Function **prototype** tells the compiler about the function - what arguments it expects and what sort of value it returns.
  - Function **definition** is the code that defines what the function actually does.
  - Function **call** is the point in the program where the function is used.
- A math example:
  - "The function *f* has domain the positive real numbers and range the real numbers." This is like the prototype - it tells you that the function *f* expects a positive real input and produces a real output.
  - "The function *f* is defined by $f(x) = \ln(x)$." This is like the function definition - it tells us what the function actually does (computes the natural logarithm). Note that *x* is just a placeholder for whatever argument is given to the function.
  - "f(3) = 1.0986" *or* "If $y = 7$, then $f(y) = 1.9459$." These are examples of function calls - they *use* the function to compute a value. Note that in math and C, the variable name used in the function definition is just a symbolic placeholder - we can call the function with any value or variable we like.
- Know how to define and use a simple function
  - General syntax, e.g. use of curly braces
  - Declaration of function parameters and their types
  - Declaration of the type of the return value of the function
  - Declaration of the function body
  - Placement of the function prototype
  - Use of the **return** statement within a function body
  - Calling a function correctly
- Relationship between parameters and arguments
  - *Parameters* are part of the definition of the function; they define the expected inputs and give them names. Parameter names are only defined within the function.

- *Arguments* are the values (constants or variables) that are passed to the function when it is called.
- When you call the function with actual constants or variables as arguments, the *values* of those constants or variables are passed to the function. Changes made to the parameters within the function body do not affect the variables in the calling function.
- For example, the following code prints the value "3". Changing `x` in the function does not affect the value of `x` in main().

```
void square(int);
int main() {
     int x = 3;
     square(x);          // call square() with argument x
     printf("%d\n", x);
     return 0;
}
void square(int x) {     // square() has one parameter, x
     x = x * x;
}
```

How would you "fix" this code so that it prints the value "9"?

- The purpose and use of header files (e.g. `stdio.h` and `math.h`).

Arrays

- An array is a group of related data items that all have the same data type and share a common name.
- A data item is known as an *element;* the elements of an array are stored contiguously in memory;
- Declaring an array: *<type> <name>*[*<number of elements>*]; for example

```
/* declares an array of five integers */
int numbers[5];

/* Declare an array of 1000 floating point numbers */
float weight[1000];

/* declare an array of 20 characters   */
/* and initialize it to "Chris Marron" */
```

```
char name[20] = "Chris Marron";
```

- Accessing array elements - [] syntax
  - For an array of `n` elements, the index (subscript) of the first element is 0, and the index of the last element is `n-1`.
  - Using the example above, the first element of the `numbers` array is `numbers[0];` the second is `numbers[1].` The last element is `numbers[4].`
  - You can read or write an array element, e.g.

```
/* print the 4th element of numbers */
printf("%d\n", numbers[3]);

/* assign the value 195 to the 10th element of weight */
weight[9] = 195;
```

  - Can use any integer expression to access an array element, e.g.

```
float data[100];
float sum;
int i;

/* Assume data is initialized here... */

/* Add every other element of data */
sum = 0.0;
for (i=0; i<50; i++) {
     sum += data[ 2*i ];
}
```

- Initializing arrays - until the elements of an array are initialized, their values are undefined.
  - Initialize short arrays with constants, e.g.

```
int numbers[5] = { 20, 21, 22, 53, 80};
```

  - Initialize data arrays with a loop, e.g.

```
int i;
float height[100];

/* initialize height array to zero */
for (i=0; i<100; i++) {
     height[i] = 0.0;
}
```

  - For our purposes, character arrays are initialized with constants, e.g. `char name[10] = "Chris";`
- Use of #define for array sizes. For example:

```
#define NUMSTUDENTS 40

int main() {
     int exam1[NUMSTUDENTS];
```

```
        etc.
}
```

- Passing arrays to functions
  - Use only the array name as argument. For example,

```
float average(float[], int);

int main() {
      float x[5] = {20.0, 21.0, 22.0, 53.0, 80.0};
      float avg;

      /* Note: only use name "x" as argument */
      /* Second argument is array length */
      avg = average(x, 5);

      printf("Average is %f\n", avg);

      return 0;
}

float average(float data[], int len) {
      float sum = 0.0;
      float avg;
      int i;

      for (i=0; i<len; i++) {
            sum += data[i];
      }

      avg = sum / len;
      return avg;
}
```

  - Functions *can* change the values of array elements passed as arguments (remember, functions can not change the value of variables passed as arguments). For example,

```
void initialize_array(float [], int);

int main() {
      float x[1000];
      float y[1000];

      /* After these calls, the elements         */
      /* of x and y are all initialized to zero. */
      initialize_array(x, 1000);
      initialize_array(y, 1000);

      etc.

}
```

```
/* initialize_array() - initialize a float array to zeros */
void initialize_array(float data[], int len) {
      int i;

      for (i=0; i<len; i++) {
            data[i] = 0.0;
      }
}
```

- Review the grade_stats.c program
  - Use of constants for array sizes
  - Declarations of multiple arrays
  - Initializing arrays in function read_scores()
  - Multiple functions with arrays as parameters

File I/O - See NOTES for Lectures 20 and 21

- File pointer `fp` is declared with `FILE *fp;` (don't have to call it `fp`).
- Reading files - open, read, close
  - Use of fopen() to open a file; "r" (read) mode
  - Use of fscanf() to read data from a file
  - Use of fclose() to close a file
- Writing files - open, write, close
  - Use of fopen() to open a file for writing; "w" (write) and "a" (append) modes
  - Use fprintf() to write data
  - Use of fclose() to close a file
- Detecting errors
  - fopen() returns NULL if it fails to open the file
  - fscanf() returns the number of items in the format string that were matched
  - fprintf() returns the number of bytes written

Command Line Arguments - See NOTES for Lecture 22

- Use of command line arguments for simple string arguments such as input and output file names
- Project gave you some practice with this