APSC 160

Headers

- Opening documentation
  - Student name `Raymond Wang`
  - Student number `xxxxxxxx`
  - Lab section `L1A`
  - UBC email `cwl@student.ubc.ca`
  - Date `January 1 2022`
  - Purpose `To determine...`
  - Input:
  - Output:
- Descriptive variable names `time` instead of `t`
- Symbolic constant `GRAV_ACCEL` instead of `9.8`
- Blank lines to separate blocks of code that perform distinct tasks
- Proper indentation
- Clear prompts for the user (include units if necessary) `Please enter the (input) in (units):`
- Output is labeled (include units if necessary) `The (output) in units is`

Code Template

```c
/*
 * Author: Raymond Wang
 * Student Number: xxxxxxxx
 * Lab Section: L1A
 * Email Address: cwl@student.ubc.ca
 * Date: [DATE]
 *
 * Purpose:
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    return 0;
}
```

Tips

- Headers

    - ```
      #include <stdio.h>
      #include <stdlib.h>
      ```
    - Only include `math.h` **if** necessary
- Input
    - Clearly asks user to input the value and includes **unit of measurement**
- Output
    - Clear output including the **units**
    - Correct number of decimals
    - Correct case
- Variables
    - Declare variables in the main function
    - Use appropriate variable type
- Branching
    - No value retesting (elegantly covers all cases)
    - Use `else if` and `else` statements
- Misc
    - Avoid code repetition

Notes

Binary, Decimal, Hexadecimal

- Binary to decimal
    - Example: `1101` binary = `1(2^3)+1(2^2)+0(2^1)+1(2^0)=1(8)+1(4)+0(2)+1(1)=13` decimal
- Decimal to binary
    - Repeatedly divide by 2, and then reverse the remainders.
    - Example: `14` decimal. 14 -> 7 -> 3 -> 1. Remainders: 0 1 1 1. Binary: `1110`
- Hexadecimal to decimal
    - Simply multiply
- Decimal to hexadecimal
    - Repeatedly divide by 16, and then reverse the remainders.

Input/Output

- Output: `printf`
    - `%[+/-]a.bf` for floating point formatting
        - `+`: always print sign of number
        - `-`: Left justify printed number with specified field width
        - `a`: field width
        - `b`: precision
      ```

    - `%c` for char
    - `%s` for stringg
- Input: `scanf`
    - `%lf` for doubles. `%d` for decimal integer. `%i` for integer. `%d` is preferred.
    - `&variable` for storing input. Ampersand is for getting memory address
    - `%*d` and `%*lf` for skipping an input

Operators

- Increment & Decrement
    - Prefix version: `--i` and `++i` evaluate to the value of the variable **after** it has been changed
    - Postfix version: `i--` and `i++` evaluate to the value of the variable **before** it has been changed

Loops

- While Loop
    - Pretest loop
        - Tests condition before iteration

```
while (condition) {
    // body of loop
}
```

    - Posttest loop
        - Tests condition after iteration

```
do {
    // body of loop
} while (condition);
```

- For Loop
    -

```
for (int i = 0; i < n; i++) {
    // body of loop
}
```

- For loop vs While loop
    - While loop is preferred for event-controlled loops
    - For loop is preferred for counter-controlled loops (particularly useful for array processing)
    - Counter of for loop `i` no longer exists after loop
- Break
    - Not preferred. Use while loop instead of for loop with break

Floating Point Values

- Can't compare directly
  - `(fabs(num - 0.3) < 0.0001)` instead of `(num == 0.3)`

File Input/Output

- File input
  - ```
    FILE* inputFile;
    inputFile = fopen("data.txt", "r");

    if (inputFile != NULL) {
        double input;
        while (fscanf(inputFile, "%lf", &input) == 1) { // check that one value was read from input
            // code
        }
        fclose(inputFile);
    } else {
        printf("Error opening input file!\n");
    }
    ```
  - Create file pointer variable
  - Open file. `"r"` is for read
  - Check that file was opened successfully
  - `fscanf` works like `scanf` but with an extra file pointer parameter. Returns number of values that were read and stored successfully
  - Remember to close file
- File output
  - ```
    FILE* outputFile;
    outputFile = fopen("results.txt", "w");

    if (outputFile != NULL) {
        fprintf(outputFile, "Hello World!");
        fclose(outputFile);
    } else {
        printf("Error opening output file!\n");
    }
    ```

Functions

- Include function documentation

```
/*
 * Purpose: [What the function does]
 * Param: [variable] - [what the variable stores]
 * Return: [what the function returns]
 * Assumption: [Assumptions]
 */
```

- Include function prototype

```
void test(int[], int);
```

  - Variable names are optional but not needed (they are ignored by the compiler)
  - They do not need to be the same as the parameters in the function

Arrays

- Array declaration with primitive type: `type arr[size]`
- Element access: `arr[i]`
  - Arrays are **0-indexed**
- Array initialization: `type arr[size] = {a, b, ...}` or `type arr[] = {a, b, ...}`

```
int age[50];
double powerOutput[1400];

age[0] = 10;
age[1] = 11;
age[2] = 12;

int data[] = {3, 8, 2, 5, -1, 9};
int SIZE = 6;

for (int i = 0; i < SIZE; i++) {
    printf("%d ", data[index]);
    index++;
}
```

- Arrays are **passed by address** (memory address) when they are passed as parameters to a function
- Arrays cannot be returned. Instead, pass the array as a parameter, and then have the function modify it
- Unsupported operations on arrays
  - Arrays cannot be the target of an assignment. Instead, copy array by iterating with loop
  - Arrays cannot be the return type of a function. Instead, pass an array as a parameter

Multidimensional Arrays

- 2D array declaration: `type arr[numRows][numCols]`

- Element access: `arr[i][j]`
  - **0-indexed**
- Array initialization:

```
int data[][3] = {{3, 1, 5}
                 {4, 0, 2}};
```

  - Necessary to specify number of columns, but not rows
  - 2D arrays are stored contiguously in memory
- Array processing: use `for` loops

```
int data[3][2];
```

Strings

Character Arrays

- Array of type `char` . i.e. `char filename[4] = {'t','e','s','t'}`

String

- A character array where the last array element is the null character `'\0'`
  - Null character has ASCII equivalent of 0
  - If a character array has a null character in the middle, treating it as a string would mean anything after the null character is ignored
- String constant
  - For example, `"sensor.txt"` or `"r"`. Last element is implied to be null character
  - Single quote for character constant, double quote for string constant
- Declaration and initialization
  - `char filename[11] = "sensor.txt";`
  - `char filename[] = "sensor.txt";`
  - `char filename[] = {'s','e','n','s','o','r','.','t','x','t','\0'}`
  - The above are equivalent. Notice that the size includes the null character
  - When assigning a string constant to a character array, extra spaces are filled with `'\0'`
- Print string
  - `printf` with `%s` format specifier. Field width (for example `%10s`) also works
  - For example, `printf("%s", label)` where `label` is a string
- Read string
  - `scanf` with `%s`. Note that there is no ampersand for the string
  - ```
    #define LINEMAX 81
    char line[LINEMAX];

    scanf("%s", line);
    ```
    ```

    - Default: reads up to next line or next space
    - Specify number of characters to read: `scanf("%4s", line)`
- User defined functions
    - Null character is helpful
    - `int myStrLen(char s[]);`
- Standard functions in C for strings. Do `#include string.h`
    - `strlen(s)`: returns the useful length of `s` (excludes null character)
    - `strcpy(s, t)`: copies string `t` to string `s`
    - `sprintf`: print to a string
        - `sprintf(destString, formatSpecifier, var1, ..., varN);`