CSE320 : System Fundamentals II Assignment 2

DUE: 23-Sep-2024

Basic C Programming

Objectives

This assignment will:

1. reinforce your understanding of essential C programming concepts. You will work on a series of tasks that cover topics such as C data types, enums, pointers, arrays, structures, file I/O, functions, macros, and the C standard library.

Background

C is a high level language with many useful but intricate low-level features. Mastering some of these features will help the student use the language effectively for system tasks.

Instructions

You should have an Ubuntu virtual machine configured on the AWS site. If not, see me as soon as possible. Find your instance and 'start it' so it is running. Note that after each session, you should stop your virtual machine so it is not running constantly. **If you exceed a certain number of hours, you may move out of the 'free tier' and it will charge you for the usage!**

Part 1: C Types

Task 1: C Data Types and Enum types

Goal: Create a C program that defines and demonstrates the use of custom data types using `typedef`.

1. Implement an enumeration type (`enum`) to represent the days of the week (from Sunday through Saturday).
2. Write a function (***printDay***) to print the day of the week on the console that corresponds to a given enum value. The function is passed an enum value. It prints the name of the weekday corresponding to the enum value on the console. Note: The enum type should use the 3 character day abbreviations (except Thursday's should be Thur).

Task 2: Pointers and Arrays

Goal: Build a C program that uses and manipulates arrays using pointers rather than array notation.

1. Implement a function (sumAndAverage) that finds the sum and average of an integer array. It should take 3 arguments:
   a. A pointer to the array of integers
   b. A count of the number of integers in the array
   c. A pointer to an array of integers where the results will be returned. This array should be created by allocating space for 2 double values with malloc inside the sumAndAverage function. The address of this array should be placed in the 3rd argument. This will act as a 'return value'. Hint: you will need to specify the last argument with a 'double pointer' (**).
   d. Upon return, the first element in the returned array should contain the sum of the integers in the array passed in (first argument) and the second element will contain the average of those values.

2. Write a program to test your function with user-defined arrays (i.e. read the array elements from the console.) Hint: Ask the user for the number of values to be summed/averaged so you know how much space to allocate.

Part 2: File Manipulation and I/O

Task 3: C Structures and File I/O

Goal: Create a C program that can store related data in a structure and write the data to or read it from a binary file.

1. Define a C structure to represent an automobile. Include fields with the attributes manufacturer, model, manufacture year, color, and list price (in KRW).
2. Write a program that creates an array of the automobile structures. Ask the user how many automobile's data they wish to store. Read that as an integer. You can use that to statically declare an array of automobile structures.
3. Implement separate file I/O functions to read and write the product data from/to a binary file. You can name these functions anything but pick something that suggests their function (i.e. read_auto_array/write_auto_array).
4. The write function will take as arguments the name of the data file, a pointer to an array of structs, and a count of structs in the array.
5. The read function will take the name of the data file, an array of 100 entries (if the count exceeds 100, only read the 1st 100 auto's data), and a pointer to an integer where the number of autos read will be placed. So if the file contains data for 5 autos, it will store 5 at the location pointed to (which will be in the calling routine's local variables.) If the file contains data for 105 autos, it will fill the array with data for the 1st 100 vehicles and write 100 into the location pointed at by the third argument.
6. Have your program read the data from the user off the console and store it in the array of automobile structs.
7. Now, have your program write the entire array of structures into a file called ***auto.data*** using the function you created to write automobile data to a file**.** Hint: you may want to start the file with an integer value that indicates the number of vehicles stored in the data file. Then, when you read it back, after reading the

initial integer which is a count of the automobiles in the file, you know how much space to allocate.

8. Finally, have the program read back all of the data (using the read function you created) from ***auto.data*** and then verify what is read matches what was written.

Part 3: C Functions, Macros, and Preprocessor Directives

Task 4: Functions and Function Pointers

Goal: Create a program that performs simple math calculations (add, subtract, multiply, square, and cube) with separate functions.

1. Create a typedef that describes a pointer to a function which takes two integer arguments and returns an integer.
2. Create basic math functions for addition, subtraction, multiplication, square, and cube that follow the typedef. Put the math functions in a file called **math_operations.c**. Also, create a header file (**math_operations.h**) to be included in any c module that you will call the math functions from.
3. Develop a calculator program (***calculator.c***) that allows users to choose and perform arithmetic operations from a menu (i.e. 1. Add, 2. Subtract, etc.). Prompt the user for 2 integers (for add, subtract and multiply) and read them from the console.
4. Finally, use a function pointer to execute the correct math function and have it return a result. The code for the actual functions add, subtract, and multiply are in math_operations.
5. Print the result to the console.
6. The **calculator.c** module provides a single entry point (calculator()) that takes no arguments and may be called from your main program.

Task 5: Macros and Preprocessor Directives

Goal: Learn how to properly develop and use macros

1. Define macros to handle two additional common mathematical operations: square and cube.
2. Add the macros to math_operations.h (but you don't need to write code in math_operations.c since these are macros and simply expand text.)
3. Add to your calculator program (***calculator.c***) so it includes those operations in the list offered to the user. For these two operations, it should prompt the user for a single integers since square and cube take only 1 argument.

Part 4: C Library Functions

Task 6: Exploring C Library Functions

Goal: Explore and utilize standard library functions from stdio (`stdio.h`), stdlib (`stdlib.h`), and string (`string.h`).

1. Create a program that asks the user for the name of a text file and then opens and reads the text file, counts the number of words, and displays the most common word.
2. Implement error handling for file I/O operations.

Part 5: Integration

Task 7: Integration and Documentation

Goal: Understand how to integrate different collections of code into a single application.

1. Combine all the tasks into a single, well-organized C program.
2. For each of the above tasks, before running the code for them, have your main program output a banner like:

   ```
   ============
   = Task 1 =
   ============
   ```

   Note: Tasks 4 and 5 can be done together with 1 call to calculator ().
3. Document your code thoroughly with comments explaining each section.
4. Create a Makefile that builds each of the individual applications from the above tasks as well as building your file integrated C program from this section. [Except for the **calculator.c** file which presents a menu to the user, all of the other simpler operations can be placed in a single file (**assignment2.c**).
5. Write a brief report summarizing the challenges use faced, and what you've learned during the assignment.

Deliverable Files: assignment2.c, calculator.c, math_operations.c, math_operations.h, Makefile, report.txt

Follow the submission instructions below.

Submission Instructions

The deliverable files for this assignment are each of the source files (including your header files) from each task along with a Makefile that builds all of the applications.

Please follow this procedure for submission:

1. Place the deliverable files into a folder by themselves. The folder's name should be CSE320_HW2_<yourname>_<yourid>. So if your name is Alice Kim and your id is 12345678, the folder should be named 'CSE320_HW2_AliceKim_12345678'

2. Compress the folder and submit the zip file.
    a. On Windows, do this by clicking the right-mouse button while hovering over the folder. Select 'Send to -> Compressed (zipped) folder'. The compressed folder will have the same name with a .zip extension. You will upload that file to the Brightspace.
    b. On Mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Brightspace.
3. Navigate to the course Brightspace site. Click **Assignments** in the top navbar menu. Look under the category 'Assignments'. Click **Assignment2.**
    a. Scroll down and under **Submit Assignment**, click the **Add a File** button**.**
    b. Click **My Computer** (first item in list).
    c. Find the zip file and drag it to the 'Upload' area of the presented dialog box.
    d. Click the **Add** button in the dialog.
    e. You may write comments in the comment box at the bottom.
    f. Click **Submit**. 🡐 Be sure to do this so I can retrieve the submission!

Grading Criteria

The assignment will be graded for the following items:

Assignment submission follows all directives (includes required files, Makefile, etc., is backaged in a zip file…) [10% of points]

Code builds without error or warning [50% of points]

Code works and provides correct results without crashing [30% of points]

Code is well commented and consistently formatted [10% of points]