Unit 3 Sample Problems - C Programming III

In this sample problem set, we will practice advanced concepts in the C programming language.

- Length: 1:15 minutes with discussion.
- Questions: Q1-Q2.

Learning Objectives:

1. Understand how the preprocessor works in C.
2. Apply multiple source files, the preprocessor, and the linker to construct modular programs.
3. Apply procedural programming techniques to perform object-oriented programming.
4. Apply a file format specification and a file I/O library to read and write binary files.

The Preprocessor

1. [Lisonbee] Below is some partially implemented C code. Using it as a template, create macros based on the instructions given in the comments.

```
// Create a macro called AMOUNT_1 and assign it any integer value.
#define


// Create a macro called AMOUNT_2 and assign it any integer value.
#define


// Create a macro called DIFFERENCE and assign it's value to be the
// result of AMOUNT_1 minus AMOUNT_2.
#define


// Determine if AMOUNT_1 is greater than AMOUNT_2 using DIFFERENCE (if
// DIFFERENCE is greater than 0, then AMOUNT_2 is larger, vice versa).
#if (                                                              )

  //If this condition is true, define a macro called OUTPUT and assign
  //it a value of 1.
  #define

#else

  //If this condition is false, define a macro called OUTPUT and assign
  //it a value of 0.
  #define

#endif
```

Object-Oriented Programming

2. [Lisonbee] Implement a string ADT using a struct and six functions. At its most simple, a string is an array of characters that is terminated by a null terminator. Although having a dedicated string type is generally an OOP concept, we can still implement it in C. The struct should use variables to represent an internal data structure, and the length of the string. Create six functions: string_create, string_destroy, string_append, string_substring, string_length, and string_display. Descriptions of each are shown on the next page. NOTE: you may only use stdlib.h and stdio.h for this problem; you should include them in your header file.

   Your ADT will be structured with three files: main.c, string.h, and string.c. The first file (main.c) is provided below, while you will need to implement string.h/c. The main.c file contains testing code that uses the string you will be implementing - it indirectly shows the syntax for the functions.

   main.c

```
#include "string.h"

int main() {
    string str, sub_str;
    str = string_create("this is a string");

    printf("String contents: ");
    string_display(str);
    printf("Length: %d\n", string_length(str));

    string_append(str, ", I think...");
    printf("String contents: ");
    string_display(str);
    printf("Length: %d\n", string_length(str));

    sub_str = string_substring(str, 18, 25);
    printf("String contents: ");
    string_display(sub_str);
    printf("Length: %d\n", string_length(sub_str));

    string_destroy(str);
        string_destroy(sub_str);
    return 0;
}
```

   Output:

```
String contents: this is a string
Length: 16
String contents: this is a string, I think...
Length: 28
String contents: I think
Length: 7
```

string.h

```
#ifndef STRING_H
#define STRING_H

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

////////////////////////////////////////////////////////////////////
//Type Definition



////////////////////////////////////////////////////////////////////
//Function Declarations

//purpose: creates a new string and returns it.
//return: new string



//purpose: destroys a string.
//return: n/a



//purpose: adds a new string onto the existing string
//return: n/a



//purpose: returns the substring of a given string from
  the start index (inclusive) to the end index (exclusive).
//return: new string that is the proper substring



//purpose: returns the length of the string.
//return: number of characters in the string



//purpose: displays the string.
//return: n/a



#endif
```

string.c

```
////////////////////////////////////////////////////////////////////////
//Include File

#include "string.h"


////////////////////////////////////////////////////////////////////////
//Function Declarations
```

Handling Binary Files

3. [Lisonbee] Consider a custom file format for storing experimental data sequentially (binary). Knowing only what's given below about this hypothetical file format, implement a function that can read this format (using the fread function) and return the results from a given trial of an experiment.

   - File Header (64 bytes)
     - File size (64 bytes)
   - Experimental Header (40 bytes)
     - Number of experiments (16 bytes)
     - Number of trials per experiment (8 bytes)
     - Size of trial data (16 bytes)
   - Experimental Data (2D array based on number of experiments and trials per experiment)
     - Each row begins with the experiment ID (16 bytes), followed by the data from each successive trial (16 bytes)

```
uint16_t get_results(FILE* file_in, uint16_t experiment_ID, uint16_t trial_num) {









}
```