Assignment 7
C/C++ Programming I

C1A7 General Information

--- No General Information for This Assignment---

Get a Consolidated Assignment 7 Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment, send an empty email to the assignment checker with the subject line **C1A7_*ID***, where ***ID*** is your 9-character UCSD student ID. Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.

C1A7E0 *(6 points total - 1 point per question – No program required)*

Assume language standards compliance and any necessary support code unless stated otherwise. Testing erroneous or implementation dependent code by running it can be misleading. These are not trick questions and each has only one correct answer. Major applicable course book notes are listed.

1. Passing an entire structure or class to a function rather than a pointer or reference to it:
   (Note 9.9)
   A. is usually much more efficient.
   B. permits the function to modify its original members.
   C. makes code more portable.
   D. is usually an arbitrary choice that doesn't affect efficiency.
   E. none of the above

2. Which is true for the following code?
```
short *sp = (short *)malloc((short)37);
short *save = sp;
free((void *)save);
```
   (Note 8.4)
   A. Memory for 37 **short**s is requested.
   B. *malloc*'s argument value must be even.
   C. There's a major problem related to the call to *free*
   D. *malloc*'s typecast should be (**void** *).
   E. none of the above

3. Identify the data type of the array elements and predict the output, respectively:
```
#define Elem(A) (sizeof(A)/sizeof(*(A)))
char *fmt[] = {"%i ", "%d ", "%o ", "%x"};
for (int idx = 0; idx < Elem(fmt); ++idx)
    printf(fmt[idx], 15);
```
   (Notes 6.1, 1.11)
   A. **char **** & *15 15 17 f*
   B. **char *** & *15 15 015 0x15*
   C. **char *** & *15 15 17 f*
   D. **char** & *15 15 17 f*
   E. **char *[4]** & implementation dependent

4. In C++, given the declaration
```
struct par test;
```
   which of the following does the type of the argument passed to function *f4* match the type of the parameter specified in the prototype to the left of it?
   (Note 6.1)
   A. **char** f4(**struct** par *); f4(*test)
   B. **char** f4(par *); f4(&test)
   C. **struct** par *f4(**char**); f4(&test)
   D. **char** f4(**struct** par &); f4(&test)
   E. none of the above

5. Select the 3 additional *printf* arguments that will output ***old bread pie***
```
const char *p[] = {"cook some good old",
    "cornbread", "and magpie"};
printf("%s %s %s", 3 arguments);
```
   (Notes 6.16, 7.3, 8.1, 8.2)
   A. &p[0][15] &*(p+1)[4] &p[2][7]
   B. &p[2][15] &*((*(p+2))+4) &p[2][7]
   C. &*(p+0)[15] &p[1][4] &p[2][7]
   D. &p[0][15] &p[1]+4 &(*(p+2))[7]
   E. &15[p[0]] p[1]+4 &(*(p+2))[7]

6. What is the most serious problem?
```
char ch = 'A';
int *ptr = (int *)malloc(128 * sizeof(int));
*ptr = ch;
if (!ptr)
    exit(EXIT_FAILURE);
```
   (Note 8.4)
   A. *malloc* returns a **void** pointer.
   B. A null pointer might get dereferenced.
   C. (**int ***)malloc must be (**char ***)malloc.
   D. **ptr* references uninitialized memory.
   E. **ptr* = *ch* should be (**char**)**ptr* = *ch*.

Submitting your solution

Using the format below place your answers in a plain text file named **C1A7E0_Quiz.txt** and send it to the assignment checker with the subject line **C1A7E0_*ID***, where ***ID*** is your 9-character UCSD student ID.

```
-- Place an appropriate "Title Block" here --
1. A
2. C
etc.
```

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

C1A7E1 *(7 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add three new ones, naming them **C1A7E1_MyTime.h**, **C1A7E1_DetermineElapsedTime.cpp**, and **C1A7E1_main.cpp**. Do not use `#include` to include either of the two **.cpp** files in each other or in any other file. However, you may use it to include any appropriate header file(s) you need and you must include **C1A7E1_MyTime.h** in any file that needs data type `MyTime`.

C1A7E1_MyTime.h must be protected by an include guard (note D.2) and must define data type `MyTime` exactly as shown below:

```
struct MyTime {int hours, minutes, seconds;};
```

C1A7E1_DetermineElapsedTime.cpp must contain a function named `DetermineElapsedTime` that computes the time elapsed between the start and stop times stored in the two type `MyTime` structures pointed to by its two parameters, stores it in another `MyTime` structure, then returns a pointer to that structure. For example, if the start time is 03:45:15 (3 hours, 45 minutes, 15 seconds) and the stop time is 09:44:03, `DetermineElapsedTime` computes 05:58:48. **IMPORTANT:** If the stop time is less than or equal to the start time, the stop time is for the next day.

Function `DetermineElapsedTime` must:

- Have only two parameters, both of data type "pointer to **const** `MyTime`"
- Not modify the contents of either structure pointed to by its two parameters
- Not declare any pointers other than its two parameters
- Not declare any structures other than the one that will hold the elapsed time
- Return a "pointer to `MyTime`" that points to a `MyTime` structure containing the elapsed time
- Not prompt or display anything

You may convert times to seconds within `DetermineElapsedTime` if you wish, but beware of possible data type overflow if you do.

C1A7E1_main.cpp must contain a function named `main` that contains a "`for`" statement whose body gets executed 3 times. No other looping statements are permitted. The following must be done in order during each execution:

1. Prompt the user to enter the start and stop times in that order, space-separated on the same line. Each must be in standard HH:MM:SS 2-digit colon-delimited format and must be input directly into the appropriate members of two `MyTime` structures;
2. Call `DetermineElapsedTime` passing pointers to the two structures containing the user-entered times, then store the pointer it returns in a type "pointer to `MyTime`" variable;
3. Display the user-entered times and the elapsed time in the standard HH:MM:SS 2-digit colon-delimited format shown below. Use the pointer variable from the previous step to access the elapsed time:

   `The time elapsed from HH:MM:SS to HH:MM:SS is HH:MM:SS`

Function `main` must:

- Not contain any "`if`" statements or "`?:`" expressions.
- Not declare more than one pointer variable and two structure variables. Additional non-pointer and non-structure variables are okay.

- Use military time for both input and output, that is: 23:59:59 is 1 second before midnight; 00:00:00 is midnight; 12:00:00 is noon.
- Use no non-constant external variables, including external structure variables.
- Use no dynamic storage allocation.

Test with at least the following three *start/stop* time pairs:

00:00:00 00:00:00 12:12:12 13:12:11 13:12:11 12:12:12

Submitting your solution

Send your three source code files to the assignment checker with the subject line **C1A7E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

Handling the colon delimiter in the HH:MM:SS input time format

In real life programming you would need to carefully parse the input to make sure it was in exactly the required format, and you may do so for this exercise too if you wish. However, the following minimalist approach is satisfactory for this course. Assuming that `tm` is a `MyTime` structure and `delim1` and `delim2` are each type **char** variables, simply retrieve the user input as follows:

```
cin >> tm.hours >> delim1 >> tm.minutes >> delim2 >> tm.seconds;
```

After the input is done you may check `delim1` and `delim2` to see if they each contain a colon character if you wish.

Where to store the elapsed time

Declare a **static** `MyTime` structure in `DetermineElapsedTime`, store the elapsed time in it, then return its address.

Potential integer overflow

Beware of potential integer overflow during multiplication and other operations! The maximum value type **int** can represent on some implementations is 32767. Any operation that attempts to store a value larger than 32767 in a type **int** object or that performs an operation with type **int** operands that could result in a value larger than 32767 being produced is not portable and is not allowed in this course. Simply assigning the result of an overflowing operation into a type **long** variable does not solve the problem since the overflow occurs before the assignment occurs.

If you approached this exercise by converting all times to seconds, you probably multiplied the number of hours by the number of seconds in a day. Since the maximum number of hours is 23 and the number of seconds in an hour is 3600, the result of the multiplication could be as large as 23 * 3600, which is 82800 and obviously exceeds 32767. You must also consider overflow when specifying or calculating the number of seconds in a full day. (See notes 2.10 and 2.11)

Returning a pointer to an automatic object

Returning a pointer or reference to an automatic object is always wrong, as is dereferencing an uninitialized pointer or a null pointer (See note 6.12).

Producing 2-digit time formats with a leading 0

Look up the `setfill` manipulator in your IDE's help, in any C++ textbook, or online. This manipulator is "sticky", meaning that once set it remains in effect until set again.

Other

1. A common student mistake is to produce a difference of 00:00:00 if both times are equal.
2. Be sure to use "include guards" (note D.2) in header file **C1A7E1_MyTime.h** and include it in files **C1A7E1_DetermineElapsedTime.cpp** and **C1A7E1_main.cpp**.

C1A7E2 *(7 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A7E2_main.c**. Write a program in that file to obtain nutritional information about several foods from the user then display a table containing this information.

The number of foods to be displayed is determined by the value of a macro named `LUNCH_QTY` that you must define and the information about each food is kept in a structure of the following type (The data types and member names must not be modified):

```
struct Food
{
    char *name;             /* "name" attribute of food */
    int weight, calories;   /* "weight" and "calories" attributes of food */
};
```

Function `main` must, and in the order specified:

1. **in one single statement**:
    a. define the **`struct`** Food data type shown above and in the same statement
    b. declare automatic array `lunches[LUNCH_QTY]`, and in the same statement
    c. explicitly initialize only elements `lunches[0]` and `lunches[1]`, initializing them to an apple and a salad, respectively. Assume that an apple weighs 4 ounces and contains 100 calories and a salad weighs 2 ounces and contains 80 calories.
2. loop through each of the non-explicitly-initialized elements of the array and do the following in order during each iteration:
    a. prompt the user to enter the whitespace-separated name, weight, and calories of a food in that order on the same line; the food name must not contain whitespace;
    b. store the food name into a temporary character buffer you've declared and store the weight and calories values directly into the corresponding members of the structure in the current `lunches` array element;
    c. determine the exact amount of space necessary to represent the food name including its null terminator character;
    d. dynamically allocate the exact amount of memory determined in the previous step and store the pointer to it in the `name` member of the structure in the current `lunches` array element. Do not use `calloc` or `realloc`. If dynamic allocation fails output an error message to `stderr` and terminate the program with an error code.
    e. Copy the food name into the dynamically allocated memory using the `memcpy` function.
3. display a table of all foods in the array along with their weights and calorie content, aligning the left edges of all foods and the least significant digits of all weights and calories. **There must be nothing between these entries except the spaces needed for alignment (no commas, dividing lines, etc.).**
4. free all dynamically allocated memory.

Your code must work for any value of macro `LUNCH_QTY` greater than or equal to 2 as well as for cases where the *weight* and *calories* are both 0. Manually re-run your program several times, testing with different values of `LUNCH_QTY` and different foods.

Submitting your solution

Send your source code file to the assignment checker with the subject line **C1A7E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

How to define a structure type and declare and initialize an array of them in 1 statement:

Here is an example in which all members of the first 3 structures in an array of 4 structures named boxes are initialized explicitly, while all members of the last structure in that array are initialized implicitly:

```
struct Box
{
   int height, width, depth, weight;
} boxes[4] = { { 8, 5, 7, 100 }, { 2, 9, 1, 4 }, { 26, 78, 16, 99 } };
```

Freeing memory

In this exercise you must individually dynamically allocate separate blocks of memory for each of the foods input by the user. As a result you must also individually free each of them before the program terminates.

Testing dynamic memory allocations

Failing to test for successful dynamic memory allocations always is an error.

Uninitialized pointers

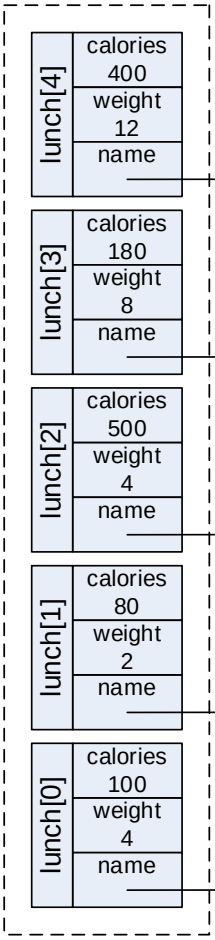
Simply declaring a pointer does not make it point to a valid location. All pointers must be explicitly initialized before dereferencing. In this exercise the three uninitialized name pointers must be made to point to a usable area of memory before the food names are stored. Dereferencing uninitialized pointers often causes core dumps (crashes) or even more subtle problems.

Pointers and Memory Diagrams

As with all exercises involving pointers, if you have any doubts or problems you should draw one or more diagrams of relevant memory objects showing how they are affected by the various program steps. In many cases it is beneficial to first draw a diagram of what those objects should look like when your program has completed its primary task. This will allow you to then step through your code and verify that it actually produces that configuration. On the next page I have drawn this "finished" memory diagram for you for some typical user food inputs...

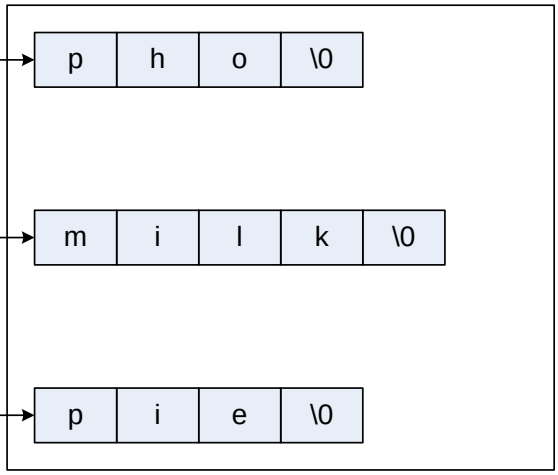
Memory Diagram for “C/C++ Programming I”, Assignment 7, Exercise 2
after all user input has been completed (5 lunch items).

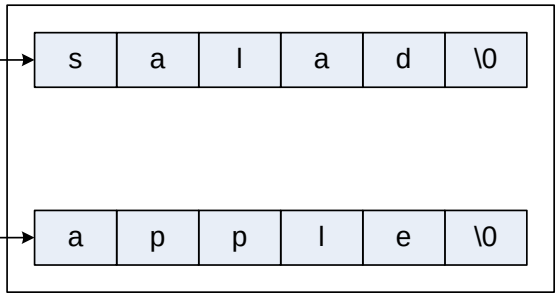
The exact amount of memory needed for this entire array was allocated by the compiler based upon the number of elements and their data type.



Array “lunch”
Each element is of type “struct Food”

The exact amount of memory needed for each of these three strings was dynamically allocated by the program at run time based upon the individual strings input by the user.





The exact amount of memory needed for these two strings was allocated by the compiler based upon the strings in the array initializer list.