Assignment 7
C/C++ Programming I

C1A7 General Information

Assignment 7 consists of THREE (3) exercises:

C1A7E0 C1A7E1 C1A7E2

All requirements are in this document.

Related examples are in a separate file.

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-body email to the assignment checker with the subject line **C1A7_174273_U09845800** and no attachments.

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.

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

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

Assume language standards compliance and any necessary standard library support unless stated otherwise. These are not trick questions and there is only one correct answer. Basing an answer on actual testing is risky. Place your answers in a plain text "quiz file" named **C1A7E0_Quiz.txt** formatted as:

a "Non-Code" Title Block, an empty line, then the answers:
1. A
2. C
etc.

1. Identify the type of the elements of array **y** and predict a possible output, respectively:
```
#define Elem(A) (sizeof(A)/sizeof((A)[0]))
const char *y[] = {"%d  ", "%i  ", "%o  ", "%x"};
for (int ix = 0; ix < Elem(y); ++ix)
   printf(y[ix], 0xD);
```
(Notes 6.1, 1.11)
A. **char *[4]** & implementation dependent
B. **const char** & 13 13 15 d
C. **char *** & 13 13 015 0xd
D. **const char *** & 13 13 15 d
E. **const char **** & 13 13 15 d

2. Passing an entire structure or class to a function rather than a pointer or reference to it:
(Note 9.9)
A. is usually more efficient.
B. is usually less efficient.
C. will cause a compiler error.
D. permits the function to modify its original members.
E. should always be the first choice.

3. If the ASCII character set is used, what is a serious problem:
```
short *ptr = (short *)malloc(sizeof(short));
if (ptr)
    *ptr = 'M';
```
(Notes 8.4, B.1)
A. *(**short ***)malloc* should be *(**char ***)malloc*.
B. **ptr* is of type **short**.
C. **sizeof**(**short**) bytes may not be enough to represent the value of 'M'.
D. *malloc* is not tested for success/failure before the allocation is accessed.
E. There is no serious problem.

4. Which is true for the following code?
```
int *ip = (int *)malloc(87 * sizeof(int)) + 6;
free(ip);
```
(Notes 6.14 & 8.4)
A. *ip* must be type **void ***
B. *malloc*'s argument value must be even.
C. *calloc* is usually faster than *malloc*.
D. If allocation succeeds, *free(ip)* frees it.
E. There is a major problem related to the call to *free*

5. In C++, given the declaration
```
class fog xy;
```
which of the following does the type of the argument passed to function *f3* match the type of the parameter specified in the prototype to the left of it?
(Notes 5.9, 6.1, 9.11)
A. **int** f3(fog &); f3(xy)
B. **int** f3(**class** fog &); f3(&xy)
C. **class** fog *f3(**int**); f3(&xy)
D. **int** f3(**class** fog *); f3(xy)
E. None of the above.

6. Which expressions must be in positions **1**, **2**, and **3**, respectively, in the *cout* statement below to output ***raid the big gray wolf***
```
const char *p[] = {"who's afraid",
    "of the big", "bad gray wolf"};
cout << 1 << " " << 2 << " " << 3;
```
(Notes 6.16, 7.4, 8.1, 8.2)
A. &p[0][8] &*(p+1)[3] &p[2][4]
B. &p[2][8] &*((*(p+2))+3) &p[2][4]
C. &*(p+0)[8] &p[1][3] &p[2][4]
D. &p[0][8] p[1]+3 &(*(p+2))[4]
E. None will do it portably.

Submitting your solution

`Send an empty-body email to the assignment checker with the subject line **C1A7E0_174273_U09845800** and with your quiz file attached.

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) 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 type `MyTime` exactly as shown below and contain a prototype for function `DetermineElapsedTime`.

```
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 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 one that will hold the elapsed time.
- Not prompt or display anything.
- Return a "pointer to `MyTime`" that points to a `MyTime` structure containing the elapsed time.

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 the time values must be input directly into the appropriate members of two `MyTime` structures.
2. Although a "real life" program would require that you carefully parse the input to ensure proper formatting, use the minimalist approach below for this exercise. `start` is a `MyTime` structure and `delim` is a type **char** variable whose value you must ignore:

   ```
   cin >> start.hours >> delim >> start.minutes >> delim >> start.seconds
   ```

3. 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.
4. 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 "`if`" statements or "`?:`" expressions.
- may declare non-pointer, non-structure, and non-**char** variables as appropriate, but only one pointer, two structures, and one **char**.

- Use military time for both input and output: 23:59:59 is 1 second before midnight, 00:00:00 is midnight, and 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 an empty-body email to the assignment checker with the subject line **C1A7E1_174273_U09845800** and with all three source code files attached.

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

Hints:

Two Common Approaches

Within the `DetermineElapsedTime` function some students prefer to perform computations directly on the hours, minutes, and seconds, whereas others prefer to convert everything to seconds first. Either technique is acceptable but if you choose the latter, be aware that there are `86,400` seconds in one day but the largest value a type **int** expression (such as a variable, a multiplication, an addition, etc.) can portably represent is `32,767`.

Where to store the elapsed time

Declare a **static** `MyTime` structure in `DetermineElapsedTime`, store the elapsed time in it, then return its address. Returning a pointer or reference to an automatic variable is always wrong.

Initializing static Variables

If an initializer is not used in the declaration of a **static** variable, the variable will be automatically initialized to all 0s. If an initializer is used, the variable will be initialized to the specified value(s). Either way, the initialization will occur once and only once during the entire lifetime of the program no matter how many times code flow reaches that declaration. Thus, using any initializer in the declaration of the **static** `MyTime` structure in the `DetermineElapsedTime` function is pointless and misleading.

Producing 2-digit time formats with a leading 0

The `setw` manipulator sets the field width for the next output item and is "not sticky", meaning that it must be re-specified for every output item for which a non-default field width is desired. The `setfill` manipulator sets the fill character to be used if an item does not occupy the entire field and is "sticky", meaning that once set it remains in effect for all fields until explicitly changed. See note 3.7.

Other

1. A common student mistake is to produce a difference of 00:00:00 if both times are equal. However, it must instead be assumed that if the stop time is less than or equal to the start time, the stop time is for the next day.
2. Be sure to use "include guards" (note D.2) in header file **C1A7E1_MyTime.h** and include that header file 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 the structures in elements `lunches[0]` and `lunches[1]`. The first must represent an "apple" weighing 4 ounces and containing 100 calories while the second must represent a "salad" weighing 2 ounces and containing 80 calories. Do not use dynamic allocation for any of these initializers.
2. Loop through each of the non-explicitly initialized elements of the array and do the following in order during each execution of the loop body:
    a. Prompt the user to enter the whitespace-separated name, weight, and calories of a food in that order on the same line. The name must not contain whitespace. To reduce clutter, you may tell the user what must entered before your code enters the loop, then use a simpler prompt like **Enter:** inside to loop each time a user entry is required.
    b. Store the food name into a temporary character buffer you have 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 an empty-body email to the assignment checker with the subject line **C1A7E2_174273_U09845800** and with your source code file attached.

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.

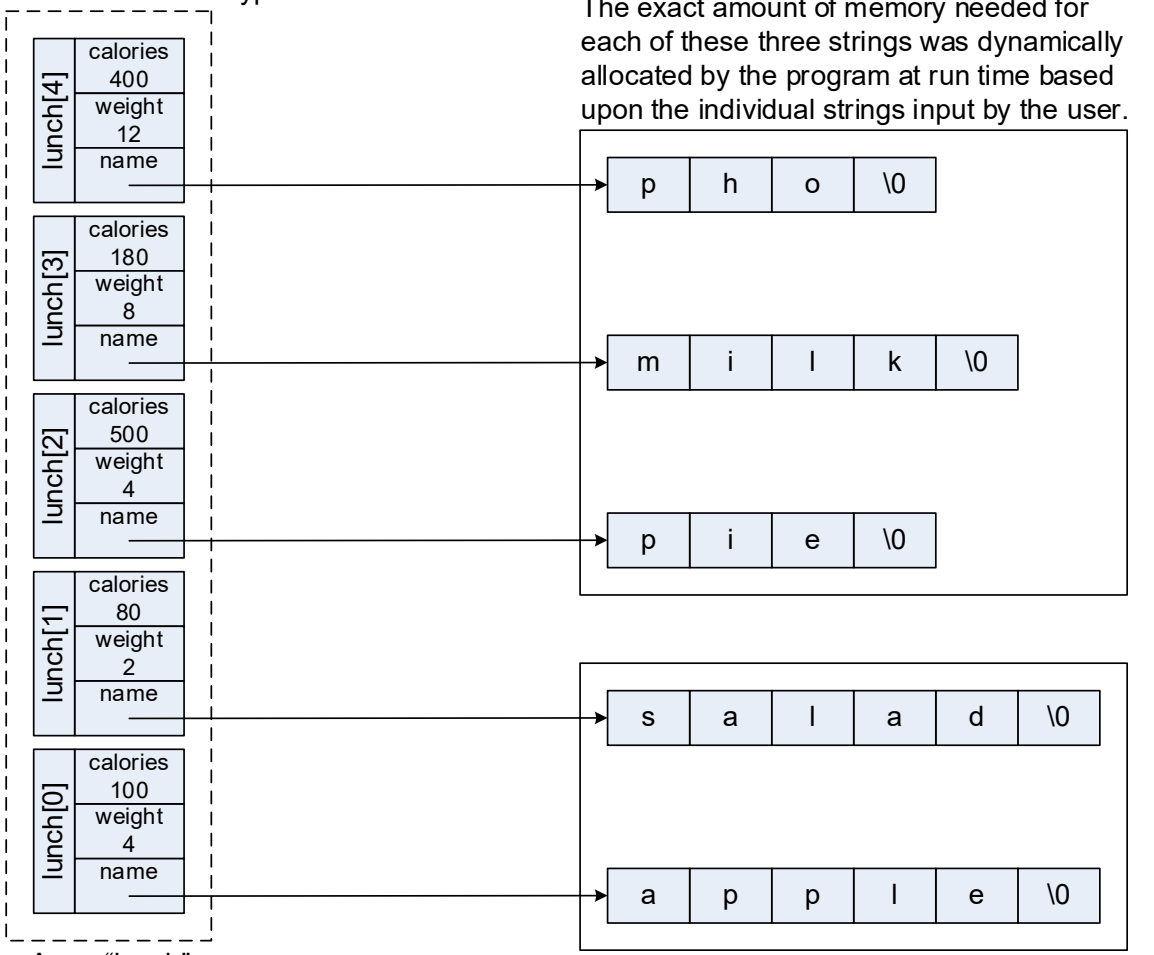
Pointers and Memory Diagrams

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.

Please see the diagram on the next page.

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.