Computer Science 245 Lab #2

Structs, String Objects, & Classes in C++

Due Date : Wednesday, January 23rd, 1:00 PM

30 points

Lab Setup

1. Make a Lab2 directory and change into it from within your CSC245 subdirectory.
2. Copy over your lab files from our class directory by typing :

   ```
   cp /pub/digh/CSC245/Lab2/* .
   ```

   (**Notice that after that asterisk there is a space followed by a period, and then your return key.**)

Structs

1. Edit the file myStruct.cpp. Below the DateType struct declaration include a struct of type EventType with the following two fields :
   (a) A place field of type string
   (b) A date field of type DateType
2. Within your main function, declare a variable Holiday of type DateType. Assign "January" to the Month field, 5 to the Day field, and 2019 to the Year field. You may use one initial assignment upon declaration or you may use three separate assignment statements using the dot operator to assign your values field by field.
3. Declare an array named Dates to hold 100 structs of type EventType. That is, a total of 100 hierarchical records. Use the SIZE constant already declared.
4. Complete the void function named PrintStruct which prints out each of the three fields of a variable of type DateType. The output of this function should be in the following format :

   ```
   January 5, 2019
   ```

5. Complete the void function named AssignMonth which assigns the string "January" to the Month field embedded within each of the records found within the 100 array slots of array Dates. There will be no output from this function.

String Objects

Edit the file `names.cpp`. We want to write a function that, given a string representing someone's name in the form `Last, First MI.` breaks that string down into three substrings representing their first name, middle initial, and last name. Each of these substrings is sent back to the calling function.

For example, if our input string is `"Adams, John Q."`, we want to return the strings `"John"`, `"Q"`, and `"Adams"`.

So, in effect we have a "smart" function here which appears to know exactly what the components which make up a name are. The key to doing this is that every string will be entered in the format above. That is, the last name will always be followed by a comma and a space, the first name, a space, the middle initial, and finally a period.

You will need to make use of the `length`, `find`, and `substr` operations. This function should not be long (like maybe around five lines). Be sure and look up how `substr` differs from the Java version.

Here are some helpful hints :

- The middle initial will always appear one notch back from the position of the period in the string. It will of course always be of length 1.
- The last name will always start at position 0, and its length will always be equal to the index of where the comma is found.
- The first name will always start two notches from where the comma appears. Using the length of the string and the position of the comma, you can come up with an expression representing its length.

Compile and test your program using your own name.

Classes

1. Compile, but do not link, the `Time` class by typing :

   ```
   c++ -c time.cpp
   ```

   This should create a `time.o` object file. Do an `ls` to see. Remember that when you decide to utilize the `Time` class in a client file, you'll link with it by typing :

   ```
   c++ client.cpp time.o
   ```

2. Create a client program in your directory named `testTime.cpp`. Bring in the `Time` class using appropriate `include` statement for `"time.h"`. Never include `"time.cpp"` in the client file.

   Declare an object variable `myTime` of the class `Time`, initialize it to 9:30 AM using a constructor, and print it out by entering :

```
Time myTime(9,30,0);
myTime.Write();
```

Notice how the `Write` operation has a set of opening and closing parentheses after it. These parentheses are required, and indicate that this operation takes no parameters.

3. Compile, link, and execute your client program to make sure everything is running smoothly. The time above should be printed to the screen.

4. Now, let's complete the steps we need to add an additional observer function called `WriteAmPm` to your `Time` class. An observer function is one that has the right to use or inspect the private data, but not actually change it. We normally include a `const` after its prototype.

5. **First off, you must add a prototype for this function within the public section of the specification file time.h**. It should look as follows :

```
void WriteAmPm() const;
```

Don't worry about adding precondition and postcondition comments.

6. Listed next is the function body for `WriteAmPm` that should be inserted into the implementation file. Notice how we include the `class` name `Time` followed by a `::` for all our function bodies in the implementation file. This makes it clear to the programmer that these functions belong to the `Time` class.

```
void Time::WriteAmPm() const
{
  bool am;
  int tempHrs;

  am = (hrs <= 11);
  if (hrs == 0)
    tempHrs = 12;
  else if (hrs >= 13)
    tempHrs = hrs - 12;
  else
    tempHrs = hrs;

  if (tempHrs < 10)
    cout << '0';
  cout << tempHrs << ':';
  if (mins < 10)
    cout << '0';
  cout << mins << ':';
  if (secs < 10)
    cout << '0';
  cout << secs;
  if (am)
    cout << " AM";
  else
    cout << " PM";

  cout << endl;
}
```

7. Re-compile your revised implementation file using the `-c` extension to get your new object file.

8. Add the following statement to your client program.

   ```
   myTime.WriteAmPm();
   ```

9. Re-compile and link your client program to see that it also prints out the time in the new format using your new public function you've added.

10. Add a destructor function to the `Time` class by including the following line in your specification file :

    ```
    ~Time();
    ```

    A destructor function is automatically called whenever an object goes out of scope. That is, the function that uses it comes to an end.

    In your implementation file, include the code on the next page which simply prints out a statement to ensure you that the destructor is automatically called upon completion of your program (when our time object goes out of scope).

    ```
    Time::~Time()
    {
         cout << "Destructor Called" << endl;
    }
    ```

11. Add a void function named `Mealtime` to your `Time` class which will print out "Breakfast" if the time is exactly 8:00 AM, "Lunch" if it is exactly noon, and "Dinner" if it is exactly 7:00 PM (19:00). All you need to do is set up an `if` statement which checks the values of `hrs`, `mins`, and `secs`. This function should be an observer function since it does not modify the private variables. **Make sure you set it up as an observer function in both the specification and implementation files.**

12. Add a statement to your client program to set the time to a meal time using the `Set` function (e.g., `myTime.Set(8,0,0)`). Also, add a statement to print the new time, and then include a statement to test your new `Mealtime` function. Compile and test your new client program. You should have output *exactly* like the following :

    ```
    09:30:00
    09:30:00 AM
    08:00:00 AM
    Breakfast
    Destructor Called.
    ```

13. Declare an array named `Schedules` that contains 10 items of type `Time` in your client program. Yes, you can have an array of objects. After your declaration, initialize the time of each object in the array to 11 AM, and the print it out. Within a `for` loop that runs `10` times, you will need to call the operation `Set` and `WriteAmPM` for each item within the array.
14. Run your revised client program. How many destructor calls did you get? Explain why below.

Finishing Up

Print out `myStruct.cpp`, `names.cpp`, and your client program for your `Time` class. Staple them together to hand in. If you have any questions on finishing up this lab, please let me know. Remember, the only stupid questions are those which are not asked.