COMSC-165 Lab 8
Intro To Object Oriented Programming

In this lab session you will exercise programming with `struct` objects. The purpose of this lab is to familiarize you with the C++ `struct` syntax, and for you to discover ways to solve problems using objects. You will reproduce some coding exactly as it appears in this lab sheet, and modify it as directed in the instructions. You will also get to write your own code, based on specifications provided.

In this lab you will design a `struct`, and use objects to store the time of day. You will write "utility functions" to add hours, minutes, and seconds to "time objects", compute the difference between two times of day, and test whether one time object precedes another or not.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab8** folder provided.

LAB 8: Working With `structs [ Tod.cpp, todA.cpp ]`
Create two console applications named **Tod.cpp** and **todA.cpp**. Do NOT submit Tod.cpp until you complete step 5, and todA.cpp after step 8.

STEP 1 of 8: Define, Declare, Initialize, and Output
Create a console application named **Tod**, with the **Tod.cpp** file as listed below. It uses an object to track the time of day. The code has one *object*, made using a time-of-day `struct`, which it initializes and outputs. Type the code as it appears below, but make changes as needed in order to follow the programming conventions for this course.

```
#include <iostream>
using std::cout;
using std::endl;

// struct definition
struct tod
{
  int hour; // the hour, 0-23
  int minute; // the minute, 0-59
  int second; // the second, 0-59
};

int main()
{
  // declare and initialize the time to noon
  tod theTime = {12};

  // print the time (should say 12:0:0)
  cout << "The time is " << theTime.hour << ':'
    << theTime.minute << ':'
    << theTime.second << endl;
}
```

Note: Do not worry about the format of the outputted time -- in step 5 you will learn how to output it as `12:00:00` instead of `12:0:0`.

Note: Do *not* use global variables. Global *constants* are okay, though..

STEP 2 of 8: Modify Object Element Values
Modify the **Tod.cpp** program so that it includes changing of the object **`theTime`** after it is originally initialized and outputted. Change the time by adding one hour (**`theTime.hour += 1;`**) and 30 minutes (**`theTime.minute += 30;`**). Output the new time on a new line of output, beginning "`Now the time is`".

STEP 3 of 8: Multiple Objects
Modify the **Tod.cpp** program by removing the **`tod`** object named **`theTime`**, and declaring 5 new objects named **`noon`**, **`midnight`**,**`lunchTime`**, **`supperTime`**, and **`bedTime`**. Initialize **`noon`** to 12:00:00 and **`midnight`** to 00:00:00. Initialize the other three objects as you please. Output each object on a separate line of output, with proper identification: e.g., "`noon is 12:0:0`".

STEP 4 of 8: Arrays Of Objects
Modify the **Tod.cpp** program so that the **`tod`** struct has a *new data element*, and the 5 objects of **`tod`** are declared in an array, instead of declaring them separately. Redefine the `struct` to include C-string-based textual description, as follows:

```
// structure definitions
struct tod // updated definition
{
  int hour; // the hour, 0-23
  int minute; // the minute, 0-59
  int second; // the second, 0-59
  char descr[32]; // the description of the time of day <-- new!
};
```

The new array allows there to be a text description of the time, of up to 31 characters. Since the array size is 32, and there needs to be a null terminator for the C string, there can be only 31 usable characters in the array.

Declare the five **`tod`** objects in an array. Initialize the objects with braces in the declaration *or* with assignment statements after the declaration (i.e., statements with equal signs), per your preference. E.g., in **`int main()`** use:

```
tod theTime[] = {{12, 0, 0, "noon"},
  {0, 0, 0, "midnight"},
  {11, 30, 0, "lunch time"},
  {18, 45, 0, "supper time"},
  {23, 59, 59, "bed time"}};
```

or, e.g.:

```
tod theTime[5];
theTime[0].hour = 12;
theTime[0].minute = 0;
theTime[0].second = 0;
strcpy(theTime[0].descr, "noon");
```

`...` *etc. for the other 4 times of day*

Output all five time objects as in step 3, *but* use a **`for`** loop to do so, and use the **`char descr[32]`** data element to identify the time. E.g., say "`lunch time is 12:0:0`" by using, e.g., **`cout << theTime[i].descr << " is " <<`**

`theTime[i].hour << ':' << theTime[i].minute << ":" << theTime[i].second << endl;` in a loop indexed by the integer `i`.

STEP 5 of 8: Sending Objects To Utility Functions
Modify the **Tod.cpp** program by adding a utility function to handle the `cout` statement. This allows formatting logic to be added to the output, without having to code that logic in the `main()` function. Add this function prototype, after your `struct` definition and before `int main()`:

```
// utility function prototype(s)
void coutTod(const tod&);
```

Your function header should look like this:

```
void coutTod(const tod& t)
```

Your function should output the time's description, its hour, and also include this logic for outputting the minute and second:

```
if (t.minute < 10)
  cout << '0';
cout << t.minute << ':';
if (t.second < 10)
  cout << '0';
cout << t.second << endl;
```

Your output should now read, e.g.: `"lunch time is 12:00:00"`, instead of `12:0:0`.

The output logic in `main()` should now be a simple `for` loop that invokes the `coutTod` function.

Note: it is not necessary to pass the `struct` object by reference -- it could be passed by value (i.e., without the `&` symbol). But it is done by reference here because it takes less stack memory to pass a reference to the memory address of the `struct` object instead of a whole new copy of the object.

*Post **Tod.cpp** to the COMSC server.*

STEP 6 of 8: Sending An Array of Objects To A Function
Create a new console application named **todA.cpp**, created by copying **Tod.cpp** from step 5, and moving the `for` loop that calls `coutTod()` in `main()` to a *new* utility function. This new function is to have the following prototype, which *overloads* the existing function by the same name:

```
void coutTod(int, const tod*);
```
// n = number of items in the array

You may use `const tod*` for the second parameter, as you learned in Comsc-110, if you prefer -- they are interchangeable. Replace the entire output coding in `main()` (e.g., the `for` loop) with the single statement `coutTod(5, theTime);`

In the new `void coutTod(int n, const tod* t)` function, put a `for` loop that invokes the original `void coutTod(const tod& t)`. *You should have two functions named **coutTod** -- do not delete the original one that you wrote in step 5!*

STEP 7 of 8: A Utility Function For The `tod` Struct
Modify **todA.cpp** by adding a function with the following prototype, to calculate the difference in seconds between two times of day:

```
int diffTod(const tod&, const tod&);
```

The function should compute and return the number of seconds between the two time objects in the parameter list. Return a *negative* number if the second time is *after* the first.

Add *three output lines* to the main function, after the 5 times of day are outputted. Each of the new output lines should show the difference in seconds between any two time objects of your choice. E.g., these lines should read something like this: `"from lunch time to noon is -1800 seconds"`, using something like this: **`cout << "from " << theTime[2].descr << " to " << theTime[0].descr << " is " << diffTod(theTime[2], theTime[0]) << endl;`**.

STEP 8 of 8: More Utility Functions
Modify **todA.cpp** to *add* two more functions with the following prototypes:

`int isAfter(const tod&, const tod&);` // true if 1st is after 2nd
`int isBefore(const tod&, const tod&);` // true if 1st is before 2nd

Each is to compare two times of day, and return 1 if true and 0 if false. Each new function should use **`diffTod()`** to compare the two times that are passed to them in the argument list.

Add *three output lines* to the main function, after the 5 times of day and the 3 differences are outputted. Each of the new output lines should use one of the two new functions, and say whether one time object precedes another -- you choose which times to compare, and which function to use. E.g., these lines should read something like this: `"lunch time is before noon"`, using something like this:

```
if (isBefore(theTime[2], theTime[0]))
  cout << theTime[2].descr << " is before " << theTime[0].descr << endl;
else
  cout << theTime[2].descr << " is not before " << theTime[0].descr << endl;
```

*Post **todA.cpp** to the COMSC server.*

GRADING POLICY
Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.
9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.

13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.