COMSC-165 Lab 9
Intro to C Pointers

In this lab session you will program with *pointers*. You will practice using pointers as an alternate syntax for passing by reference, and for building indexes on a database of objects.

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 **lab9** folder provided.

LAB 9a: Getting To Know Pointers `[ pPrimer.cpp ]`
Create a *console application* named **pPrimer.cpp** by modifying **Tod.cpp** from Lab 8. In the **`main()`**function, remove all output statements, and remove values assigned to the tod's -- that is, declare the tod array, but let the values be uninitialized, "unpredictable" values. Then add **`cout`** statements to output the *memory location addresses* of *every* variable and object that appears in **`main()`**. Here's how:

1. Include **`cout << "theTime: " << &theTime << endl;`** to output the location in memory of the *array* of **`tod`** objects. This will output a memory address, of the form **`0xABCD1234`** or **`ABCD1234`**, without the **`0x`** prefix (depending on your compiler and operating system).

2. Include **`cout << "theTime[0]: " << &theTime[0] << endl;`** for the location of the zeroth object in the array, and similar statements for *all the other objects* in the array.

3. Include **`cout << "theTime[0].hour: " << &theTime[0].hour << endl;`** for the location of the *hour* element in the zeroth object of the array, and similar statements for *all the other objects* in the array. Also include the same for the *minute* and *second*. Include the *descr* element of the `struct`, too (e.g., **`&theTime[0].descr`**).

4. Include the addresses of any **`int`** objects you may have in the code. E.g.:
```
cout << "int i: " << &i << endl;
```

5. Output the size, in bytes, of the **`tod`** `struct`, and of pointers to the `tod` objects.
```
cout << "Size of tod = "
  << sizeof(tod) << endl;
cout << "Size of tod pointers = "
  << sizeof(tod*) << endl;
```

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

LAB 9b: Pointer Syntax `[ pFun.cpp ]`
Create a *console application* named **pFun.cpp**, which is to be created as a modification of **todA.cpp** from Lab 8. Change all of the function prototypes and headers to use pointer syntax instead of reference syntax. That includes all reference variables and arrays that appear in the parameters lists.

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

LAB 9c: Indexes For An Array `[ pIndex.cpp ]`
Create a *console application* named **pIndex.cpp**, which can be created from a copy of **Tod.cpp** from Lab 8. In the `main()` function, make the following additions, so that the `tod` objects appear in both alphabetical order and in chronological order in the screen output:

1. Add an "index" as an array of pointers, `tod* tIndex[5];`

2. Set the `tIndex` values to the addresses of the 5 `tod` objects, in *chronological* (i.e., time-based) order. E.g., `tIndex[0] = &theTime[1]; // midnight`, etc. Remember that midnight comes at the *start* of the day.

3. Add another "index", `tod* aIndex[5];`, to store the records in alphabetical order.

4. Set the `aIndex` values to the addresses of the 5 `tod` objects, in *alphabetical* order. E.g., `aIndex[0] = &theTime[4]; // bedtime`, etc.

5. Add two `for` loops at the end of `main()`, to output out the `tod` objects in order of the two indexes -- one loop for each index. Use `coutTod()` to output the objects. Label each of the two lists: e.g., `cout << "Chronological Order" << endl;` and then `cout << "Alphabetical Order" << endl;`

Keep the original output, which outputs the `tod` objects in their order of appearance in the array. Label the output as *Original Order*. If you use the original `coutTod(tod& t)` function from Lab 8, remember to use `coutTod(*aIndex[i]);` to refer to it in `main()`. But you may elect to rewrite the function to accept pointers (e.g., `coutTod(tod* t)`) so that you can use `coutTod(aIndex[i]);` to refer to it in `main()`, instead.

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

BONUS EXERCISE: Clever Uses Of Pointers
Create a *console application* named **Clever.cpp** by modifying **Strings.cpp** from Lab 7. Use the "clever" code presented in the lecture on pointers to perform the six functions' manipulations (i.e., use pointer incrementing:`*p++`).

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.