COMSC-165 Lab 0
Programming Conventions

A "programming convention" is a made-up standard for how to write programs, when working with a group of other people. These standards help to guide the way programming decisions get made, so that all programs are consistent in organization and appearance, making them easier for others to follow. In this Comsc-200 course, your instructor has certain standards which you are expected to follow. The purpose of this assignment is to practice and apply those standards.

To do so, you'll take a program previously written by you, and make any changes that are required in order to meet the standards for this course. Refer to the "free programming resources" page on the Computer Science department's section of the DVC website, under "Programming how to's" for detailed steps in compiling C++ code in Windows, Mac OSX, or Linux/UNIX. It does not matter which compiler you use, because the CPPs that you develop are expected to work on any system with any C++ compiler.

This assignment is worth ZERO points. But it must be completed fully correctly before your first lab will be considered for scoring.

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 0-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 **lab0** folder provided.

LAB 0a: Who Are You? [`AboutMe.txt`]
Just so your instructor knows a bit about your programming background, complete this non-programming exercise. With a code editor of your choosing, write a text file named **AboutMe.txt**. In this file, in any format you choose, include the following:

1. Your name (like "Robert Burns", and what you prefer to be called (like "Professor Burns").
2. Prerequisites: When did you take COMSC 110? Who was your instructor? If you did not take COMSC 110, explain what equivalent course or preparation enabled you to fulfill the prerequisite requirement.
3. Compiler(s): What compiler(s) do you expect to use in this course?
4. Editor(s): What code editor(s) do you expect to use in this course?

*Post **AboutMe.txt** to the COMSC server.*

LAB 0b: The "Hello, World" Program [`HelloWorld.cpp`]
Take a C++ program you wrote previously and rename it as **HelloWorld.cpp**. It can be from any source -- a previous course, like Comsc-110, or a project of your own, or just about anything else. Here are some guidelines for choosing your program:

1. It must be 200 lines or fewer.
2. It must involve console input of at least one `int` or `double` value.
3. It must NOT involve more than one source file (i.e., no "projects").
4. It must NOT use input files.

Modify your program as needed in order to adhere to these programming conventions, which are to be applied to all of your Comsc-200 work:

PROGRAMMING CONVENTIONS
Filenames. Spell and case this and ALL filenames in this COMSC course exactly as specified in the lab writeup.

Signing your work. Identify the lab, yourself, and your programming environment in TWO ways in each CPP you write. Do so once in // comments at the top of your CPP file, per the example below, and do so again in `cout` statements in your program.

Console input. Do not read `int`s or `double`s directly from `cin`. Use a string buffer, and convert with `atoi` or `atof`.

stdafx.h and pragma. Do NOT use either of these Microsoft-specific statements in ANY of programs for this COMSC course. If you are using an IDE, and it's adding them for you, and you can't stop it, then maybe you should not be using that IDE...

#include's. Use the proper library includes for all functions or other resources that you get from any C or C++ libraries. Do NOT #include anything that you do not need or use.

iostream vs iostream.h. Do NOT use the pre-standard C++ iostream.h in ANY of your work in this COMSC course. If you have a .h in a C or C++ library name, you are using the wrong version of the library.

cmath vs math.h. The C language libraries ending in ".h" have been replaced in standard C++. The new versions of these libraries drop the ".h" and prepend a "c". When you write a CPP (or H) file for this COMSC course, ALWAYS use these new versions of C-libraries -- that is, the ones starting with the letter "c" and not ending in ".h". Do NOT use `using std::` statements for anything from a C-library.

using namespace std;. Do NOT use `using namespace std;` in any of your work in this COMSC course. Use `using std::` statements instead, but ONLY for items from C++ libraries and NEVER for items from C libraries. Place the `using std::` statements immediately BELOW the `#include` statement for the library to which they pertain, so that it's clear that YOU know what libraries things come from.

using std vs std::. While it is possible to leave out `using std::cout;` and similar statements, and write (for example) `std::cout` instead of just `cout` wherever it appears in your code, we are NOT doing that in this COMSC course. Always use `using std::` statements, and place them directly below the `#include` statement for the library to which they pertain.

void main. Do NOT use `void main()` because that is not standard C++.

return 0. Do NOT use `return 0;` as the last statement in `int main()` in ANY of your programs for this COMSC course. It's not required by the C++ language.

endl vs \n. You may print `\n` to do the same thing that `endl` does. The difference is that `endl` also "flushes the output buffer", while `\n` does not. It's okay to use `\n` in your work in this COMSC course, in place of `endl`. Make sure that the *last-executed* output statement in ALL your programs ends with `\n` or endl.

Alignment and indenting. Use 2-space indenting -- do NOT use TABs, unless you set your code editor to insert spaces for TABs. Align your code so that statements with exactly the same scope as each other appear with the same level of indent.

Code blocks. Write your code in "code blocks". Include a // comment heading for each code block to explain what it does. Separate your code blocks with single skipped lines, so that it is easy to see where one code block ends and the next begins.

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

Follow this sample:

```
// Lab 0, Programming Conventions, part b
// Programmer: YOUR NAME HERE
// Editor(s) used: XP Notepad
// Compiler(s) used: VC++ 2010 Express

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

int main()
{
  // print my name and this assignment's title
  cout << "Lab 0, Programming Conventions, part b\n";
  cout << "Programmer: YOUR NAME HERE\n";
  cout << "Editor(s) used: XP Notepad\n";
  cout << "Compiler(s) used: VC++ 2010 Express\n";
  cout << "File: " << __FILE__ << endl;
  cout << "Complied: " << __DATE__ << " at " << __TIME__ << endl << endl;

  // a code block
  ...
}
```

*Post **HelloWorld.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.