COMSC-165 Lab 3
Advanced C/C++ Control Structures

In this lab session, you will practice practical control structures. If you can store variables (by declaring `ints` and `doubles`), perform computations (using operators and built-in math functions), control the order in which statements are executed (using `if` to bypass and `while` to reverse), and perform console I/O (with `cin` and `cout`), then you basically have all the logical and calculational tools you need to solve almost any problem. While the additional use of functions, arrays, structs, data structures, recursion, and object-oriented programming makes it easier to write some solutions, they do not really provide any more basic capabilities than you have already.

Lab 3 involves a problem from an annual IBM-sponsored ACM (Association for Computing Machinery) Pacific North West Regional Programming Contest, a competition among 2- and 4- year colleges from Alaska to Nevada to Hawaii. The solution is not as easy as it might seem. It involves the completion of several complicated steps, and then assembling those steps into a solution. As a programmer, you need to develop the skill to attack a large problem by breaking it down into smaller ones that you can solve. In order to guide you through such a process, this lab assignment is presented in multiple steps.

There are 5 separate steps. In the first, you will create a program named **Change.cpp**. In the remaining steps, you will add to **Change.cpp**, so that it develops into a complete solution.

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 **lab3** folder provided.

LAB 3: Change Calculator `[ Change.cpp ]`
Modern grocery stores now often have a "U-Scan" checkout lane, allowing the customer to scan and check out their own groceries, without the need of a human checker. These lanes require that change be provided automatically, after the customer enters his/her cash. You are to write a program that computes the bills and coins to be dispensed, minimizing the total number of bills and coins. (That is, for change totaling $5.50, you should not dispense 5 ones and 50 pennies, but a $5 bill and a 50-cent piece instead.)

The bills and coins available for you to dispense are as follows: $50 bill, $20 bill, $10 bill, $5 bill, $1 bill, 50-cent coin, 25-cent coin, 10-cent coin, 5-cent coin, 1-cent coin.

The console-based program, which can easily be converted to an internet CGI program at some future date, prompts the user to input 2 numbers. The first number is the *amount of the purchase*, and the second one is the *amount tendered* by the customer. You may assume that the amount tendered is greater than or equal to the amount of purchase. The console output will be a series of lines showing the *amount of change returned* and detailing the *number of bills and coins* that will be dispensed as change, in descending order of monetary amount, one unit per line. If a bill/coin is not needed in the change returned, no output is produced for that bill/coin. (In other words, do not display "0 $1 bills".)

Plural logic. Proper use of plurals is required, as shown in the sample below. This *will require* some if-else logic to decide whether or not to append an "s" to the end of a denomination name.

Here the sample -- for a purchase of `42.15`, the amount tendered is `50`. There are no `$`'s or commas in the input -- just positive real numbers that may or may not contain a decimal. Here is the output:

```
$7.85
1 $5 bill
2 $1 bills
1 50-cent coin
1 25-cent coin
1 10-cent coin
```

The program terminates after the output is produced.

STEP 1 of 5: Getting Started
Write a version of **Change.cpp** that has a `main` function, but which does nothing, and just has its curly brace container. You may include libraries that you expect to use, either in this step, or as you need them in later steps, per your choice.

Verify that you can compile and run the program.

STEP 2 of 5: Collecting Input
Add statements to `main` to prompt the user for the two numbers -- purchase amount and amount tendered. Include variables for collecting and storing these values. Allow the user to enter both amounts on the same input line, space-separated -- unlike lab 1's TheBasics.cpp. To do so, simply leave out the `cin.ignore(1000, 10);` statements. Verify that you can compile and run the program.

STEP 3 of 5: Producing Output
Calculate the amount of change returned, and output it with the proper formatting. Show the amount with two decimal places, representing cents, and be sure to account for floating point round-off errors. For example, if the purchase is `1.02`, and the amount tendered is `1.10`, then the change should be `0.08` -- not `0.079999999` and not `0.080000000`, and certainly not `0.07`.

Here's code that causes cout to show numbers rounded to the nearest hundredth:

```
#include <iostream>
using std::ios;

#include <iomanip>
using std::setprecision;

int main()
{
  cout.setf(ios::fixed|ios::showpoint);
  cout << setprecision(2);
  ...
```

STEP 4 of 5: Solving The Problem
Write the C statements needed to determine the numbers of bills and coins to dispense in order to make the correct change. There is no trick logic in the US monetary system -- start with the highest denomination, and use as many of that denomination as you can before going to the next lower denomination to make change for the remaining amount.

Do not worry about formatting of the output at this point -- focus on getting exact results.

STEP 5 of 5: Final Formatting
Complete the formatting of the results as per the problem statement. Compile and run.

Include identifying comments at the top of the CPP file, as you did in lab 1. *Be sure to include such identification at the top of every source file that you write for this class throughout the semester.*

*Post **Change.cpp** to the COMSC server for credit.*

HINTS:
This problem is not as easy as it seems. Be aware of the effects of round-off errors, and remember not to test floating point values for equality. Remember that with floating point values and computers, 4 minus 2 can often result in 1.9999999999999999 instead of 2, and that is not greater or equal to 2!

There are at least two ways around this problem -- you can convert decimal dollars to integer pennies at the start, or you can check for `>=1.999` instead of `>=2`.

Cashiers solve this problem every day, without using division and dealing with remainders. So your program should be able to solve it too, without divide or modulus. Think about it. Be sure to write an algorithm representing your logic to guide your solution.

Another problem is that non-numeric input can cause the program to to haywire! If you declare a `double` variable, and use `cin` to read it, but someone enters `$50`, the program chokes. But if you apply what you should have learned in lab 1, this should not be a problem. Non-numeric inputs should default to zero, as in lab 1.

"Never assume anything!". Test your program with various input values. Try to break your own program before you give someone else the opportunity to do so.

Here's a program you can use to find pairs of inputs that will test your program. It generates values that "invite" round-off errors:

```
#include <algorithm>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>

double getNumber(char c[])
{
  int a = rand() % 1000; // dollars
  int b = rand() % 100; // cents
  sprintf(c, "%d.%02d", a, b); // in cstdio
  return atof(c);
}

int main()
{
  // initialize the random number generator
  srand(time(0)); rand(); // in cstdlib and ctime

  // get ten pairs of inputs that challenge the change program's logic
  char a[100], b[100]; // buffers for numeric inputs, each with 2 decimal digits
  for (int i = 0, count = 1; i < 10; count++)
  {
    // get test values to use for inputs
    double d1 = getNumber(a); // purchase amount
    double d2 = getNumber(b); // tendered amount
    if (d2 < d1) {std::swap(a, b); std::swap(d1, d2);} // assure that tendered is not less than purchase

    // get amounts as #of cents
    int i1 = atoi(a) * 100 + atoi(a + strlen(a) - 2); // in cstdlib and cstring
    int i2 = atoi(b) * 100 + atoi(b + strlen(b) - 2); // in cstdlib and cstring

    // compute change in cents, two different ways
    double d = (d1 - d2) * 100.0; // calculation, subject to round-off error
    int n = i1 - i2; // calculation, not subject to round-off error

    // if the results don't match, these inputs challenge the change program
    if (n != (int)d)
    {
      std::cout << "Pair #" << count++ << ":\n"; // not every pair makes it this far
      std::cout << "Use for purchase amount: " << a << std::endl; // in iostream
      std::cout << "Use for tendered amount: " << b << std::endl << std::endl; // in iostream
      i++;
    }
  }
}
```

Rewrite the above for **float** instead of **double** to get another set that you should test. Please do NOT ask me to explain how this works! It would take too many hours to do so, and it's way beyond the scope of this course.

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.