CMPT 115 - ASSIGNMENT # 1

Summer 2016

Assignment marks: 40

Assignment 1 Exercises (40 pts)

This is a programming assignment. Get started early! Unanticipated problems will arise in your work, and you will need time to sort them out.

Clarifying questions can be posted on the CMPT 115 Moodle forums. Help with debugging can be obtained by working in the lab during Help Desk hours of your instructors and TAs. Expect high demand for this kind of help on Fridays.

All C++ source code files must contain your name, student number, and NSID; and the course, assignment number, and question number as comments at the top of the file. All C++ programs must compile with g++ on the command line without errors or warnings, using the standard flags described in tutorial, namely -Wall -pedantic.

You may, of course, work on your assignment using Eclipse, or any other C++ IDE, on your personal notebooks or desktops, or on any computer in the Spinks labs. We cannot always help you with problems arising specifically from the IDE, but C++ should have the same bahaviour on all these systems. There are good reasons to use an IDE like Eclipse or MSVS, but there are good reasons to learn to use the command-line too. Learn both!

Use of String class or string or other object-oriented code (except for cin or cout and any others explicitly allowed in an assignment specification) will result in a flat deduction of 25%. (That means: don't use advanced techniques to avoid learning the concepts we are attempting to teach you!)

Background

A Magic Square is an arrangement of numbers in square, so that every row, column, and diagonal add up to the same value. Below are two squares, but only one of them is a Magic Square.

8	1	6
3	5	7
4	9	2

1	9	6
5	3	7
4	8	2

The square on the left is a 3 x 3 magic square, whose rows, columns, and diagonals all sum to 15. On the right, is a 3 x 3 square of numbers whose rows columns and diagonals don't have the same sum.

There are magic squares of all sizes, but we'll be concerned with 3x3 squares, and checking if a given arrangement of 9 numbers is a magic square or not.

Definition: A 3 x 3 magic square is formally defined by the following three criteria:

- It contains the integers 1 through 9 inclusively.
- Every integer in the range 1 through 9 appears exactly once.
- Every row, column, and diagonal sums to 15.

Task

In this assignment you will implement a program that does the following:

- It asks the user for a sequence of 9 numbers. The order of the numbers is important, as the rows of the grid use this order.
- It checks whether the sequence of numbers is a magic square or not. Your program should display the message “Yes!” if it’s magic, or “No!” if it’s not.

It’s very important to point out that you are not being asked to construct a magic square; only to check if a square is magic or not.

Exercise 1: Implementing a program starting with a given design (36 of 40 pts)

We’ve given you a design document called MagicSquareDD.txt (available on Moodle), which is the result of a fairly careful design process. It describes a collection of algorithms which, when used together, should solve the problem. You must implement the program according to the design in this document. There will be some very small decisions you still have to make about the implementation. Every “Algorithm” in the design document will be a procedure or function written in C++. Function names should be very strongly similar to the given design, if not identical.

It’s a top-down design, and algorithms are presented in the order they were designed. Sometimes a function is called before the algorithm is given; this is the opposite of the order that C++ uses. The order is not important for a human, though C++ has a preference because of its age.

Because everyone is starting with the same design document, every program will have a high degree of similarity. The value of this assignment is the experience you gain from doing it. Hopefully, you will agree that a good design makes implementation fairly easy. Hopefully, you will see the earmarks of a thorough (if not high quality) design.

Here’s a bottom-up guideline for completing the work.

1. Start by downloading and reading the design document carefully. We’ve done our best to make things as clear as we can, but there may be some details that are still unclear. This is normal and natural, and your job is to ask.
2. Create a basic C++ “Hello World” program. You could start with the source.cc program used in the Tutorial material, too.
3. Start with Algorithm 2. Copy the design specification into your C++ file and replace the pseudo-code with C++ code. You should leave the Algorithm header (pre, post, and return) in your C++ file

as documentation. You'll observe that it's already nicely formatted as a C++ comment! An experienced reader of your code should be able to look at the documentation, and your implementation, and verify that your code meets the design correctly. Implement the function as designed, and write test code in main() to be sure you've got it right. Do not go on to the next part until you are convinced your function is correct. When you're ready to move on, leave the testing code in main() without change. Your function should continue to display correct testing output as you work on other parts of the design.

4. Implement Algorithms 1.2.2–1.2.4 one at a time (copy the design, replace the pseduo-code, and keep the header). These functions are all pretty similar, so implement one of them first, and test it thoroughly. When you're convinced it's correct, the other two should be fairly easy. But test all three equally thoroughly in main(). Do not go on to the next part until you are convinced your functions are correct. When you're ready to move on, leave the testing code in main() without change. Your functions should continue to display correct testing output as you work on other parts of the design.

   Hint: Use the boolean operator AND && in these functions. Your function should be pretty simple and short.

5. Implement Algorithm 1.2.1. Note that the design mentions a known bug. Here's a chance to think about robustness. Write test code in main() to be sure you've got it right. Do not go on to the next part until you are convinced your function is correct. When you're ready to move on, leave the testing code in main() without change. Your function should continue to display correct testing output as you work on other parts of the design.
6. Implement Algorithm 1.2. It's basically a function that calls the functions you've already implemented, and have already tested. Test it with apprpriate arrays that are declared and initialized in main(). Use squares that are magic, and squares that are not magic. Try to break your own code with your tests. Try squares with all 5s. Try squares with negative values. Try squares with very large values. Don't change your test squares; create a new square for each test. Do not go on to the next part until you are convinced your function is correct. When you're ready to move on, leave the testing code in main() without change. Your function should continue to display correct testing output as you work on other parts of the design.

   Hint: Use the boolean operator AND && in these functions. Your function should be pretty simple and short.

7. Implement Algorithm 1.1. This is a very simple function, and depends on console input, so testing is a bit more cumbersome. Try to be thorough. Do not go on to the next part until you are convinced your function is correct. In this case, and in the case of functions that are basically designed to manage console input, you can remove the testing code from main(), because it really would hinder your progress to leave it in.
8. Implement Algorithm 1. You can put the code for this algorithm in main(), after all your testing code for all the above functions. Test thoroughly.

9. When you are convinced your program is correct, hand it in, with all the testing code still present and active in main(). Use comments to indicate what's being tested.

What to hand in:

- Your program in a file called a1.cpp (or .cc if you prefer). It should contain all the functions mentioned in the design document, and all the algorithm headers given in the design document. It should also contain all the testing code you wrote in main().
- A text-only file called a1testing.txt. Run your finished program (or as far as you got), and copy paste the console output into the text document.

Your name, student ID, NSID, and lecture section must appear in every file you hand in for grading. This is for the markers' benefit, and really helps them. You must submit documents named according to our specification, because it also helps the markers. Please help the markers out as much as possible!

Grading (36 marks)

- 24 marks for the implementation, as follows: a total of 3 marks each for 8 algorithms named in the design document: 2 marks for each correct function, and 1 mark for including the algorithm header for the function as a C++ comment.
- 12 marks for testing. 2 marks each for testing 6 algorithms (you do not have to show testing for Algorithm 1.1 getSquare(), or for Algorithm 1 main()). For each algorithm, good testing will get 2 marks; poor testing will get 1 mark, and no testing will get 0 marks.

Exercise 2: Reflection (4 of 40 pts)

Answer the following questions about your experience implementing the program. You may use point form, and informal language.

1. (1 mark) How confident are you that your program (or the functions that you completed) are correct? What new information (in addition to your current level of testing) would raise your confidence? What new information (in addition to your current level of testing) would lower your confidence?
2. (1 mark) The design for Algorithm 1.2.1 mentions a "known bug". How did you address this particular bug, i.e., what did you do to make your implementation of Algorithm 1.2.1 more robust?
3. (1 mark) Algorithm 1.2.1 is probably the only algorithm in the design that could have more than one design. We used a boolean array, but that wasn't the only way to solve the problem. How hard would it be to replace Algorithm 1.2.1 with a different algorithm (ignore the difficulty of designing the new algorithm; assume someone else gives a new design to you, and your job is simply to use it)? What other parts of the whole program would have to change?

4. (1 mark) How much time did you spend completing this assignment? If you had to solve this problem without the given design document, how much longer do you think it would have taken you to complete, start to finish? Would you have been as confident in your program's correctness?

Grading (4 marks)

- 1 mark per questions.
- Your answer to each question above demonstrated thoughtful reflection. There are no right answers.

What to hand in:

- Your responses to the questions above in a file called a1questions (PDF, DOC, DOCX, RTF, TXT documents are all acceptable).

Your name, student ID, NSID, and lecture section must appear in every file you hand in for grading. This is for the markers' benefit, and really helps them. You must submit documents named according to our specification, because it also helps the markers. Please help the markers out as much as possible!