CMPT 115 – ASSIGNMENT # 2

Summer 2016

Assignment marks: 29

Part 1 Programming Section

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!)

Exercise 1: Implement arrangeThree Function using References (9 pts)

In the lecture, we saw a function called swap(), that swaps the values of 2 reference parameters.

1. Write a function called arrangeThree that swaps the values of three integers, given as three separate parameters. This function should not have any return value, but, like swap() in the lecture notes, moves the values from one place to another. When the algorithm returns, the

smallest value should be in the first parameter, and the largest is in the last (according to the order of your parameter list).

For example, suppose you have three references whose values are in the following order: 3,2,5. When your function finishes, the references will have values in increasing order: 2,3,5. The following code might help explain it:

```
// in main() somewhere
int a = 3;
int b = 2;
int c = 5;
cout << a << ' ' << b << ' ' << c << endl;  // displays 3 2 5
arrangeThree(&a, &b, &c);                   // organize!
cout << a << ' ' << b << ' ' << c << endl;  // displays 2 3 5
```

You may implement other functions to be used to solve this problem. Hint: You can use swap() to help here. A variant of swap() that swaps conditionally might also be helpful.

Note that your implementation should include an algorithm header (pre, post and return) and the function body. If you use other procedures or functions, they should all have complete headers and bodies.

Test this function and any others that you created thoroughly in main().

Note: Do not solve this problem using arrays. That's not what you're here to learn. Practice using references and pointers! Otherwise, this is just a useless exercise.

Your main function only needs to contain testing for these two functions, and a couple (2) of test cases (like the above code). Every function that you write must be tested thoroughly! The testing output must be readable by a third party. As in A1, you should implement the solution one function at a time, and test each function separately. Do not go on to the next function until you've tested the current one thoroughly. Testing must remain active while you are programming other functions, because you may create bugs that break functions you thought were working!

What to hand in:

- Hand in your C++ program in a file called a2q1.cc or a2q1.cpp.
- A text-document called a2q2testing.txt that shows your program working on a few examples, along with all the testing you did along the way. You may copy/paste from the console.

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

Grading should be based on correct use of references and pointers. Solutions that evade the harder bits of logic by using arrays (with or without using array-based sorting functions) should receive no marks for the implementation (but would get full marks for algorithm headers).

- 3 marks: Your implementation of arrangeThree has a complete algorithm header describing pre/post/return requirements, as a comment in C++.
- 3 marks: Your implementation of arrangeThree is correct. Marks will be deducted for:
  - incorrect use of references, pointers, and operators (1 mark each)
  - other obvious errors (1 mark each)
- 3 marks: Your main function has active testing code that tests your functions thoroughly. No marks will be given if the test code is commented out, or deleted.

Part 2 Design Task

In this part, you're asked to create design documents for a number of short problems, that are all somewhat related. You should approach your design in a top-down manner. Your top-level design should call several sub-algorithms to do specific parts of the whole task. Each algorithm should have a full algorithm header, including the name of the function, pre-, post-, and return descriptions.

You should approach this part as an attempt to practice the design of software. Even if you think you can solve the problem all in one algorithm, you should break it down into a few (at least) smaller sub-algorithms. It is important to practice the skill of dividing tasks into functions like this.

In your work, consider the C.E.R.A.R. goals. You don't have to apply all of them, but your should be thinking about them, and applying them when it seems appropriate.

You will be graded on whether or not your design represents an attempt to employ top-down design, and to adhere to the standards for pseudocode that we have established in lecture. You should take the design document from Assignment 1 as a model to follow. Each design exercise will be shorter than A1's design document.

Your answers to Part 2 should be submitted in separate documents for each exercise (file formats such as PDF, DOC, DOCX, RTF, TXT are acceptable). A document that can't be opened by the markers will not be graded.

Background

In this assignment you will employ top-down design to develop algorithms dealing with Latin Squares. A Latin Square is an array of NxN integers. Each row and each column contains all the numbers 1-N (inclusive).

For example:

1	2	3
2	3	1
3	1	2

is a 3x3 Latin Square. It's a boring one, because the rows and columns are obtained by a pattern of rotations.

Latin Squares are used in many fields, and are very useful in scientific experimental design, but only if they are randomized. To create a randomized 5x5 Latin Square, start with a boring one, like the above. Then repeatedly "swap" two random rows, and then two random columns. If you do 50 or so random row and column swaps, the square looks pretty random. For example, we've swapped the first and second row of the boring 3x3 LS above:

2	3	1
1	2	3
3	1	2

After the swap, the grid is marginally less boring. You should swap pairs of rows, and pairs of columns equally often.

Exercise 2 The Latin Square Detection Problem (10 pts)

Design an algorithm to determine if a given 5x5 array is a Latin Square. You don't need to generalize to NxN squares; 5x5 is fine for now. You may use algorithms from A1 for this, but you have to include them in your design document.

What to hand in

Submit your design document in a document named a2q2 (file formats such as PDF, DOC, DOCX, RTF, TXT are acceptable). A file that can't be opened by the markers will not be graded.

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

- 4 marks: your design shows evidence of top-down design, being composed of several algorithms that do specific simpler tasks.
- 2 marks: every algorithm in your design is expressed in pseudocode
- 4 marks: every algorithm has a complete algorithm header, including pre, post and return.

Exercise 3 The Latin Square Randomization Problem (10 pts)

Design an algorithm to randomize a boring 5x5 Latin Square (see above for an example of a boring Latin Square). Use your algorithm from part 1 to verify that the result of your randomization is indeed a Latin Square. (This is not the best way to randomize a Latin Square, but it's the simplest.)

In pseudo-code, you can use the phrase "generate a random number in the range 0..4" when you need it. You don't need to say more than that about random numbers.

What to hand in

Submit your design document in a document named a2q3 (file formats such as PDF, DOC, DOCX, RTF, TXT are acceptable). A file that can't be opened by the markers will not be graded.

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

- 4 marks: your design shows evidence of top-down design, being composed of several algorithms that do specific simpler tasks.
- 2 marks: every algorithm in your design is expressed in pseudocode
- 4 marks: every algorithm has a complete algorithm header, including pre, post and return.