Lab 1 – Introduction to C++

Overview

This assignment's purpose is to get you a bit more familiar with C++ and start digging into its basic constructs. You are going to take several Java files and convert them into a single C++ program.

Description

The three files you are going to work from are as follows:

- QuadraticRootTool.java
- GradeCalculator.java
- CreditCardValidator.java

You are going to combine the functionality of these three files into a single program, with a basic menu prompt. Typically in C++, header files contain the prototypes of functions, while .cpp files contain the definitions. You are going to follow that convention, writing the code for this assignment across 3 files:

- main.cpp
- Functions.cpp
- Functions.h

The first thing your program will need is some sort of prompt for the person using the application (aka "the user"). Not all programs need this human-centered element, it is good practice for you to do so (i.e. it's a requirement in this assignment). Nothing too complicated, just a simple menu, like this:

```
1. Quadratic Root
2. Grade Calculator
3. Credit Card Validator
Enter a number:
```

After getting a number (and verifying that it not only IS a number, but is within the valid range), you will call the appropriate function:

```
QuadraticRoot();
GradeCalculator();
CreditCardValidator();
```

The details of what each function should do are in the next sections.

Quadratic Root Tool

The roots of a quadratic equation in form of $\boldsymbol{ax^2 + bx + c = 0}$ can be obtained using these formulae:

$$\boldsymbol{r1} = \frac{-b+\sqrt{b^2-4ac}}{2a} \text{ and } \boldsymbol{r2} = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

$\boldsymbol{b^2 - 4ac}$ is called the discriminant of the quadratic equation. If it is **positive**, the equation has two real roots; if it is **zero**, the equation has one root. If it is **negative**, the equation has no real roots.

The QuadraticRoot() function will prompt the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display two roots. If the discriminant is 0, display one root. Otherwise, display "The equation has no real roots."

Some sample runs:

```
Enter a, b, c: 1.0 3 1
The equation has two roots: -0.38166 and -2.61803

Enter a, b, c: 1 2.0 1
The equation has one root: -1

Enter a, b, c: 1 2 3
The equation has no real roots.
```

Grade Calculator

The GradeCalculator() function will read student scores, get the best score (`bestScore`) from them, and then assign letter grades based on the following scheme:

A: [bestScore, bestScore – 10]
B: [bestScore – 10, bestScore – 20]
C: [bestScore – 20, bestScore – 30]
D: [bestScore – 30, bestScore – 40]
E: Lower than bestScore – 40

The function should ask for a number, indicating how many scores to calculate. After that, a number of scores equal to that number will need to be entered (ex: enter a 3, and then 3 scores). A sample run:

Enter the number of students: 4

```
Enter 4 scores: 40 55 70 58
Student 1 - Score: 40, Letter: C
Student 2 - Score: 55, Letter: B
Student 3 - Score: 70, Letter: A
Student 4 - Score: 58, Letter: B
```

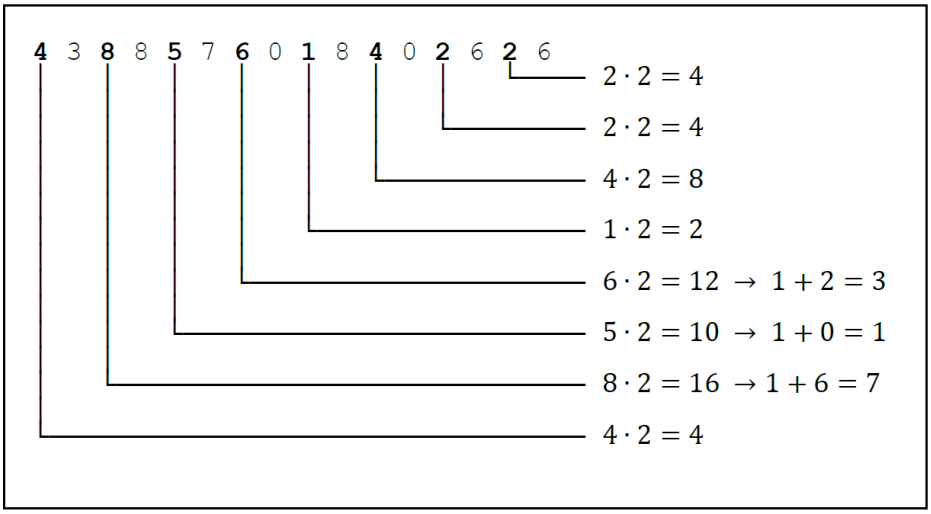
Credit Card Validator

Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. In addition, cards have prefixes by type:

4 – Visa
5 – Mastercard
37 – American Express
6 – Discover

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card is entered correctly or whether a credit card is scanned correctly by a scanner. Credit card numbers are generated following this validity check, commonly known as the Luhn check. The Luhn check can be described as follows. Consider the card number 4388576018402626:

1. Double every second digit from right to left. If doubling of a digit results in a two digit number, add up the digits to get a single-digit number.



2. Now add all single-digit numbers from Step 1.

4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37

3. Add all digits in the odd places from right to left in the card number.

6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38

4. Add the results from the previous two steps.

37 + 38 = 75

5. If the result of the addition is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is valid.

The CreditCardValidator() function will ask the user for a credit card number, and then display whether or not that number is valid. A problem this is made of many smaller steps. Oftentimes breaking a single function into multiple functions makes it easier to understand and implement the solution. To that end, use the following functions:

```
// Return true if the card number is valid.
bool isValid(long long number)

// Get the result of Step 2.
int sumOfDoubleEvenPlace(long long number)
// Return this number if it is a single digit; otherwise, return
// the sum of the two digits.
int getDigit(int number)

// Return sum of odd-place digits in number.
int sumOfOddPlace(long long number)
// Return true if the digit is a prefix for this number.
bool prefixMatched(long long number, int digit)

// Return the number of digits in number
int getSize(long long number)

// Return the first numDigits digits from number. If the no. of
// digits in number is less than numDigits, return number.
long getPrefix(long long number, int numDigits)
```

Here are example runs of the program:

```
Enter a credit card number: 4388576018410707
4388576018410707 is valid.

Enter a credit card number: 4388576018402626
4388576018402626 is invalid.
```

Conversion Tips

- None of the functions in the C++ version will exist in a class. They will all exist in global space.
- In Java, the "long" data type is 64-bits in length.
- In C++, the "long" data type is AT LEAST as large as an integer (but possibly not any larger). A "long long" is a 64-bit variable type, and large enough to hold a credit card number.
- Keywords like "public" and "static" on functions serve very specific purposes--in classes.
- In C++, if you don't know the size of an array at compile time, you need to dynamically allocate it. A common alternative to that is using a vector<>, which is an expandable array. The vector<> data type can take an integer as a parameter to its constructor, which determines its size. After that, you can access elements using brackets [] just like an array. Vectors can do a bit more (and we'll discuss them later in the semester), but this will suffice for now.