CISS240: Introduction to Programming
Assignment 7

Name: ______________________________

Objectives

1. Write complex nested if-else statements

This is a continuation on branching, i.e. on if and if-else statements. The programs are now longer and the logic has a lot more intricate layers. This is to increase your ability to read, analyze, and write longer programs. In some of the questions, you will see repetition of logic, hinting at the need to have some kind of repetition structure for efficient programming.

Note that Q2-Q5 are actually related and solves one problem. This is deliberate: it shows you that usually problem–solving usually involves breaking down a problem into smaller subproblems and solving the subproblems, growing the collection of sub-solutions until you have solved the whole problem. In Q2-Q5, you will see that a problem that is considered very simple to human beings requires a lot of thought and programming logic. Q4 and Q5 will be very "dense". I suggest you try Q1, Q2, Q3 and then jump to Q6 if you find that you need time to think about Q4 and Q5.

Q1. This is a continuation of the "column addition problem". Write a program that shows the work for adding two numbers using column addition; the numbers have at most 3 digits. See the output of the test cases for the format of the output.

Note that the "idea" is similar to the previous "column addition problem" you did earlier. The point here is to make sure that you organize your variables in a systematic way (for instance by using better names) to handle a larger problem.

Note that the test cases listed below is not exhaustive.

Test 1.

```
120 345
  1 2 0
+ 3 4 5
-------
  4 6 5
-------
```

Test 2.

```
120 45
  1 2 0
+   4 5
-------
  1 6 5
-------
```

Test 3.

```
120 5
  1 2 0
+     5
-------
  1 2 5
-------
```

Test 4.

```
123 0
  1 2 3
+     0
-------
  1 2 3
-------
```

Test 5.

```
20 45
    2 0
+   4 5
-------
    6 5
-------
```

Test 6.

```
0 45
      0
+   4 5
-------
    4 5
-------
```

Test 7.

```
17 38
    1
    1 7
+   3 8
-------
    5 5
-------
```

Test 8.

```
17 98
  1 1
    1 7
+   9 8
-------
  1 1 5
-------
```

Test 9.

```
17 298
  1 1
    1 7
+ 2 9 8
-------
  3 1 5
-------
```

Test 10.

```
17 998
  1 1
    1 7
+ 9 9 8
-------
1 0 1 5
-------
```

Test 11.

```
317 998
  1 1
  3 1 7
+ 9 9 8
-------
1 3 1 5
-------
```

Test 12.

```
311 998
  1
  3 1 1
+ 9 9 8
-------
1 3 0 9
-------
```

Test 13.

```
311 978
  3 1 1
+ 9 7 8
-------
1 2 8 9
-------
```

Test 14.

```
311 278
  3 1 1
+ 2 7 8
-------
  5 8 9
-------
```

Q2. Questions 2-5 are essentially the same problem. The goal is to show you how to break a problem down into smaller subproblems, attacking them one at a time. You might want to read all of Q2-Q5 before beginning work on Q2.

The following is the product of two polynomials of degree 2:

$$(ax^2+bx+c)(dx^2+ex+f) = (ad)x^4+(ae+bd)x^3+(af+be+cd)x^2+(bf+ce)x+(cf)$$

The following code segment:

```
int a = 0, b = 0, c = 0, d = 0, e = 0, f = 0;

std::cin >> a >> b >> c >> d >> e >> f;

std::cout << '('
          << a << "x^2 + "
          << b << "x + "
          << c << ")("
          << d << "x^2 + "
          << e << "x + "
          << f << ") = "
          << a * d << "x^4 + "
          << (a * e + b * d) << "x^3 + "
          << (a * f + b * e + c * d) << "x^2 + "
          << (b * f + c * e) << "x + "
          << (c * f) << std::endl;
```

computes the product of two polynomial with integer coefficients. For instance this is an execution of the program:

```
1 2 3 4 5 6
(1x^2 + 2x + 3)(4x^2 + 5x + 6) = 4x^4 + 13x^3 + 28x^2 + 27x + 18
```

Modify the program so that zero terms are not printed. Here's a test case:

```
0 1 1 0 1 1
(1x + 1)(1x + 1) = 1x^2 + 2x + 1
```

Note that none of the `0x^2` terms are printed.

Test 1.

```
0 1 1 0 1 1
(1x + 1)(1x + 1) = 1x^2 + 2x + 1
```

Test 2.

```
0 1 0 1 1 1
(1x)(1x^2 + 1x + 1) = 1x^3 + 1x^2 + 1x
```

Test 3.

```
0 1 1 0 1 -1
(1x + 1)(1x + -1) = 1x^2 + -1
```

Test 4.

```
2 0 1 0 1 -1
(2x^2 + 1)(1x + -1) = 2x^3 + -2x^2 + 1x + -1
```

Test 5.

```
1 0 2 2 0 1
(1x^2 + 2)(2x^2 + 1) = 2x^4 + 5x^2 + 2
```

Test 6.

```
2 -3 0 3 2 0
(2x^2 + -3x)(3x^2 + 2x) = 6x^4 + -5x^3 + -6x^2
```

Test 7.

```
3 4 0 3 -4 0
(3x^2 + 4x)(3x^2 + -4x) = 9x^4 + -16x^2
```

Test 8.

```
2 0 3 -2 0 3
(2x^2 + 3)(-2x^2 + 3) = -4x^4 + 9
```

Q3. This is a continuation of Q2.

Modify the program so that instead of printing `1x`, `1x^2`, `1x^3`, `1x^4`, the program prints `x`, `x^2`, `x^3`, `x^4` (respectively). The following is a test case:

```
0 1 1 0 1 1
(x + 1)(x + 1) = x^2 + 2x + 1
```

Note that `(x + 1)` is printed instead of `(1x + 1)`.

Make sure your code passes all the tests in the previous question.

Test 1.

```
0 1 1 0 1 1
(x + 1)(x + 1) = x^2 + 2x + 1
```

Test 2.

```
0 1 1 0 1 -1
(x + 1)(x + -1) = x^2 + -1
```

Test 3.

```
1 1 1 0 1 0
(x^2 + x + 1)(x) = x^3 + x^2 + x
```

Test 4.

```
1 2 3 0 1 -1
(x^2 + 2x + 3)(x + -1) = x^3 + x^2 + x + -3
```

Q4. This is a continuation of Q3.

Make sure that your program prints a zero whenever a polynomial has zeroes for all its coefficients. The following is a test case:

```
0 1 1 0 0 0
(x + 1)(0) = 0
```

Make sure your code passes all tests in the previous question.

Test 1.

```
0 1 1 0 0 0
(x + 1)(0) = 0
```

Test 2.

```
0 0 0 1 2 3
(0)(x^2 + 2x + 3) = 0
```

Test 3.

```
3 0 0 1 2 0
(3x^2)(x^2 + 2x) = 3x^4 + 6x^3
```

Test 4.

```
1 2 3 4 5 6
(x^2 + 2x + 3)(4x^2 + 5x + 6) = 4x^4 + 13x^3 + 28x^2 + 27x + 18
```

Q5. This is a continuation of Q4.

Modify your program to handle negative coefficients as follows. Here's a test case

```
0 1 1 0 1 -1
(x + 1)(x - 1) = x^2 - 1
```

Note that for the second polynomial on the left, instead of printing `(x + -1)` the program prints `(x - 1)`. Likewise, the polynomial on the right, instead of printing `x^2 + -1` the program prints `x^2 - 1`.

Make sure your code passes all tests in the previous question.

Test 1.

```
0 1 1 0 1 -1
(x + 1)(x - 1) = x^2 - 1
```

Test 2.

```
0 1 -1 0 1 -1
(x - 1)(x - 1) = x^2 - 2x + 1
```

Test 3.

```
0 1 -2 0 1 -3
(x - 2)(x - 3) = x^2 - 5x + 6
```

Test 4.

```
1 -2 3 0 -1 0
(x^2 - 2x + 3)(-x) = -x^3 + 2x^2 - 3x
```

Q6. You are working for a game company and you have been asked to write the logic for a smart tic-tac-toe AI agent. (I assume you know the tic-tac-toe game ... otherwise ... hmmm ...)

Here's the game board:

and here's what we call a game state (i.e., a snapshot of the game in progress):

X	O	-
X	-	-
O	-	X

I use `-` to denote the fact that a square is available.

We can model the game state with 9 variables. I'll call them `board00`, `board01`, `board02`, `board10`, `board11`, `board12`, `board20`, `board21`, and `board22`. The variables will hold character values, i.e., they are `char` variables. For instance for the game state above

X	O	-
X	-	-
O	-	X

the variables have the following values:

```
board00 = 'X',  board01 = 'O',  board02 = '-'
board10 = 'X',  board11 = '-',  board12 = '-'
board20 = 'O',  board21 = '-',  board22 = 'X'
```

For memory aid, the variables are named according to this format: `board` is followed by two numbers, the row number and the column number where rows and columns are numbered starting from 0. For instance `board12` refers to the character at row 1 and column 2. Run this program

```
#include <iostream>

int main()
{
    char board00, board01, board02;
    char board10, board11, board12;
    char board20, board21, board22;
    std::cin >> board00 >> board01 >> board02
             >> board10 >> board11 >> board12
             >> board20 >> board21 >> board22;

    std::cout << "+-+-+-+\n"
              << '|' << board00 << '|' << board01 << '|' << board02 << "|\n"
              << "+-+-+-+\n"
              << '|' << board10 << '|' << board11 << '|' << board12 << "|\n"
              << "+-+-+-+\n"
              << '|' << board20 << '|' << board21 << '|' << board22 << "|\n"
              << "+-+-+-+\n";

    return 0;
}
```

Run it with input

```
X O - - - - O - X
```

and you will get this output

```
+-+-+-+
|X|O|-|
+-+-+-+
|-|-|-|
+-+-+-+
|O|-|X|
+-+-+-+
```

Your goal is to write a program that prompts for a game state and decide on a good move for the player playing `X` by printing a suggested row and column.

```
X O - - - - O - X
```

and you will get this output

```
+-+-+-+
|X|O|-|
+-+-+-+
|-|-|-|
+-+-+-+
|O|-|X|
+-+-+-+
1 1
```

since in this case is `X` is placed at row 1 and column 1, the player playing `X` will have a winning diagonal.

The strategy is as follows. The following is a list of actions to take. You should use the one listed earlier if it's possible to execute that action. For instance if you can either have winning row (see item 1) or a winning column (see item 2), then your strategy is to take the winning row (item 1).

1. If there's a winning row, take the winning row, preferring the topmost when there is more than one.
2. If there's a winning column, take the winning column, preferring the leftmost when there is more than one.
3. If there's a winning diagonal that goes from top-left to bottom-right, take the diagonal.
4. If there's a winning reverse diagonal that goes from top-right to bottom-left, take the winning reverse diagonal.
5. Block player `O` from making a winning row, column, diagonal, or reverse diagonal in the same order of preference as above. (In other words, if you can block a top row and bottom row, then you should block a top row.)
6. Take the center.
7. Take a corner, looking for an available corner going top-to-bottom, left-to-right.
8. If none of the above works, take one of the remaining positions, looking for an available position going top-to-bottom, left-to-right.

For this question, I will only test your code for scenarios corresponding to items 1–4. You are of course more than welcome to code up all the items if you like.

[Note: The above strategy considers the current game state and not future game ststes, i.e., it does not consider all the possible moves by the opponent. A strategy that does not consider all cases and in particular only considers the current data is called a *greedy* algorithm. A smarter AI strategy would consider possible future scenarios as well. However when a game complex, you usually cannot consider all scenarios. For instance the total number of chess games can be as large as 10^{120}; the number of atoms in the observable universe is approximately 10^{80}. If your program considers all possible scenarios, then it's called a *brute force* algorithm. You can learn more about algorithms for solving computationally intensive problems in the AI class CISS450. For such cases, one would need to use *heuristic* algorithms, otherwise you would need trillions and trillions of years to compute a single chess move!]

You only need to know that you can compare character values. Try this on your own:

```
char a = '!', b = '@', c = '!';
std::cout << (a == b) << '\n';
std::cout << (a == c) << '\n';
std::cout << (a != b) << '\n';
std::cout << (a != c) << '\n';
```

You can of course also perform input/output for `char` variables:

```
char a = ' ', b = ' ', c = ' ';
std::cin >> a >> b >> c;
std::cout << a << ' ' << b << ' ' << c << '\n';
```

You should use the following skeleton code. [Note: Skeleton code is definitely incomplete and might contain errors.]

```
#include <iostream>

int main()
{
    const char PLAYER = 'X';
    const char ENEMY = 'O';
    const char SPACE = '-';

    char board00, board01, board02;
    char board10, board11, board12;
    char board20, board21, board22;
    std::cin >> board00 >> board01 >> board02
             >> board10 >> board11 >> board12
             >> board20 >> board21 >> board22;

    std::cout << "+-+-+-+\n"
              << '|' << board00 << '|' << board01 << '|' << board02 << "|\n"
              << "+-+-+-+\n"
              << '|' << board10 << '|' << board11 << '|' << board12 << "|\n"
              << "+-+-+-+\n"
              << '|' << board20 << '|' << board21 << '|' << board22 << "|\n"
              << "+-+-+-+\n";

    int row = -1, col = -1;

    if (board00 == SPACE && board01 == PLAYER && board02 == PLAYER)
    {
        // Take (0, 0) for a winning row 0
        row = 0;
        col = 1;
    }
    else
    {
        if (board00 == PLAYER && board01 == SPACE && board02 == PLAYER)
        {
            // Take (0, 1) for a winning row 0
            row = 0;
            col = 2;
        }
        // MORE CODE HERE
    }

    std::cout << row << ' ' << col << std::endl;

    return 0;
}
```

Preventing indentation in a stack of if-else

Note that if you have code like this where in the `else` block you only have an `if` statement or an `if-else` statement:

```
if (bool1)
{
   ...
}
else
{
    if (bool2)
    {
        ...
    }
    else
    {
        if (bool3)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

then, it's the same as the following (the `else` has only one `if-else` statement so the block is redundant – check your notes):

```
if (bool1)
{
   ...
}
else
    if (bool2)
    {
        ...
    }
    else
        if (bool3)
        {
            ...
        }
        else
        {
            ...
        }
```

which is the same as the following (because C++ ignores whitespaces – check your notes):

```
if (bool1)
{
   ...
}
else if (bool2)
{
    ...
}
else if (bool3)
{
    ...
}
else
{
    ...
}
```

In other words, we join an `if` with the preceding `else`.

Writing it this way will prevent too much indentation toward the right and make your program look like a table. This is one of the very few cases where we ignore the standard indentation rules in order not to over–indent.

Note that this won't work if the `else` contains more than just an `if-else`, for instance

```
if (bool1)
{
   ...
}
else
{
    x = 1;
    if (bool2)
    {
        ...
    }
    else
    {
        ...
    }
}
```

In this case, since the outermost `else` has two statements, you cannot get rid of the block symbols `{}` to join up the inner `if` with the outer `else`.