Lab 2. Loops, Arrays, and Number Sequences.

How to submit your programs.

Each program should be submitted through Gradescope.

Write separate programs for each part of the assignment.

Submit only the source code (.cpp) files, not the compiled executables.

Each program should start with a comment that contains your name and a short program description, for example:

```
/*
Author: your name
Course: CSCI-136
```

```
Instructor: their name
Assignment: title, e.g., Lab1A

Here, briefly, at least in one or a few sentences
describe what the program does.
*/
```

Task A. Input validation.

Write a program `valid.cpp`, which asks the user to input an integer in the range ***0 < n < 100***. If the number is out of range, the program should keep asking to re-enter until a valid number is input.

After a valid value is obtained, print this number *n* squared.

Example

```
$ ./valid
Please enter an integer: -10
Please re-enter: 1200
Please re-enter: 100
Please re-enter: 7

Number squared is 49
```

Hint: You can use a `while` or `do ... while` loop.

Task B. Print all integers from the requested interval.

Write a program `print-interval.cpp` that asks the user to input two integers `L` and `U` (representing the lower and upper limits of the interval), and print out all integers in the range ***L ≤ i < U*** separated by spaces. Notice that we include the lower limit, but exclude the upper limit.

Example

```
$ ./print-interval
Please enter L: -5
Please enter U: 10

-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9
```

You can use any loop construct to achieve this result, but the most idiomatic way to do such an iteration in C++ is to use a `for` loop that starts with a counter variable `i = L` and keeps iterating while `i < U`:

```
for(int i = L; i < U; i++) {
    // the body of the loop will get executed for
    // all values of i starting at L
    // and ending at the largest integer less than U

}
```

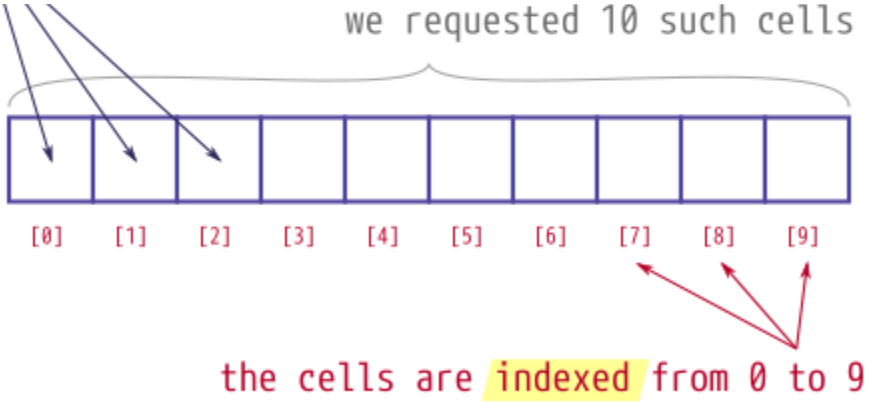
As a side note, also notice that if you change the middle condition in the loop to `i <= U`, then the iteration would go from `L` to `U` inclusive its upper bound (***L ≤ i ≤ U***).

Introducing arrays.

If you need to store values in a table-like fashion, C++ provides us with the array data structure. The following code

```
int myData[10];
```

creates an array called `myData` with 10 cells each storing integer values:



The cells of the array are indexed from 0 through 9, and can be accessed by specifying their index in square brackets, `myData[i]`. **You can treat arrays as tables whose elements can be accessed by their index.** For example:

```
myData[0] = 1984; // update element at index 0
myData[7] = 1975; // update element at index 7
myData[2] = 1966; // update element at index 2

cout << myData[2] << endl; // print the value of the cell
                           // at the index 2 on the screen,
                           // which is 1966.
```

Task C. Using arrays to store, update, and retrieve numbers.

Write a program `edit-array.cpp` that creates an array of 10 integers, and provides the user with an interface to edit any of its elements. Specifically, it should work as follows:

1. Create an array `myData` of `10` integers.
2. Fill all its cells with value `1` (using a `for` loop).
3. Print all elements of the array on the screen.
4. Ask the user to input the cell index `i`, and its new value `v`.
5. If the index `i` is within the array range (0 ≤ i < 10), update the asked cell, `myData[i] =`

`v`, and go back to the step 3. Otherwise, if index `i` is out of range, the program exits.

The repetition of the steps 3-4-5 can be implemented with a `do while` loop:

```
// make array and fill it with 1
do {
    // print the array
    // get i and v from the user
    // if i is good, update the array at index i
} while (...); // if the index was good, repeat
```

The program should keep running until the user inputs an out-of-range (invalid) index.

Example:

```
$ ./edit-array

1 1 1 1 1 1 1 1 1 1

Input index: 8
Input value: 99

1 1 1 1 1 1 1 1 99 1

Input index: 0
Input value: 300

300 1 1 1 1 1 1 1 99 1

Input index: 10
Input value: 5

Index out of range. Exit.
```

Test your program. Check that all array elements are editable, for instance, use the program interface to make it print out sequence `5 10 15 20 25 30 35 40 45 50`. Check that all possible edge cases are correctly handled, and the program exits when invalid index is input.

Task D. Computing Fibonacci Numbers with Loops and Arrays 0, 1, 1, 2, 3, 5, 8, 13...

First, a quick intro:

Fibonacci numbers is a sequence of numbers that starts with ***F(0) = 0*** and ***F(1) = 1***, with all the following numbers computed as the sum of two previous ones, ***F(n) = F(n−1) + F(n−2)***:

0

1

1 (=1+0)

2 (=1+1)

3 (=2+1)

5 (=3+2)

8 (=5+3)

13 (=8+5)

… and so on …

To make a C++ program to keep track of the previous numbers so that we can compute the new ones, we can use an array of integers:

```
// make an array
int fib[60];
// first two terms are given
fib[0] = 0
fib[1] = 1
// and all the following ones can be computed iteratively as
fib[i] = fib[i-1] + fib[i-2]
```

The task:

Write a program `fibonacci.cpp`, which uses an array of ints to compute and print all Fibonacci numbers from *F(0)* to *F(59)*.

Example:

```
0
1
1
2
3
5
8
13
...
```

Once your program is complete and works, check carefully the values printed on the screen. Specifically, what is happening when the numbers approach two billions? We expect that at some point the numbers start diverging from what they should be. Describe what you observe and explain why it is happening in a program comment.