CISS240: Introduction to Programming
Assignment 13

Name: _____

Objectives

1. Use arrays
2. Write functions
3. Write functions with array parameter
4. Use an array to simulate a container of values where the number of values is from 0 to the size of the array.

You only need to complete Q1, Q2, Q3. Q4 is an optional exercise.

Q1. Complete the following program:

```
#include <iostream>


void println(int x[], int xSize)
{
    for (int i = 0; i < xSize; i++)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << std::endl;
}


//------------------------------------------------------------------------
// Swap the values of x[i] and x[j]
// For instance if the array arr has values 9, 8, 7, 6, then on return
// from calling swap(arr, 1, 3), arr will have values 9, 6, 7, 8.
//------------------------------------------------------------------------
void swap(int x[], int i, int j)
{
}


//------------------------------------------------------------------------
// Performs bubble sort algorithm on x[0], ..., x[xSize - 1] (in
// ascending order)
//------------------------------------------------------------------------
void bubbleSort(int x[], int xSize)
{
}


//------------------------------------------------------------------------
// Returns index where target appears in array x. If target is not found,
// -1 is returned.
//------------------------------------------------------------------------
int binarySearch(int x[], int xSize, int target)
{
}
```

```
int main()
{
    //-------------------------------------------------------------------
    // Declare and prompt user for values to be stored in array x
    //-------------------------------------------------------------------
    const int SIZE = 8;
    int x[SIZE] = {0};
    for (int i = 0; i < SIZE; i++)
    {
        std::cin >> x[i];
    }

    //-------------------------------------------------------------------
    // Sort and print array x
    //-------------------------------------------------------------------
    bubbleSort(x, SIZE);
    println(x, SIZE);

    //-------------------------------------------------------------------
    // Prompt for target to search in array. The index of target in the
    // array is displayed. If target is not found, -1 is printed.
    //-------------------------------------------------------------------
    int target = 0;
    std::cin >> target;
    std::cout << binarySearch(x, SIZE, target)
              << std::endl;

    return 0;
}
```

Note that your `bubbleSort()` function must use the `swap()` function where appropriate. Although the `main()` tests your functions with an array of size 8, your functions should work for arrays of any size. In other words, you must not hardcode 8 in your functions.

Test 1.

```
3 5 2 1 5 12 4 3
1 2 3 3 4 5 5 12
0
-1
```

Test 2.

```
3 5 2 1 5 12 4 3
1 2 3 3 4 5 5 12
1
0
```

Test 3.

```
3 5 2 1 5 12 4 3
1 2 3 3 4 5 5 12
12
7
```

Test 4.

```
3 5 2 1 5 12 4 3
1 2 3 3 4 5 5 12
6
-1
```

Q2. The concept of an array can simulate the notation of a "container". A container is conceptually a data structure (i.e., variable(s) and function(s) together) that allows us to hold values and we can either put values into or get values out from the container. A container starts off by being empty:

[ ]

Suppose we put 5 into the container. Conceptually the container now looks like this:

[5]

At this point, suppose we want to put 9 into the container. Where will it go? After putting the value 9 into the container, the container can be either

[9, 5]

or

[5, 9]

So we need to be more precise about describing the operation. Note also that our concept of an array has a fixed size whereas a container's size can change. How do we use an array to model a container? We need an array and another variable to denote the number of things already placed in the array. Let's call our array `x` and suppose that it's an array of integer values of size 5. Let's say we initialize the array with zeroes. Let's call the number of things in the container `len` (for length). For instance,

concept to model	variables implementing the concept
[5, 9]	`x = {5, 9, 0, 0, 0}, len = 2`

On the left we see the concept we want to model and on the right we see the actual implementation in a programming language (for instance C++.) Of course, array `x` actually has 5 values. But the `len` variable tells us that we should only consider `x[0]` and `x[1]` as values we put into the container; we simply ignore other values `x[2]`, `x[3]`, `x[4]`. Since we ignore `x[2]`, `x[3]`, `x[4]`. This could work too:

concept to model	variables implementing the concept
[5, 9]	`x = {5, 9, 78, 79, 80}, len = 2`

If the container has values $5, 3, 6, 2$ then

concept to model	variables implementing the concept

[5, 3, 6, 2]	`x = {5, 3, 6, 2, 0}, len = 4`

Since `len` has value `4`, we only consider `x[0]`, `x[1]`, `x[2]`, and `x[3]` values in the container and ignore `x[4]`. The following works too:

concept to model	variables implementing the concept
[5, 3, 6, 2]	`x = {5, 3, 6, 2, 99}, len = 4`

From now on, I will write `?` for values in the array that we will ignore. So in the previous example above, I will write this:

concept to model	variables implementing the concept
[5, 3, 6, 2]	`x = {5, 3, 6, 2, ?}, len = 4`

Remember that when you see `?` it does not mean that there's no value there – there *is* a value. It's just a value we don't care about.

There are two main things you can do to a container: you can put something into the container and the you take something out of the container. Let me explain these two operations.

Let's call the operation to put a value into the container "insert". Initially, the container looks like this:

concept to model	variables implementing the concept
[ ]	`x = {?, ?, ?, ?, ?}, len = 0`

In other words, initially, the container has nothing.

We can insert a value into the container. To do so, we have to specify the index position. For instance when we insert a 5 at index position 0 we get this:

concept to model	variables implementing the concept
[5]	`x = {5, ?, ?, ?, ?}, len = 1`

Now the container has one value, i.e., 5 at index 0. If we now insert 9 at index position 0 we get this:

concept to model	variables implementing the concept
[9, 5]	`x = {9, 5, ?, ?, ?}, len = 2`

Notice that 5 is shifted to the right by one position. Now if we insert 11 at index position 1 we get this:

concept to model	variables implementing the concept
[9, 11, 5]	`x = {9, 11, 5, ?, ?}, len = 3`

If we insert 1 at index position 3 we get

concept to model	variables implementing the concept
[9, 11, 5, 1]	`x = {9, 11, 5, 1, ?}, len = 4`

Now let me talk about removing a value from the container. If we remove the value at position 2, we get

concept to model	variables implementing the concept
[9, 11, 1]	`x = {9, 11, 1, ?, ?}, len = 3`

Look at the array `x` and `len` carefully. Note that the `len` $= 3$ tells us that the container we're modeling has three values, therefore `x[0]` $= 9$, `x[1]` $= 11$, `x[2]` $= 1$ are the only values in the container; we ignore `x[3]` and `x[4]`.

If we now remove the value at position 0, we get

concept to model	variables implementing the concept
[11, 1]	`x = {11, 1, ?, ?, ?}, len = 2`

Get it? Note that you can only insert and remove at certain places. For instance, right now we have

concept to model	variables implementing the concept
[11, 1]	`x = {11, 1, ?, ?, ?}, len = 2`

You can only insert at position $0, 1, 2$ and remove at position $0, 1$.

For this assignment, if you insert or remove not within the valid range, the variables are not changed. For instance, if we insert 7 at position 4 or remove the value at position 3, the container is still the same:

concept to model	variables implementing the concept
[11, 1]	`x = {11, 1, ?, ?, ?}, len = 2`

In the general case, if the container has length `len`, you can insert at index positions $0, 1, 2, \ldots,$ `len` if `len` is less than the size of the array and you can remove the value at index position $0, 1, 2, \ldots,$ `len` $- 1$.

The goals of this assignment is to build such a container using an array and a length variable.

(Note that if you restrict the usage of the container so that you only insert at position `len - 1` and remove only at the index position `0`, i.e., values go into the container at one end and go out from the *other end*, you are then simulating a **queue**. Queues occur frequently in the real world. Examples include a manufacturing line. An industrial engineer is interested in, for instance, how frequently the queue of a manufacturing line is congested based on the probabilistic distribution of items entering and leaving the queue. Queues are also used for computer network communication. Now, suppose you only insert at `len - 1` and remove at `len - 1`, i.e., the values go into the container at one end and also the container at the *same end*, then you're simulating a **stack**. You have seen that before: think of a stack of plates at a buffet: plates go into the container of plates at the top and plates leave the container at the top. Stacks are called last-in-first-out (LIFO) containers while queues are called first-in-first-out (FIFO) containers. Both the stack and queue are very important and are used extensively in computer science including artificial intelligence, computer networks, computer graphics, etc.)

The following code skeleton is given. Here we are simulating our container with a maximum size of 5. The container contains integer values (the main idea can be applied to an array of any type of values.) The program continually prompts the user for an option: 0 is to quit, 1 is to insert, 2 is to remove. If the user enters an option not in the above choices, the program reprompts the user. If the user enters 1, the program prompts the user for a position and a value to insert into the container. If the user enters 2, the program prompts the user for a position and removes the value at the position. If the user enters an invalid position, the program does not perform the requested option. If the user attempts to insert a value into the container when it's already full, no action is taken. See the test cases.

Your functions must work for arrays of any size. Therefore, you should not assume the array has at most 5 elements. (See hints below.)

Note that the `insert` and `remove` functions do not only change the array, they must change the `len` variable as well. You already know that functions cannot change the values of variables in the calling function. For instance,

```
#include <iostream>
```

```
void inc(int x)
{
    x++;
}

int main()
{
    int a = 5;
    inc(a);
    std::cout << a << std::endl; // value of a is still 5

    return 0;
}
```

If you want the function to modify the variable in the caller function, you make the variable a reference variable. Try this:

```
#include <iostream>

void inc(int & x) // x *will* affect the a in main()
{
    x++;
}

int main()
{
    int a = 5;
    inc(a);
    std::cout << a << std::endl; // a is changed!

    return 0;
}
```

Note that parameter "`int x`" is changed to "`int & x`". This tells C++ that the parameter will affect the corresponding variable in the caller function. Note that in the functions `insert()` and `remove()`, the `len` parameter is a reference variable. Note once again that our container is described by three variables: the array `x`, the `len` which denotes the number of things in the container, and `size` which describes the maximum number of values you can have in the container.

```
#include <iostream>

void println(int x[], int len)
{
    std::cout << "[ ";
    for (int i = 0; i < len; i++)
    {
        std::cout << x[i] << ' ';
```

```
    }
    std::cout << "]" << std::endl;
}


// Inserts newvalue into container x at position index
void insert(int x[], int & len, int size, int index, int newvalue)
{
}


// Removes the value at position index of the container x
void remove(int x[], int & len, int size, int index)
{
}


int main()
{
    const int SIZE = 5;
    int x[SIZE] = {0};
    int len = 0;
    println(x, len);

    while (1)
    {
        int option = 0;
        std::cout << "option (0-quit, 1-insert, 2-remove): ";
        std::cin >> option;
        // Break the while loop if option is 0

        // Prompt for index
        switch (option)
        {
             case 1:
                 // Prompt for new value and call the insert() function.
                 break;

             case 2:
                 // Call the remove() function.
                 break;
        }

        std::cout << std::endl;
        println(x, len);
    }

    return 0;
}
```

Test 1.

```
[ ]
option (0-quit, 1-insert, 2-remove): 0
```

Test 2.

```
[ ]
option (0-quit, 1-insert, 2-remove): 1
index: 7
value: 2

[ ]
option (0-quit, 1-insert, 2-remove): 2
index: 4

[ ]
option (0-quit, 1-insert, 2-remove): 0
```

Test 3.

```
[ ]
option (0-quit, 1-insert, 2-remove): 1
index: 0
value: 7

[ 7 ]
option (0-quit, 1-insert, 2-remove): 1
index: 0
value: 3

[ 3 7 ]
option (0-quit, 1-insert, 2-remove): 1
index: 1
value: 2

[ 3 2 7 ]
option (0-quit, 1-insert, 2-remove): 1
index: 3
value: 8

[ 3 2 7 8 ]
option (0-quit, 1-insert, 2-remove): 1
index: 1
value: 4

[ 3 4 2 7 8 ]
option (0-quit, 1-insert, 2-remove): 2
index: 0
```

```
[ 4 2 7 8 ]
option (0-quit, 1-insert, 2-remove): 2
index: 4

[ 4 2 7 8 ]
option (0-quit, 1-insert, 2-remove): 2
index: 3

[ 4 2 7 ]
option (0-quit, 1-insert, 2-remove): 2
index: 1

[ 4 7 ]
option (0-quit, 1-insert, 2-remove): 2
index: 1

[ 4 ]
option (0-quit, 1-insert, 2-remove): 2
index: 0

[ ]
option (0-quit, 1-insert, 2-remove): 2
index: 0

[ ]
option (0-quit, 1-insert, 2-remove): 2
index: 0

[ ]
option (0-quit, 1-insert, 2-remove): 0
```

Hint: SPOILERS AHEAD!!! WATCHOUT!!!

- If you want to insert at a position, you need to move the values starting at the index position to the right by one from the index position where insertion is to occur. Use a for–loop!!! Once that's done, you put the new value at the index position where insertion should occur. You should increment the `len` variable. For instance, suppose the array has `len` 5 (and is, say, of `size` 10) and you need to insert a value of 6 at index position 2. Say the array looks like this:

  $$\texttt{x} : 3, 7, 2, 7, 8, ?, ?, ?, ?, ?$$

  You need to write a for–loop to copy the values from index 2 onward to the right by one step:

  $$\texttt{x} : 3, 7, 2, 7, 8, ?, ?, ?, ?, ?$$
  $$\downarrow$$
  $$\texttt{x} : 3, 7, 2, \underline{\mathbf{2}, \mathbf{7}, \mathbf{8}}, ?, ?, ?, ?$$

  and then you put 6 at index 2:

  $$\texttt{x} : 3, 7, \underline{\mathbf{6}}, 2, 7, 8, ?, ?, ?, ?$$

- If you want to remove the value at an index position, you move all the values to the right of that index position to the left by one index position. Once that's done, you need to decrement the `len` variable. The idea is very similar to the case of inserting a value into the array except that you're moving a chunk of values in the opposite direction. For instance, suppose you have the following with `len` 6:

  $$\texttt{x} : 3, 7, 6, 2, 7, 8, ?, ?, ?, ?$$

  Then, to remove the value at index position 3, (i.e., the value 2) you simply copy all the values to the right of 2

  $$\texttt{x} : 3, 7, 6, 2, \underline{\mathbf{7}, \mathbf{8}}, ?, ?, ?, ?$$

  to the left by one to get this:

  $$\texttt{x} : 3, 7, 6, \underline{\mathbf{7}, \mathbf{8}}, ?, ?, ?, ?, ?$$

  and decrement the `len` variable by 1.

Q3. The following is a method for generating primes up to a given upper bound. For our first example, suppose we want to compute the primes up to 99. First, we write down the integers from 0 to 99:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

First of all 0 and 1 are not primes: By definition a prime is at least 2. So first cross out 0 and 1:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Now we begin the actual algorithm. The first number that is not crossed out is 2. Cross out all the multiples of 2 in the above table except for 2 itself. That means crossing out $4, 6, 8, 10, \ldots$ in the list. Think of 2 as a prime that kills off numbers which are not primes because they are divisible by 2; 2 is a killer prime.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Now we move from 2 to the next number that is not crossed out. From the table you see that it is 3: 3 is the next killer prime. We now cross out numbers which are multiples of 3 except for 3 itself. Therefore we cross out $6, 9, 12, 15, \ldots$. (Note that 6 is crossed out twice but that's fine. This basically means that 6 is not prime because both 2 and 3 divide 6.) We get the following:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

We now move from 3 to the next number that is not crossed out. Note that in this case 4 is already crossed out. The number after that is 5: 5 is the next killer prime. We now cross out all the multiples of 5 except for 5 itself, i.e., we cross out $10, 15, 20, \ldots$ and we get this:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

We move on to the first number after 5 that is not crossed out. In this case the next killer prime is 7. We cross out all the multiples of 7 except for 7 itself to obtain:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

The first number after 7 that is not crossed out is 11. Note that this algorithm terminates when you reach a killer prime that is greater than the square root of the upper bound which in this case is 99. The square root of 99 is approximately $9.9489\ldots$. At this point, the number we're looking at is 11 which is greater than $9.9489\ldots$. Hence the algorithm stops. Why? If you look at killer prime 11, you notice that the multiples it crosses out are $22, 33, 44, 55, \ldots, 99$. They are already crossed out by primes less than 11. The fact that in this algorithm the primes larger than the square root of 99 will only cross out numbers which are already crossed out can be proven mathematically (although I won't do it here.)

The numbers in the table that are not crossed out must be primes. From the table

we see that these numbers are:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97$$

There are 25 primes less than 100.

This method works for any upper bound n. For instance, to compute all the primes upto $n = 250$, you write down a table of integers from 0 to 250, cross out 0 and 1, cross out all the multiples of 2 except for 2, cross out the multiples of 3 except for $3, \ldots$. In general, after each crossing out, you move on to the next number that is not crossed out and repeat the process. This algorithm terminates when the killer prime is larger than the square root of the upper bound, i.e., 250.

Write a function

```
void es(int n, int prime[], int & len);
```

that prints the list of remaining integers at the end of each crossing out stage. For instance, on calling `es(99, prime, len)`, the function goes through the above algorithm, and prints the following:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
2: 2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97
99
3: 2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 65 67 71 73
77 79 83 85 89 91 95 97
5: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 49 53 59 67 71 73 77 79 83 85
89 91 97
7: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Note that you must first print all the number from 2 to the upper bound, then for each "killer prime", print the killer prime and the resulting array after the killer prime has removed the relevant integers, and, at the end of the algorithm, the array `prime` is set to contain the following values:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

and `len` is set to 25. The `main()` that tests the function `es()` is

```
int main()
```

```
{
    int n, len;
    int prime[1000];
    std::cin >> n;

    es(n, prime, len);
    std::cout << "\nprimes from 2 to " << n << ":\n";
    for (int i = 0; i < len; i++)
    {
        std::cout << prime[i] << ' ';
    }
    std::cout << "len: " << len << std::endl;

    return 0;
}
```

Test 1.

```
10
2 3 4 5 6 7 8 9 10
2: 2 3 5 7 9
3: 2 3 5 7

primes from 2 to 10:
2 3 5 7
len: 4
```

Test 2.

```
20
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2: 2 3 5 7 9 11 13 15 17 19
3: 2 3 5 7 11 13 17 19

primes from 2 to 20:
2 3 5 7 11 13 17 19
len: 8
```

Test 3.

```
50
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
2: 2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
3: 2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49
5: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 49
7: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

primes from 2 to 50:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
len: 15
```

Test 4.

```
99
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
2: 2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
3: 2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 65 67 71 73
77 79 83 85 89 91 95 97
5: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 49 53 59 67 71 73 77 79 83 85
89 91 97
7: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

primes from 2 to 99:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
len: 25
```

Note: The important thing about this function is that it computes and stores primes in an array. The printing of the work done is not that important. The only reason why I'm requiring the printing is to verify that you are coding the algorithm correctly.

Spoiler Warning . . . Hints Ahead!!! Watchout!!!

HERE IT COMES . . .

HINT:

Use an array `x` of integers. Set all the values of `x` to 0. Crossing out the number `5` corresponds to the action of setting `x[5]` to 1. In this case I'm using 0 to mean "not crossed out" and 1 to mean "crossed out". You can switch the meaning of 0 and 1. The only reason why I'm using 0 to mean not crossed out is because it's easier to initialize all values to 0 rather than 1. (An array of boolean values can also be used since we only need to store 2 possible values in the array.)

Q4. The goal is to write a simple calculator. This program will give you an idea how string parsing and interpretation of data works in a compiler or interpreter (which will be helpful in CISS360, CISS445, CISS375) and how memory is used (which will be helpful in the above courses and also in your understanding of pointers).

I will describe the features in stages to make it easier for you. Less hints will be given in later stages.

PART 1. First when you run the program you see the following prompt:

```
>>>
```

When you press the enter key (i.e., you entered a blank line) you see this:

```
>>>
>>>
```

In other words, when you enter a blank line, the program does not compute anything, prints the prompt again, and waits for an input. This is the same if you enter spaces or tabs followed by pressing the enter key.

You can exit the program by typing `q` (and pressing the enter key):

```
>>>
>>> q
```

(the program ends.) Note that there is a space between the input and the prompt. In other words the prompt is `">>> "` with a space after the last `>`. The following skeleton will be helpful:

```
#include <iostream>

// Print prompt and get a string from user
void get_input(char input[])
{
}

int main()
{
    char input[1024] = "";

    while (1)
    {
         get_input(input);
    }

    return 0;
```

```
}
```

PART 2. Besides `q`, the program accepts an integer expression with the operator `+`, `-`, `/`, or `*` . For instance:

```
>>> 1+2
3
>>> 1 - 3
-2
>>>   -3      -1
-4
>>> 2 *       5
10
>>> 4      / -2
-2
>>>     0 - -1
1
>>>
```

Note that the evaluated integer value is printed. Note also that redundant whitespaces can be included in the input. As shown in the above examples, you must allow negative integer values.

The following skeleton can be used:

```
const int PLUS = 0;
const int MINUS = 1;
const int MULT = 2;
const int DIV = 3;

// ... get_input ...

// Returns an integer value from the characters in input starting at position i
// For instance:
// 1. If the input is "123" and i is 0 (the index position of character '1', then
// the integer 123 is returned and i is set to 3.
// 2. If the input is "123 + -456" and is 0 (the index position of character '1',
// then the integer 123 is returned and i is set to 3.
// 3. If the input is "123 + -456" and is 5 (the index position of character ' '
// after the '+', then the integer -456 is returned and i is set to 10.
int get_int(char input[], int & i)
{
}
```

```
// Returns one of the following:
// * PLUS if the character in input starting at i (with whitespaces ignored) is
//   '+'.
// * MINUS if the character in input starting at i (with whitespaces ignored)
//   is '-'
// * MULT if the character in input starting at i (with whitespaces ignored) is
//   '*'
// * DIV if the character in input starting at i (with whitespaces ignored) is
//   '/'.
// and with i set to index position just after the relevant character.
//
// For instance if input is "456   +   789" and i is 3, then PLUS is returned
// i is set to 7 (i.e., the index position just after '+' in the input.)
int get_op(char input[], int & i)
{
}


// Evaluates and returns x op y
int evaluate(int x, int op, int y)
{
}


int main()
{
    while (1)
    {
        char input[1024] = "";
        get_input(input);

        int i = 0;
        int x = get_int(input, i);
        int op = get_op(input, i);
        int y = get_int(input, i);

        int z = evaluate(x, op, y); // z is the result of evaluating the
                                    // expression
    }

    return 0;
}
```

PART 3. Your program must also allow entering integer values. Here are some examples:

```
>>> 1
1
>>> -3
-3
>>>        123
123
>>>    0
0
>>>  -42
-42
>>>
```

Here's the skeleton code:

```
// ... constants ...

// ... get_input ...
// ... get_int ...
// ... get_op
// ... evaluate ...

int main()
{
    while (1)
    {
        ...

        int z = evaluate(x, op, y); // z is the result of evaluating the
                                    // expression, i.e., z is x op y or
                                    // z is x
    }

    return 0;
}
```

PART 4. Now, we allow users of your program to store values in memory. The memory is modeled using an array of integers. The first integer in this array is called `%0`, the second integer in this array is called `%1`, ... Altogether 1024 integers can fit in this memory. Therefore, the last integer is `%1023`. Integer values can be stored in

this memory. Here are some examples:

```
>>> %0 = 42
```

This stores integer 42 into %0. You must also handle whitespaces. For instance, the following commands achieve the same goal:

```
>>>       %0 = 42
>>> %0     =42
>>> %0=     42
```

Note that when you store values into the memory, the value is not printed. You can print the value in the memory like this (see second command below):

```
>>> %0 = 42
>>> %0
42
```

Here's the skeleton code:

```
// ... constants
const MEMORY_SIZE = 1024;

// ... get_input ...
// ... get_int ...
// ... get_op

int main()
{
    int memory[MEMORY_SIZE] = {0};

    while (1)
    {
        ...
    }

    return 0;
}
```

PART 5. You can evaluate–and–store like this:

```
>>> %0 = 42 + 1
>>> %0
43
>>> %1 = -3 * -2
>>> %1
```

```
6
```

PART 6. You can also evaluate–and–store using memory like this:

```
>>> %0 = 42 + 1
>>> %0
43
>>> %1 = -3 * -2
>>> %1
6
>>> %2 = %0 + 1
>>> %2
44
>>> %3 = 100 - %1
>>> %3
94
>>> %4 = %0 / %1
>>> %4
7
```

Here's the skeleton code:

```
// ... constants
// ... get_input ...
// ... get_int ...
// ... get_op ...

int main()
{
    int memory[MEMORY_SIZE] = {0};

    while (1)
    {
        ...

        int i = 0;
        int x = get_int(input, i, memory);
        int op = get_op(input, i);
        int y = get_int(input, i, memory);

        ...
    }
```

```
    return 0;
}
```

You need not handle error checking for this program. (Although it would be a good exercise.) For instance, you need not handle inputs like this:

```
>>> 123 + + + 456
```

or

```
>>> 1    3     5
```

or

```
>>> 1 +
```

or

```
>>> %1 = %
```

or

```
>>> %99999
```

Even from this simple program you can see the the complexity grows as you allow more language features in your program. For instance, the program does not allow multiple operators. And allowing multiple operators in a single input will introduce the question of operator precedence.