**University of Michigan – Dearborn**

**CIS 200 – Computer Science II**

**Lab# 2**

**Professor Robert Mann**

Demetrius Johnson

[meech@umich.edu](mailto:meech@umich.edu)

September 22, 2020

**Table Content**

Contents

[Question 1 3](#_Toc51874701)

[Source Code –see Q1\_CIS-200-LAB2-DemetriusJohnson.cpp 3](#_Toc51874702)

[Screenshots 3](#_Toc51874703)

[Question 2 11](#_Toc51874704)

[Source Code –see Q2\_CIS-200-LAB2-DemetriusJohnson.cpp 11](#_Toc51874705)

[Screenshots 11](#_Toc51874706)

[Question 3 15](#_Toc51874707)

[Command line arguments in C/C++ 15](#_Toc51874708)

[Source Code –see Q3\_CIS-200-LAB2-DemetriusJohnson.cpp 16](#_Toc51874709)

[Screenshots 16](#_Toc51874710)

# Question 1

// LAB 2 - CIS 200

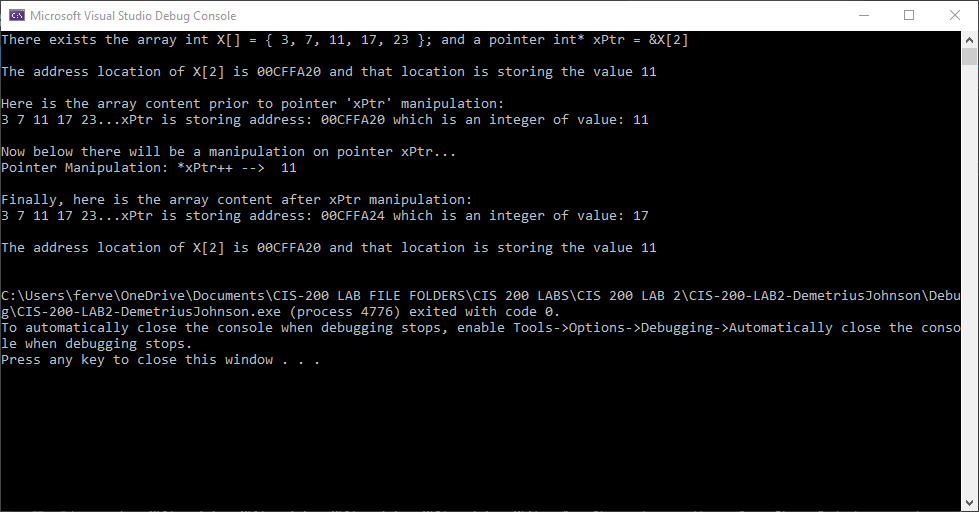
Question 1

-------------------------

## Source Code –see Q1\_CIS-200-LAB2-DemetriusJohnson.cpp

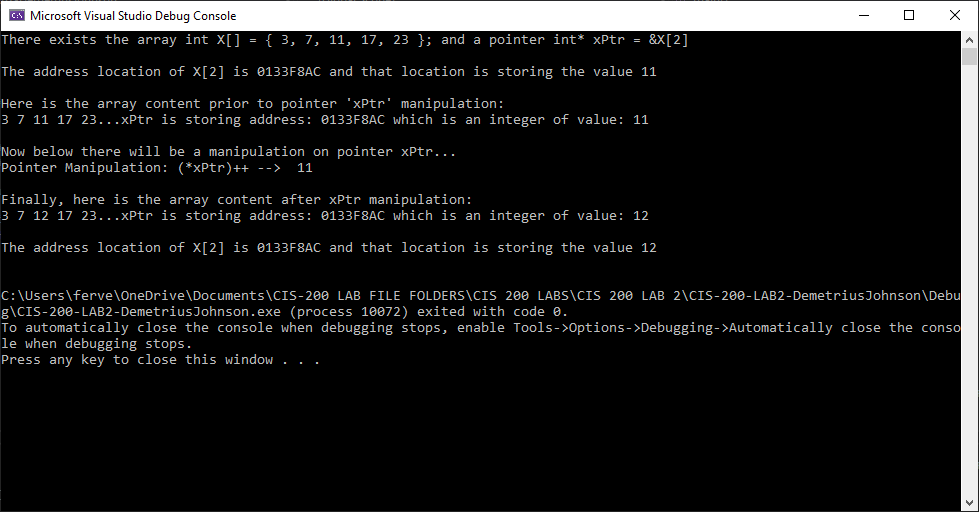
## Screenshots

TEST 1: \*xPtr++



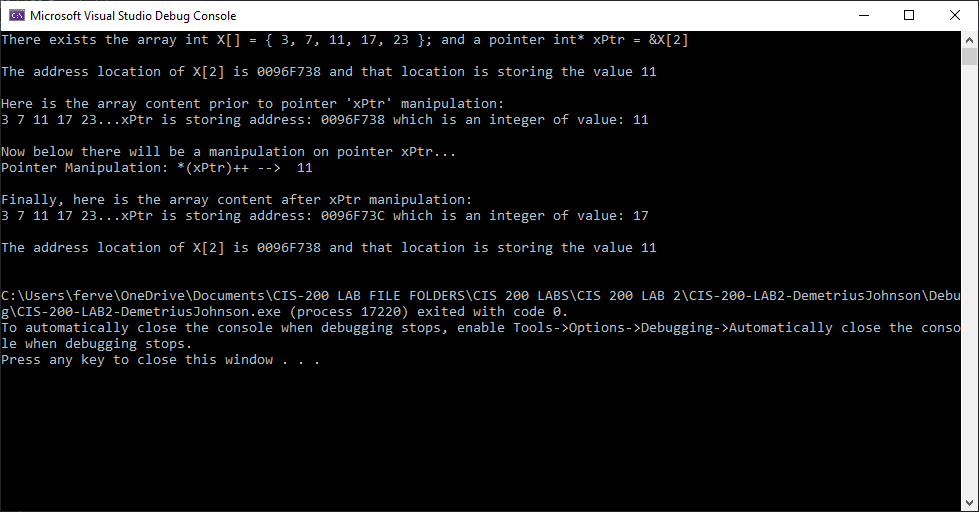
Results of test 1 – the manipulation causes the pointer to point to a new location, namely the location that is storing the 4th element (array position 3) of the array. This is because the ++ operator is performed before the deference operator ‘\*’. So, the address location of the pointer is incremented by a value equal to the next block size of its type – in this case type int. The array values are unchanged. In summary, this manipulation causes the pointer to point to the next element in an array.

TEST 2: (\*xPtr)++



Results of test 2 – the address in the pointer is unchanged, but the value stored at its location is changed; since the pointer references the location of element 2 in the array, that array value also changes. In summary: the pointer is first dereferenced (\*xPtr) so that the stored integer value pointed to by the pointer is incremented by 1 (++).

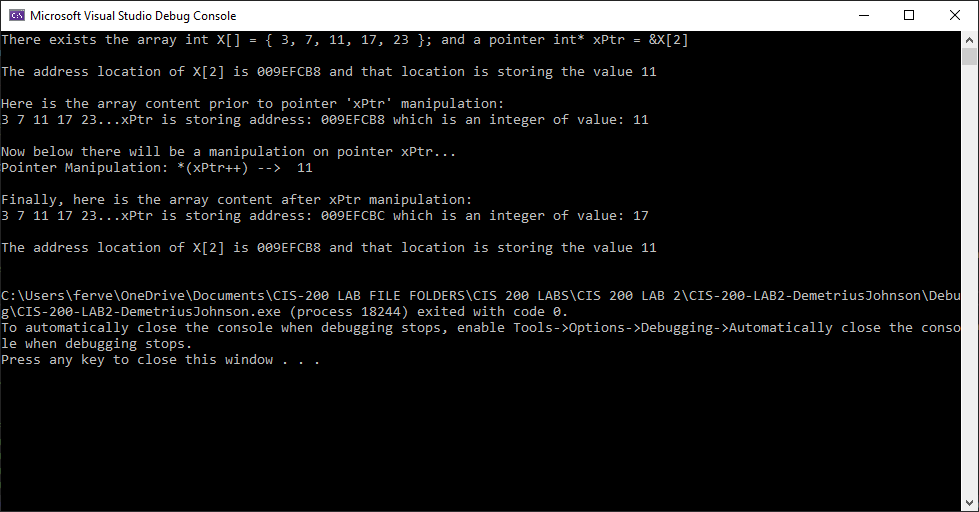
TEST 3: \*(xPtr)++



Results of test 3 – the values in the array are unchanged. The pointer points to a new location, namely, the location at element 3 (the 4th element of the array). In summary: this causes the pointer to point to the next element in the array. Essentially, \*(xPtr)++ is equal to the statement \*xPtr++.

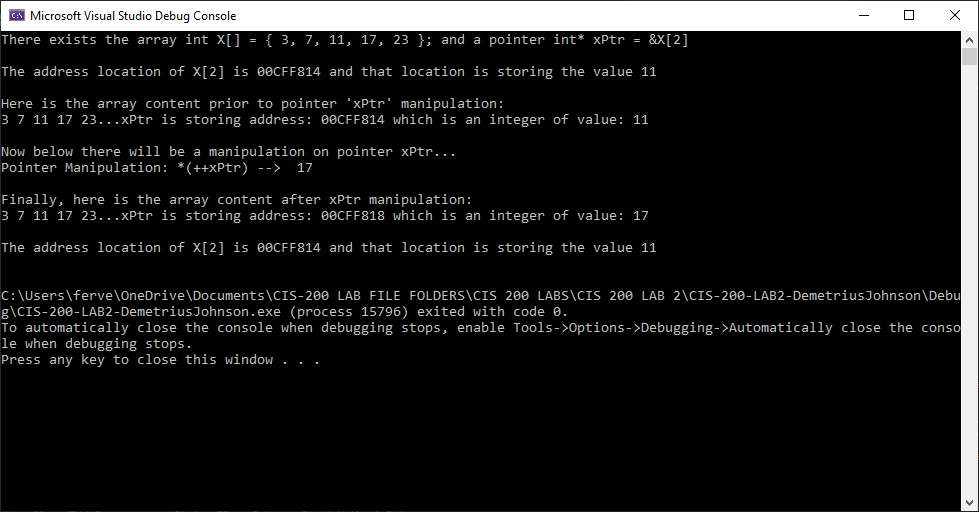
\*\*\* note: Postfix increment/decrement ( ++ / -- ) operators have the higher precedence than dereferencing ( \* ).

TEST 4: \*(xPtr++)



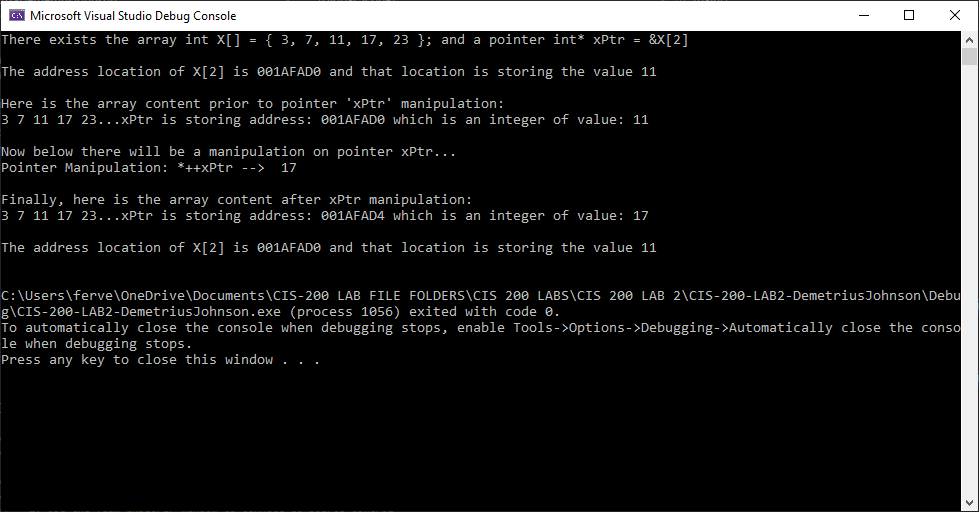
Results of test 4 – The array is unchanged. The address of type int stored inside of the pointer is incremented and thus points to the next address in the array storing the next element. In summary: this causes the pointer to point to the next element in the array. Essentially, \*(xPtr++) is also equal to the statement \*xPtr++.

TEST 5: \*(++xPtr)



Results of test 5 – the values in the array are unchanged. The pointer points to a new location, namely, the location at element 3 (the 4th element of the array). In summary: this causes the pointer to point to the next element in the array. Essentially, \*(++xPtr) is also equal to the statement \*xPtr++.

TEST 6: \*++xPtr



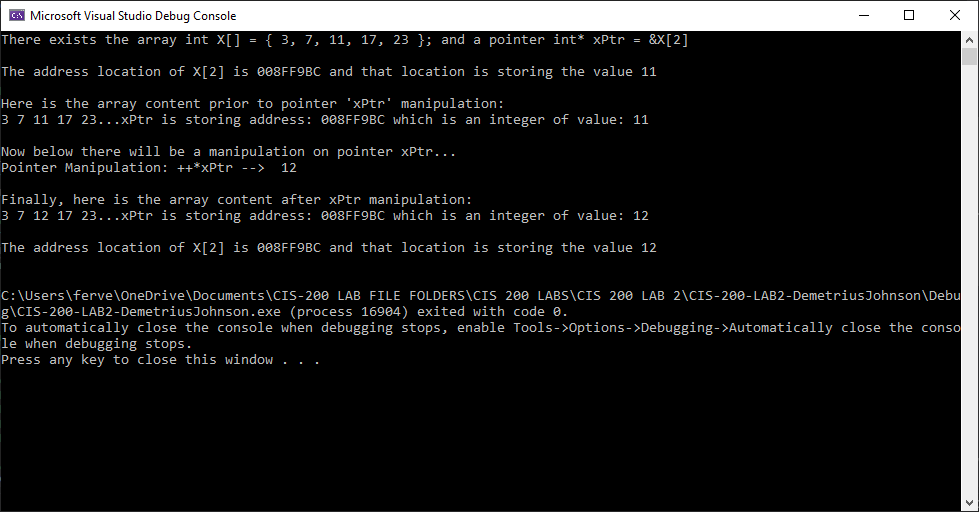
Results of test 6 – the values in the array are unchanged. The pointer points to a new location, namely, the location at element 3 (the 4th element of the array). In summary: this causes the pointer to point to the next element in the array. Essentially, \*++xPtr is equal to the statement \*xPtr++. I also notice the increment of the address is incremented by 4 and the address is represented by hexadecimal. I assume ++operator for an address of type Int causes the pointer to point to the next consecutive 2 bits of memory.

TEST 7: (\*++)xPtr

Compilation Error (code E0029): “expected an expression”…

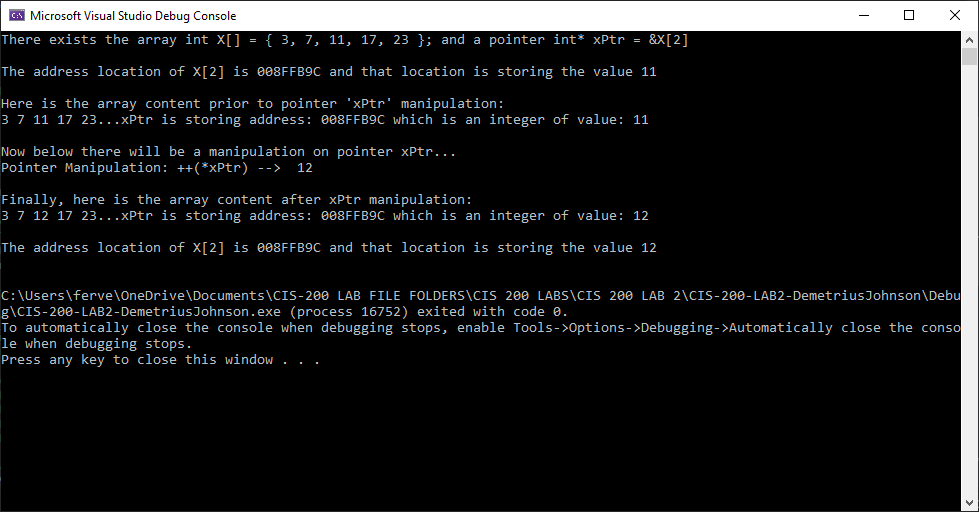
You cannot use operators inside of a parenthesis alone; the compiler sees the parenthesis and does order of operations; then it goes into the parenthesis to evaluate an expression and sees only ++operator and the \*operator and does not know what to increment or dereference. When you put an expression such as (xPtr) – a pointer variable – inside parentheses, the compiler can evaluate the expression as its variable type in the form of raw data (what is stored inside of the variable/its parameters), then when it is finished evaluating the variable it can see operators such as ++ or \* and then apply it to the raw data that was evaluated from the parenthesis.

TEST 8: ++\*xPtr



Results of test 8 – the address in the pointer is unchanged, but the value stored at its location is changed; since the pointer references the location of element 2 in the array, that array value also changes. In summary: the pointer is first dereferenced (\*xPtr) so that the stored integer value pointed to by the pointer is incremented by its equivalent of 1 (++).

TEST 9: ++ (\*xPtr)



Results of test 9 – similar to test 2 and test 8 the address in the pointer is unchanged, but the value stored at its location is changed; since the pointer references the location of element 2 in the array, that array value also changes. In summary: the pointer is first dereferenced (\*xPtr) so that the stored integer value pointed to by the pointer is incremented by its equivalent of 1 (++).

TEST 10: (++\*)xPtr

Compilation Error (code E0029): “expected an expression”…

Exactly the same case as test 7…

You cannot use operators inside of a parenthesis alone; the compiler sees the parenthesis and does order of operations; then it goes into the parenthesis to evaluate an expression and sees only ++operator and the \*operator and does not know what to increment or dereference. When you put an expression such as (xPtr) – a pointer variable – inside parentheses, the compiler can evaluate the expression as its variable type in the form of raw data (what is stored inside of the variable/its parameters), then when it is finished evaluating the variable it can see operators such as ++ or \* and then apply it to the raw data that was evaluated from the parenthesis.

# Question 2

// LAB 2 - CIS 200

Question 2

-------------------------

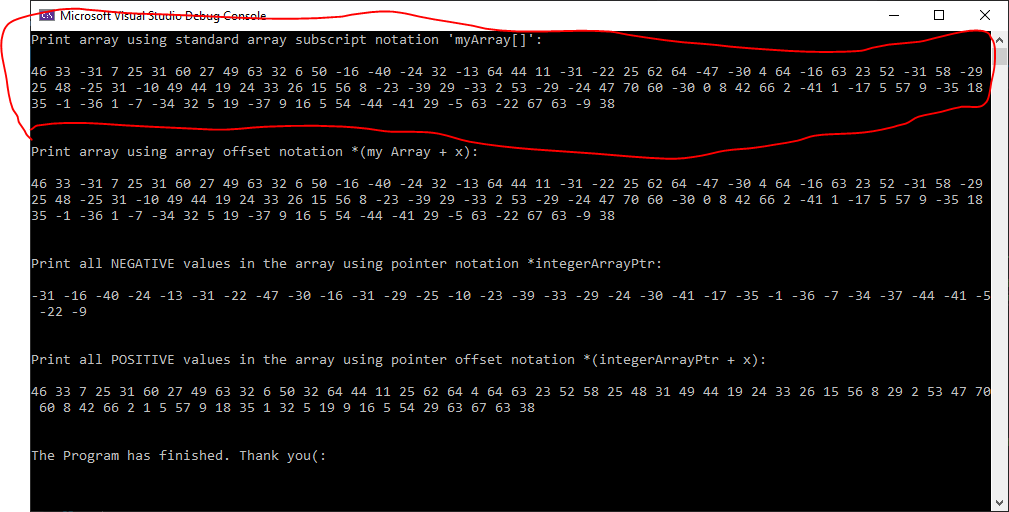
## Source Code –see Q2\_CIS-200-LAB2-DemetriusJohnson.cpp

## Screenshots

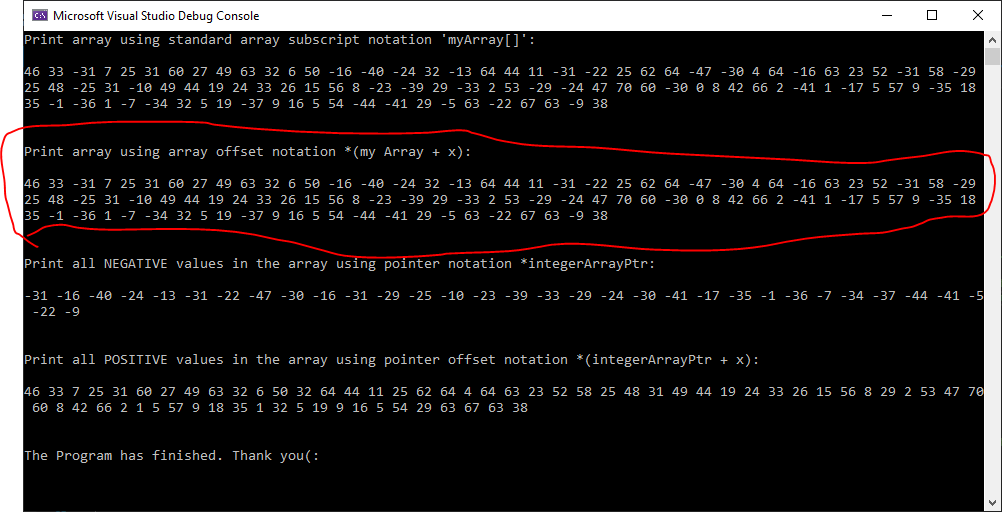
Test Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test # | Valid / Invalid Data | Description of test | Actual Output | Test Pass / Fail |
| 1 | valid | Print array using standard array subscript notation 'myArray[]': | See screenshot | pass |
| 2 | valid | Print array using array offset notation \*(my Array + x): | See screenshot | pass |
| 3 | valid | Print all NEGATIVE values in the array using pointer notation \*integerArrayPtr: | See screenshot | pass |
| 4 | valid | Print all POSITIVE values in the array using pointer offset notation \*(integerArrayPtr + x): | See screenshot | pass |
|  |  |  |  |  |
|  |  |  |  |  |

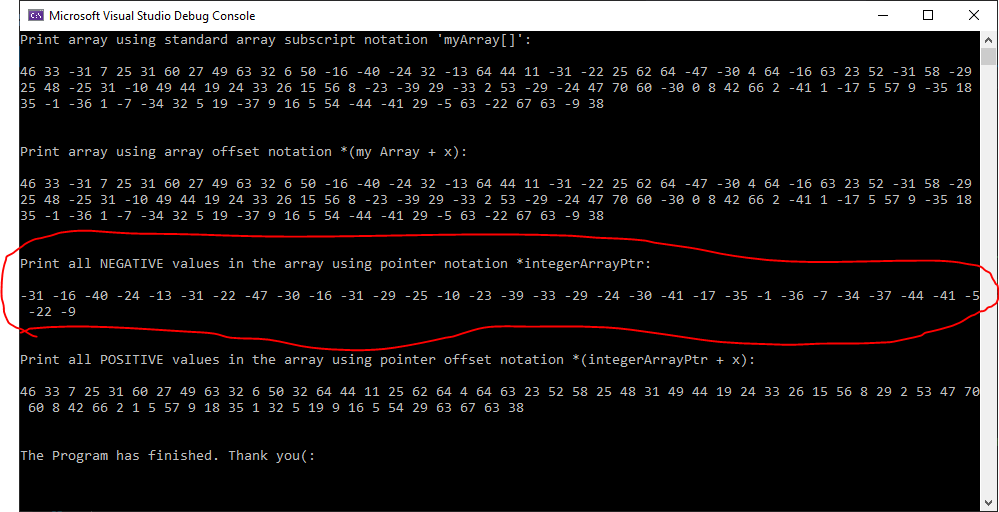
TEST 1



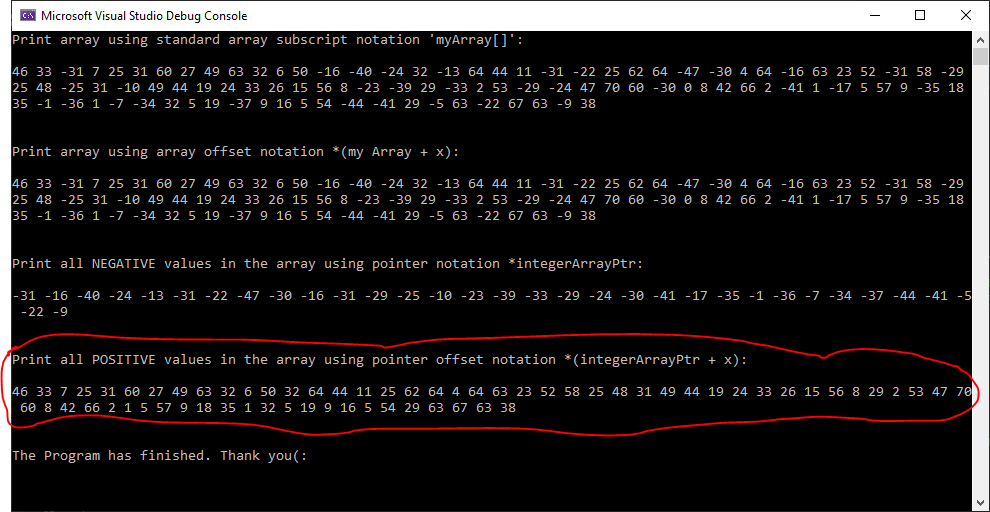
TEST 2



TEST 3



TEST 4



# Question 3

// LAB 2 - CIS 200

Question 3

***Notes:***

The **main function** can be defined with no **parameters** or with two **parameters** (for passing command-line **arguments** to a program when it begins executing). The two **parameters** are referred to here as argc and argv, though any names can be used because they are local to the **function** in which they are declared. ([www.itec.suny.edu](http://www.itec.suny.edu)).

Command line arguments in C/C++

Last Updated: 21-12-2018 (<https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/#:~:text=argv(ARGument%20Vector)%20is%20array,element%20is%20command%20%2Dline%20arguments>.)

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

int main() { /\* ... \*/ }

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.  
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

int main(int argc, char \*argv[]) { /\* ... \*/ }

or

int main(int argc, char \*\*argv) { /\* ... \*/ }

* **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
* The value of argc should be non negative.
* **argv(ARGument Vector)** is array of character pointers listing all the arguments.
* If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
* argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.
* argc: The number of arguments in the command line that invoked the program. The value of *argc* is nonnegative.
* argv: Pointer to an array of character strings that contain the arguments, one per string. The value argv[*argc*] is a null pointer.

## Source Code –see Q3\_CIS-200-LAB2-DemetriusJohnson.cpp

## Screenshots

Test Table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test # | Valid / Invalid Data | Description of test | Input Value | Expected Output | Actual Output | Test Pass / Fail |
| 1 | valid | Run program and see if it displays number of arguments and the filename and path of user .EXE file for the program | NA | NA | See screenshot | pass |

TEST 1

