CS 100 Lab 11 | Fall 2025 | 10 Nov 2025

CS100 Computational Problem Solving

Instructor (Section1): Shafay Shamail

Lab# 11:

Pointers and 2D Arrays

Lab Guidelines

1. You are allowed to perform/submit the lab only during the lab timings.
2. Make sure you do not leave the lab without submitting your work on LMS.
3. Put all cpp files into a folder **YourRollNo_Lab11_TAname** and submit it on LMS (Tests and Quizzes -> Lab 11).
4. Note that the submission tab closes at the same time when the lab ends.
5. Please make sure that you have gone through the Lab Manual before starting the lab.
6. Talking to each other is NOT permitted. If you have a question, ask the lab assistants.
7. The objective is not simply to get the job done, but to get it done in the way that is asked for in the lab.
8. Copying/sharing code is strictly prohibited. Using any unfair means will lead to immediate disqualification.
9. Any cheating case will be reported to Disciplinary Committee without any delay.

Programming Conventions

(See: C++ Language Coding Guidelines)

1. Use tabs instead of spaces.
2. Variable names are lowercase.
3. Constant names are uppercase.
4. Function names are first word small, all other words camel case. For example, thisIsAFunction().
5. Class names start with an uppercase letter, and all following words are camel case. For example, ThisIsAClass.
6. There are spaces after reserved words and between binary operators.
7. Braces must be vertically aligned.
8. No magic numbers may be used.
9. Every function must have a comment.
10. At most 30 lines of code may be used per function.
11. Global variables are not allowed unless specifically instructed.
12. Use of *goto* statement is not allowed.

Lab10 Marks Distribution

Task 1	Task 2	Task 3			Total
30	30	40			100

Move to the next page to view the lab tasks.

Let’s Begin …

Lab Task Instructions:

1. VS code installation on MAC/Windows
2. Install C++ extension on VS code
3. Install Coderunner extension on VS code
4. Save an empty file with the extension ".cpp" to start your lab task
5. For each task create a separate file.

Task 1: Array Reversal (30 marks)

In this task, you will write a program that reverses a specific portion of an integer array in place using pointer arithmetic and functions.

You **must not** use the indexing operator ([]) anywhere in your program, except the one time the array is declared in the main function.

Your solution should demonstrate:

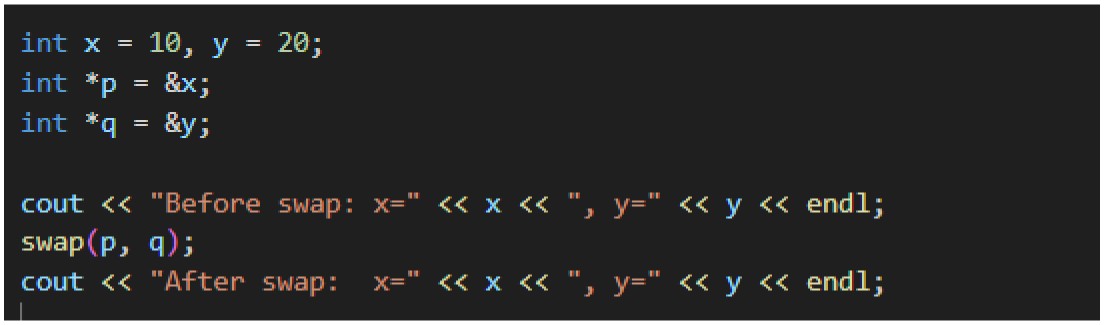
- how to pass pointers to functions,
- how to manipulate array segments through pointer offsets, and
- how to validate user input before performing operations.

Description:

You are required to implement three functions:

1. **`void swap(int *a, int *b)`** **(5 marks)**

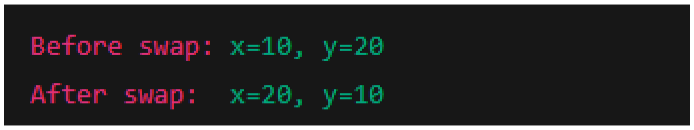
   - This function receives two integer pointers as parameters.
   - It swaps the **values stored at the addresses** they point to, **not** the pointers themselves.
   - **Example usage:**



```
int x = 10, y = 20;
int *p = &x;
int *q = &y;

cout << "Before swap: x=" << x << ", y=" << y << endl;
swap(p, q);
cout << "After swap:  x=" << x << ", y=" << y << endl;
```

Expected output:

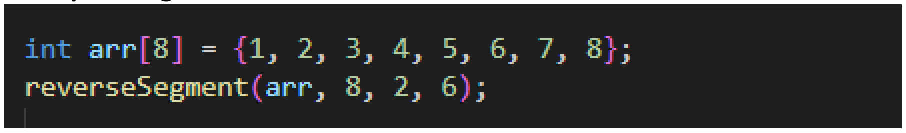


```
Before swap: x=10, y=20
After swap:  x=20, y=10
```

2. `void reverseSegment(int *arr, int size, int start, int end)`

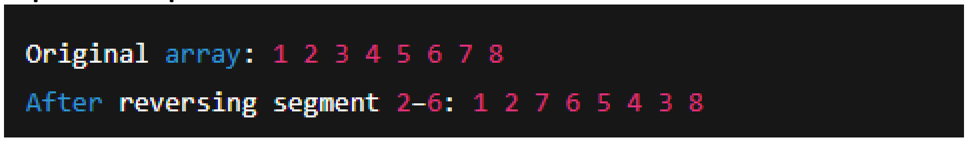
Reverses the portion of the array between the indices start and end **in place**, using the **swap(a, b)** function and pointer arithmetic. The rest of the array remains unchanged. **`(10 marks)`**

Example Usage:



```
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
reverseSegment(arr, 8, 2, 6);
```

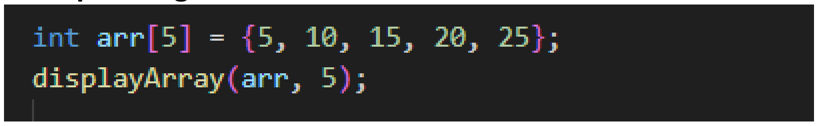
Expected Output:



```
Original array: 1 2 3 4 5 6 7 8
After reversing segment 2-6: 1 2 7 6 5 4 3 8
```

3. `void displayArray(int* arr, int size)` **`(5 marks)`**

- Displays all elements of the array in a single line, separated by spaces.
- The array should be accessed using **pointer arithmetic only.**
- **Example Usage:**



```
int arr[5] = {5, 10, 15, 20, 25};
displayArray(arr, 5);
```

Expected outcome:

Main Function Requirements

In your **`main()`** function: **`(10 marks)`**

1. Ask the user to enter the **size** of the array (must be positive and ≤ 20).

2. **Declare** the array normally using square brackets (e.g. **`int arr[20];`**).

   Declaring the array with [] is allowed and required.
   Once declared, you must **not** use indexing (**`arr[i]`**) to access or modify elements.
   All access and traversal must use pointer arithmetic, for example:

```
cin >> *(arr + i);

cout << *(arr + i);
```

3. Input all elements of the array using pointer arithmetic.

4. Ask the user to enter the **start** and **end** indices of the segment to reverse.

5. **Validate inputs:**
   Validate all the inputs (`size, start, end`). If any validation fails, display an appropriate message **following this format**:

   ```
   Invalid input. <VariableName> is out of bounds.

   Please try again.
   ```

   and continue taking the input until the user enters a valid one.

6. Display the **original array** using **`displayArray()`**.

7. Call **`swap()`** for reversing the chosen segment in **`reverseSegment().`**

8. Display the **array after reversal** using **`displayArray()`**.

Sample Output 1:

```
Enter the size of the array: 8
Enter 8 integers: 1 2 3 4 5 6 7 8
Enter start and end indices to reverse: 2 6
Original array: 1 2 3 4 5 6 7 8
Array after reversing segment 2 to 6: 1 2 7 6 5 4 3 8
```

Sample Output 2:

```
Enter the size of the array: 6
Enter 6 integers: 5 10 15 20 25 30
Enter start and end indices to reverse: 4 2
Invalid input. Start index is out of bounds.
Please try again.
Enter start and end indices to reverse: 1 3
Original array: 5 10 15 20 25 30
Array after reversing segment 1 to 3: 5 20 15 10 25 30
```

Constraints

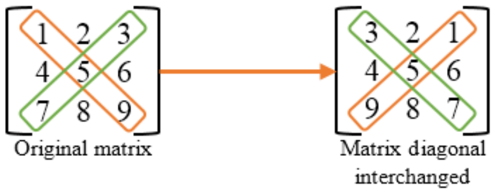
- No indexing operators ([]) for accessing elements.
- Only pointer arithmetic (*(arr + i) or pointer increments).
- No global variables.
- Static arrays only.

- Proper indentation, naming, and documentation required.

Task 2: Swap Diagonals (30 marks)

You will write a program that swaps the **left (main)** and **right (secondary)** diagonals of a square matrix of size n × n. The maximum matrix size is 5 × 5.

You must use **only pointer arithmetic** to access elements. You are **not allowed** to use array indexing like matrix[i][j] inside your functions.



Note*: In this task, you will see the following declaration in the code we give you:*

int matrix[5][5];

This does ***not*** *mean the matrix is always 5×5 in our logic.*

(matrix[5][5] simply reserves space in memory for 25 integers, it is just the ***maximum*** *size we might ever need. The actual matrix size n × n is chosen by the user at runtime, where n is between 2 and 5. For any given input, you should treat only the first n rows and the first n columns of this array as your real matrix. In other words, if the user enters n = 3, you are effectively working with a 3 × 3 matrix stored in the top-left part of matrix[5][5], and you ignore the remaining unused elements.)*

You will write three functions:

1. **readMatrix(int *base, int n)**
   - This function will read n × n integers from the user.
   - The parameter base will be a pointer to the first element of the matrix (i.e., the address of matrix[0][0]).
   - Inside this function, you will use pointer arithmetic to store values in the matrix.
   - Use pointer arithmetic to access array elements.
2. **printMatrix(int *base, int n)**
   - This function will print the matrix in a formatted way.

- Use pointer arithmetic to access array elements.

3. **swapDiagonals(int *base, int n)**

    - This function will swap the elements on the **left diagonal** with the elements on the **right diagonal** of the n × n matrix.

    - You are not allowed to use indexing; all access must be via pointer expressions.

In your main() function, you will control the flow of the program and repeatedly handle matrices until the user chooses to exit.(look at sample outputs below for a better understanding).

1. Repeatedly ask the user to enter the size n of the matrix.

    a. If the user enters a **negative** value for n, the program should **exit**.

2. If the user enters a value of n that is **less than 2** or **greater than 5**, you must:

    a. Display a message telling them the size is invalid and that it must be between 2 and 5.

    b. Ask them to enter n again.

    c. If they enter a negative value at this point, you should also exit the program.

3. Create a pointer to the first element of the matrix:

    a. Set a pointer (for example named base) to point to the first element of the array, i.e., to matrix[0][0].

    b. This pointer will be passed to all three functions.

    c. They will use this as the base address to access the entire n × n matrix using pointer arithmetic.

4. Call the function to read the matrix:

    a. Call readMatrix(base, n) to input all n × n elements from the user.

5. Print the original matrix:

    a. Display a suitable message like “Original matrix:”

   b. Call printMatrix(base, n) to print the matrix before any changes.

6. Swap the diagonals:

   a. Call swapDiagonals(base, n) to interchange the left and right diagonals of the matrix.

7. Print the updated matrix:

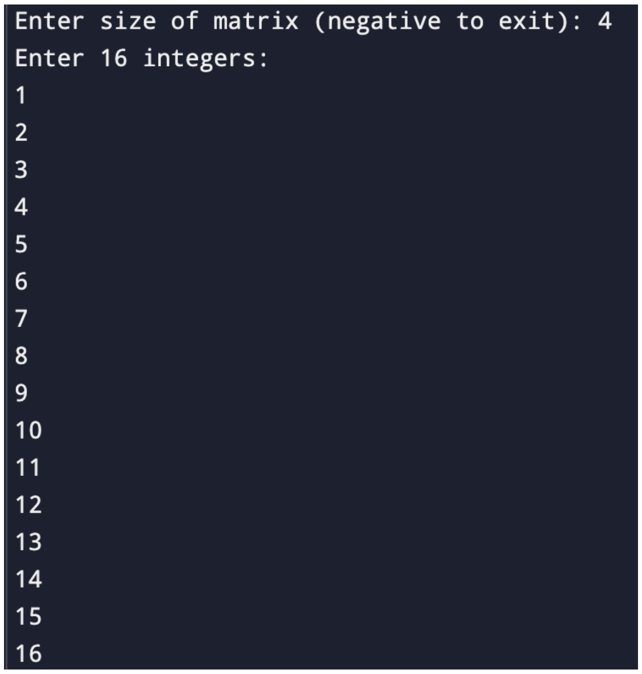
   a. Display a suitable message like "Matrix after swapping diagonals:"

   b. Call printMatrix(base, n) again to show the matrix after the diagonal swap.

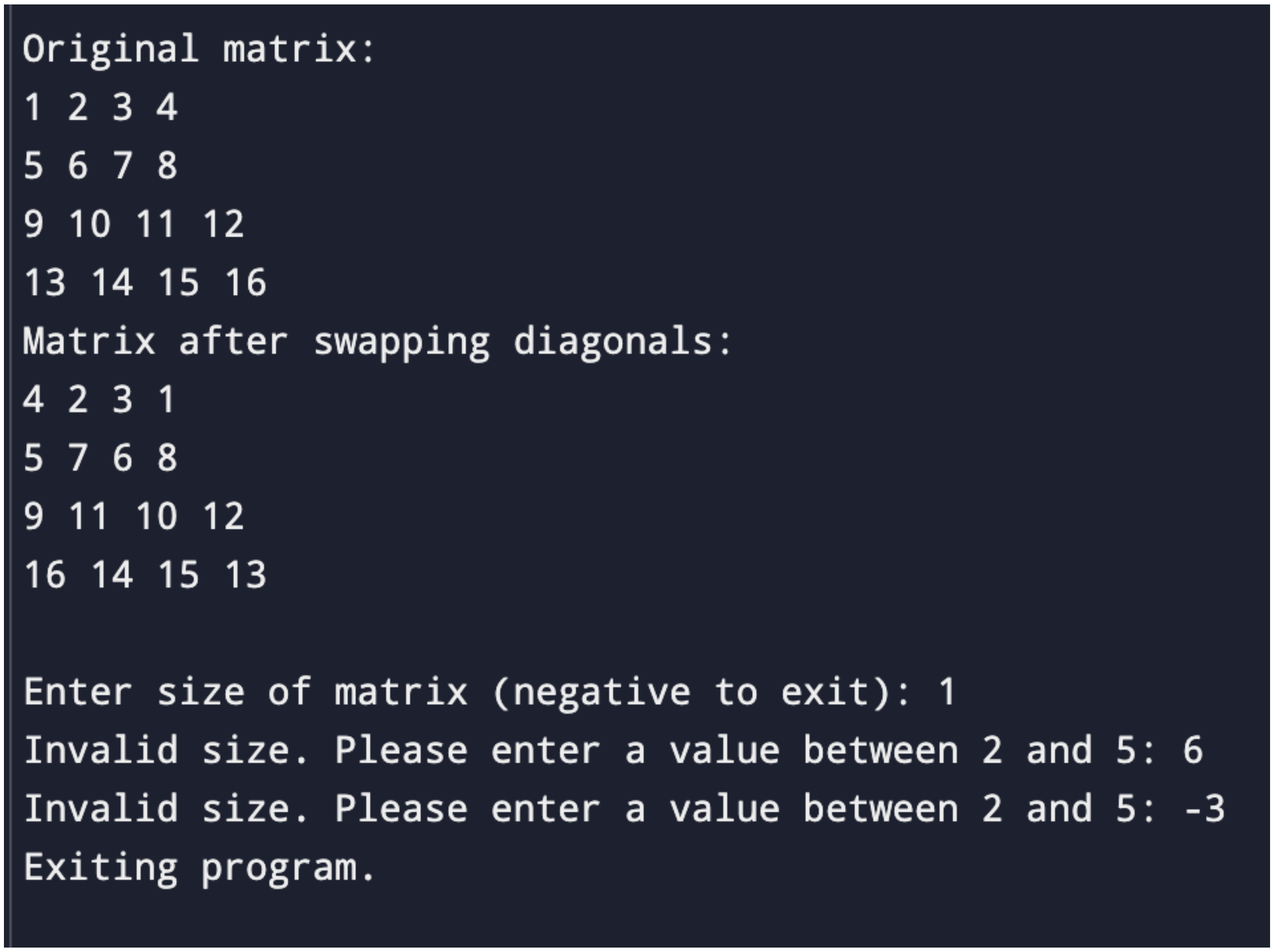
8. After finishing one matrix, go back to step 1:

   a. Ask the user again for a new size n.

   b. Repeat the process until they enter a negative value.



```
Enter size of matrix (negative to exit): 4
Enter 16 integers:
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```



```
Original matrix:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
Matrix after swapping diagonals:
4 2 3 1
5 7 6 8
9 11 10 12
16 14 15 13

Enter size of matrix (negative to exit): 1
Invalid size. Please enter a value between 2 and 5: 6
Invalid size. Please enter a value between 2 and 5: -3
Exiting program.
```

Task 3: Word Processing (40 marks)

You will write a program that reads multiple lines of text, stores them inside a single character buffer, and then prints all the lines in **reverse order** using an array of pointers. The goal is to practice working with:

- a char buffer[1000] that holds all the text, and
- a char *lines[100] array, where each element points to the beginning of one line in the buffer,
- **using only pointer arithmetic** (no array indexing while traversing).

You are **not** allowed to use array indexing like buffer[i] or lines[i] when moving through the buffer or the lines array. You must use pointer arithmetic instead.

You are given:

- a character buffer:
    - char buffer[1000];
      This reserves space for up to 1,000 characters. All lines of text will be stored **one after another** in this buffer.
- an array of pointers:
    - char *lines[100];
      This can store up to 100 pointers. Each pointer in this array will point to the **first character of a line** inside the buffer.

Overview of how the text is laid out in memory

Conceptually, after reading a few lines, your buffer might look like this in memory (with \0 marking the end of each line):

```
Line1\0Line2\0Line3\0...
```

and:

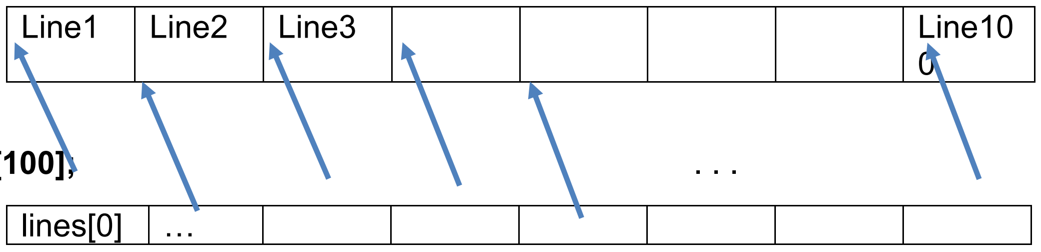
- lines[0] will point to the 'L' of "Line1" inside buffer,
- lines[1] will point to the 'L' of "Line2",

- lines[2] will point to the 'L' of "Line3", and so on.

When you later print lines in reverse order, you will start from the last stored pointer in lines and move backward.

char buffer[1000]:

char *lines[100];



Functions you will write

1. char* insertLine(char *buff, string line)

This function is responsible for inserting one line of text into the buffer.

- The parameter buff is a pointer that indicates **where in the buffer** the new line should be stored. It points to the first free position inside the buffer.

- The parameter line is a C++ string that contains the text of one input line (without the final newline character).

Inside this function, you will:

1. Copy all characters from the line into the character buffer starting at the address buff. You must move through the buffer using pointer arithmetic

2. After copying the characters of the line, write a '\0' character at the end. This marks the end of that line inside the buffer.

3. Return a pointer to the position **just after** this '\0'. This returned pointer tells the caller where the **next line** should start in the buffer.

2. void printLine(char *ptr)

This function is responsible for printing a single line of text.

- The parameter ptr points to the **first character** of a line stored inside the buffer.

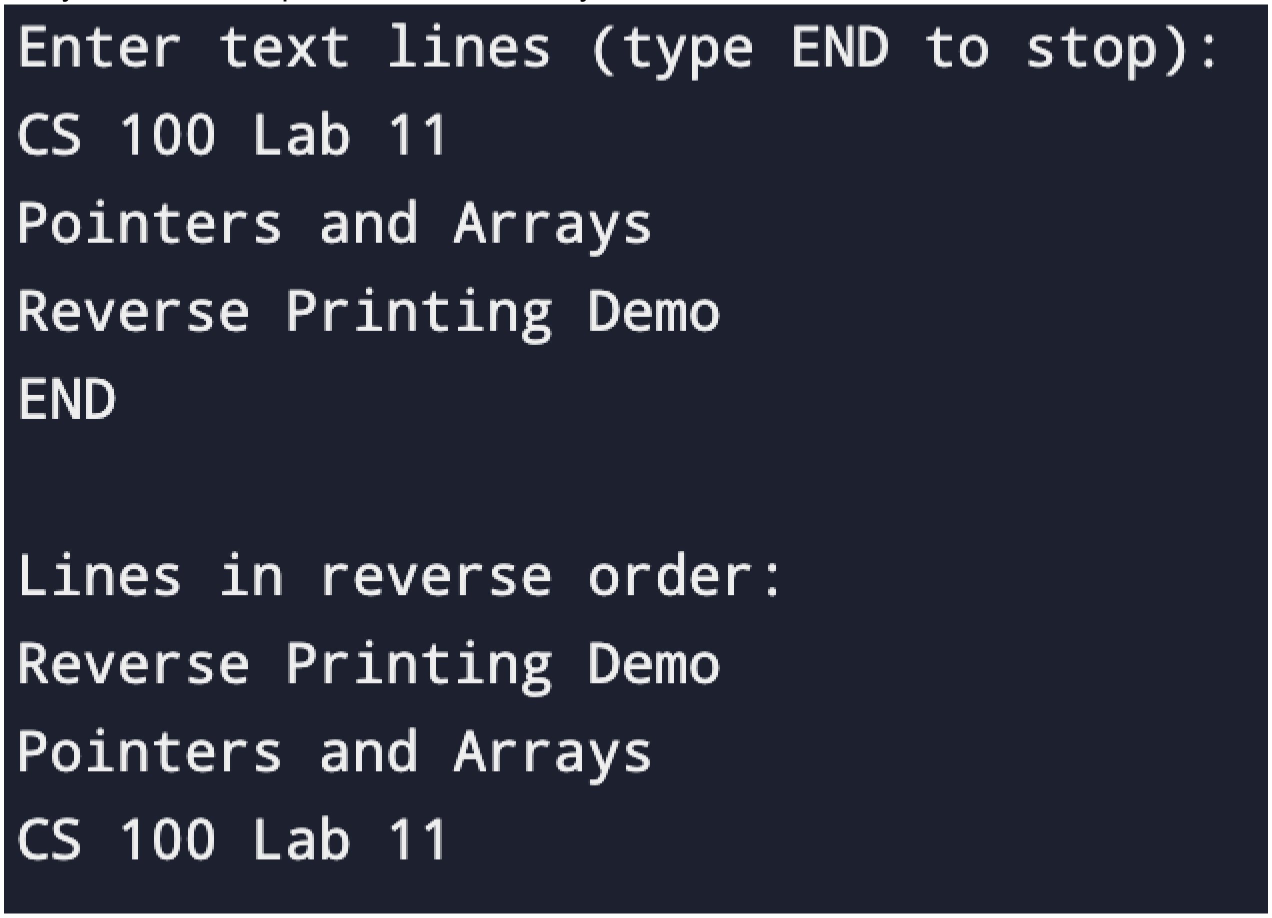
Inside this function, you will:

1. Start from ptr and print characters one by one.
2. Continue printing until you reach the '\0' terminator (which marks the end of that line).
3. When you reach '\0', stop printing characters and print a newline character ('\n') so that the next line appears on a new line on the screen.
4. Move from one character to the next using pointer arithmetic

This function prints exactly one line, given a pointer to its starting position.

Write a main function to test your functionality

You are not allowed to use array indexing while moving through the **buffer** and **lines** array. You can use pointer arithmetic only.



```
Enter text lines (type END to stop):
CS 100 Lab 11
Pointers and Arrays
Reverse Printing Demo
END


Lines in reverse order:
Reverse Printing Demo
Pointers and Arrays
CS 100 Lab 11
```

End of Lab 11