Data Structures Lab 2

Course: Data Structures (CL2001) **Semester:** Fall 2022
Instructor: Muhammad Monis **T. A:** N/A

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Lab Title: Learn to implement a Dynamic Safe Array with one/two dimensional pointers.
Objectives: Get the knowledge of how dynamic memory is used to implement 1D and 2D arrays which are more powerful as compared to the default array mechanism of the programming language C/C++ supports.
Tool: Dev C++ (You can also use any editor of your choice)

1D & 2D Array:

A **one-dimensional** array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

Syntax:

```
char name[5];
int mark[5] = {5,11,14,65,85};
int mark[] = {5,11,14,65,85};
```

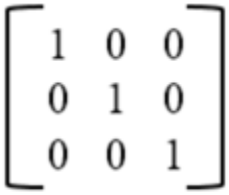
Like a 1D array, a **2D array** is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.

Syntax:

```
float x[3][4];
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Task # 1

Write a C++ program to read elements in a matrix and check whether the matrix is an Identity matrix or not.



Identity matrix

Dynamic Memory Allocation for arrays:

Memory in your C++ program is divided into two parts

1. The **`stack -`** All variables declared inside the function will take up memory from the stack.
2. The **`heap -`** this is unused memory of the program and can be used to allocate the memory dynamically when the program runs.

A **dynamic array** is an array with a big improvement: **automatic resizing**.
One limitation of arrays is that they're fixed size, meaning you need to specify the number of elements your array will hold ahead of time.
A dynamic array expands as you add more elements. So you don't need to determine the size ahead of time.

Strengths:

1. **Fast lookups**. Just like arrays, retrieving the element at a given index takes O (1) time.
2. **Variable size.** You can add as many items as you want, and the dynamic array will expand to hold them.
3. **Cache-friendly.** Just like arrays, dynamic arrays place items right next to each other in memory, making efficient use of caches.

Weaknesses:

1. **Slow worst-case appends.** Usually, adding a new element at the end of the dynamic array takes **O (1)** time. But if the dynamic array doesn't have any room for the new item, it'll need to expand, which takes **O(n)** time.
2. **Costly inserts and deletes.** Just like arrays, elements are stored adjacent to each other. So adding or removing an item in the middle of the array requires "scooting over" other elements, which takes **O(n)** time.

Factors impacting performance of Dynamic Arrays:

The array's initial size and its growth factor determine its performance. Note the following points:

1. If an array has a **small size** and a **small growth factor**, it will keep on **reallocating** memory more often. This will **reduce** the performance of the array.
2. If an array has a **large size** and a **large growth factor**, it will have a **huge chunk** of **unused** memory. Due to this, resize operations may take longer. This will reduce the performance of the array.

The new Keyword:

In C++, we can create a dynamic array using the **new keyword**. The number of items to be allocated is specified within a pair of square brackets. The type name should precede this. The requested number of items will be allocated.

Syntax:

```
int *ptr1 = new int;
int *ptr1 = new int[5];
int *array { new int[10]{}};
int *array { new int[10]{1,2,3,4,5,6,7,8,9,10}};
```

Resizing Arrays:

The length of a dynamic array is set during the allocation time. However, C++ doesn't have a built-in mechanism of resizing an array once it has been allocated. You can, however, overcome this challenge by allocating a new array dynamically, copying over the elements, then erasing the old array.

Dynamically Deleting Arrays:

A dynamic array should be deleted from the computer memory once its purpose is fulfilled. The delete statement can help you accomplish this. The released memory space can then be used to hold another set of data. However, even if you do not delete the dynamic array from the computer memory, it will be deleted automatically once the program terminates.

Syntax:

```
delete ptr;
delete[] array;
```

NOTE: To delete a dynamic array from the computer memory, you should use delete[], instead of delete. The [] instructs the CPU to delete multiple variables rather than one variable. The use of delete instead of delete[] when dealing with a dynamic array may result in problems. Examples of such problems include **memory leaks, data corruption, crashes,** etc.

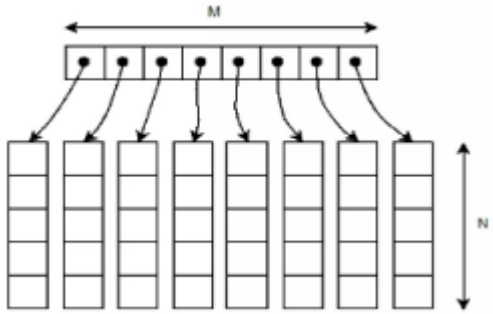
Example:

Single Dimensional Array:

```
#include <iostream>
using namespace std;
main(){
    int* darray = new int[3] {1,2,3}; //Initializing a dynamic array
    cout << *darray+1 << endl;
    cout << darray[2];
    delete[] darray; //Deleting the dynamic array to save memory space
    //cout <<darray[2] << endl; If we try to print the array we would get random values
}
```

Two Dimensional Array Using Array of Pointers:

We can dynamically create an array of pointers of size M and then dynamically allocate memory of size N for each row as shown below.



Example:

```
// Dynamically Allocate Memory for 2D Array in C++
int main(){
	int** A = new int*[M]; // dynamically create array of pointers of size M
	srand (time(NULL)); /* initialize random seed: */
	for (int i = 0; i < M; i++)	// dynamically allocate memory of size N for each row
		A[i] = new int[N]; // assign values to allocated memory
		cout << A[i]; // print the 2D array


	for (int i = 0; i < M; i++) // deallocate memory using delete[] operator
		delete[] A[i];
	delete[] A;
	return 0;
}
```

Task # 2

Write a program that will read **10 integers** from the keyboard and place them in an **array**. The program then will sort the array into **ascending** and **descending** order and print the sorted list. The program must not change the original array or not create any other integer arrays.

Task # 3

Write a C++ program to **rearrange** a given **sorted** array of **positive** integers. Note: In the final array, **first element** should be **maximum** value, **second minimum** value, **third second maximum** value, **fourth second minimum** value, **fifth third maximum** and so on.

Safe Array:

In C++, there is **no check** to determine whether the **array index** is **out of bounds**.
During program execution, an out-of-bound array index can cause **serious problems**. Also, recall that in C++ the array index starts at 0.
Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative.
"Safely" in this context would mean that access to the array elements must not be **out of range**. i.e. the position of the element must be **validated** prior to access.
For example, in the member function to allow the user to set a value of the array at a particular location:

```
void set(int pos, Element val){      //set method
        if (pos<0 || pos>=size){     //this line can also be written as (pos<0 or pos>=size)
                cout<<"Boundary Error\n";
        }
        else{
                Array[pos] = val;
        }
}
```

Task # 4

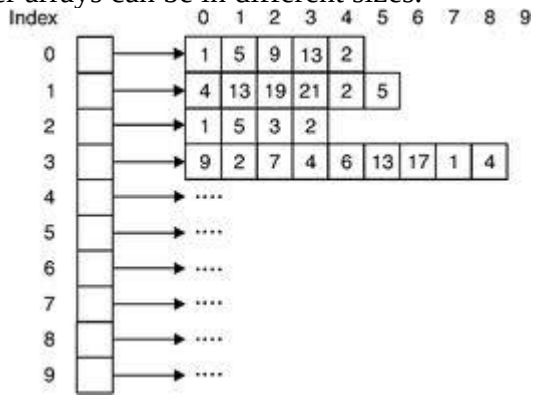
Write a program that creates a **2D array** of **5x5** values of **type Boolean**. Suppose indices represent people and the value at **row i**, **column j** of a **2D array** is true just in case **i and j** are **friends** and **false otherwise**. Use initializer list to instantiate and initialize your array to represent the following configuration: (*** means "friends"**)

	0	1	2	3	4
0		*		*	*
1	*		*		*
2		*			
3	*				*
4	*	*		*	

Write a method to check whether **two people** have a **common friend**. For example, in the example above, **0 and 4** are **both friends with 3** (so they have a common friend), whereas **1 and 2** have **no common friends**.

Jagged Array:

Jagged array is similar to an array but the difference is that its **an array of arrays** in which the member arrays can be in different sizes.



Example:

```
int *arr = new int*[3];
int Size[3];
int i,j,k;
for(i=0;i<3;i++){
   cout<<"Row "<<i+1<< " size: ";
   cin>>Size[i];
   arr[i] =new int[Size[i]];
}
for(i=0;i<3;i++){
   for(j=0;j<Size[i];j++){
   cout<<"Enter row " <<i+1<<" elements: ";
   cin>>*(*(arr + i) + j);
  }
}
// print the array elements using loops
// deallocate memory using delete[] operator as mentioned in the previous example
```

Task # 5

Write a program to calculate the GPA of students of all subjects of a single semester. Assume all the courses have the same credit hour (let's assume 3 credit hours). Do this task with Jagged Array?

	Data Structure	**Programming for AI**	**Digital Logic Design**	**Probability & Statistics**	**Finance & Accounting**
Ali	3.66	3.08	4.0	3.09	2.45
Hiba	3.33	3.0	3.66	3.0	---
Asma	4.0	---	2.66	---	3.45
Zain	2.66	2.33	4.0	---	---
Faisal	3.33	3.66	4.0	3.0	3.33