Data Structures Lab

Session 01

Course: Data Structures (CL2001) **Semester:** Fall 2022
Instructor: Mafaza Mohi **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 DynamicSafeArray with one/two dimensional pointers, Rule of Three, and Dynamic Object Allocation.

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++

Dynamic Allocation of Objects:

[Dynamically allocate memory to 5 objects of a class.]

Just like basic types, objects can be allocated dynamically, as well.
But remember, when an object is created, the constructor runs. Default constructor is invoked unless parameters are added:

```
Fraction * fp1, * fp2, * flist;
fp1 = new Fraction; // uses default constructor
fp2 = new Fraction(3,5); // uses constructor with two parameters
flist = new Fraction[20]; // dynamic array of 20 Fraction objects
// default constructor used on each
```

Deallocation with delete works the same as for basic types:

```
delete fp1;
delete fp2;
delete [] flist;
```

Notation: dot-operator vs. arrow-operator:

dot-operator requires an object name (or effective name) on the left side

```
objectName.memberName // member can be data or function
```

The arrow operator works similarly as with structures.

```
pointerToObject->memberName
```

Remember that if you have a pointer to an object, the pointer name would have to be dereferenced first, to use the dot-operator:
(*fp1).Show();

Arrow operator is a nice shortcut, avoiding the use or parentheses to force order of operations:
fp1->Show(); // equivalent to (*fp1).Show();

When using dynamic allocation of objects, we use pointers, both to a single object and to arrays of objects. Here's a good rule of thumb:
For pointers to single objects, the arrow operator is easiest:
fp1->Show();
fp2->GetNumerator();
fp2->Input();
For dynamically allocated arrays of objects, the pointer acts as the array name, but the object "names"; can be reached with the bracket operator. Arrow operator usually not needed:
flist[3].Show();
flist[5].GetNumerator();
// note that this would be INCORRECT, flist[2] is an object, not a pointer, therefore flist[2]->Show() is incorrect;

Task # 1
Create an Animal class (having appropriate attributes and functions) and do following steps.
Dynamically allocate memory to 5 objects of a class.
Order data in allocated memories by Animal names in ascending order.

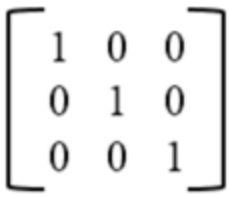
1D & 2D Array:
A **one-dimensional** array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.
Syntax:

```
float mark[5];
int mark[5] = {19, 10, 8, 17, 9};
int mark[] = {19, 10, 8, 17, 9};
```

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 # 2
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:

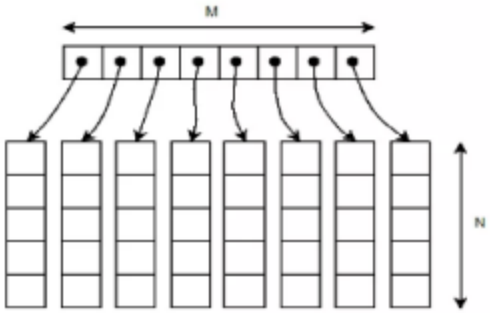
```
#define N 10
// Dynamically Allocate Memory for 1D Array in C++
int main(){
    // dynamically allocate memory of size 5 and assign values to allocated memory

      int *array{ new int[5]{ 10, 7, 15, 3, 11 } };

    // print the 1D array
    // deallocate memory
}
```

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(){
//Enter two dimensions N and M
//Dynamically allocate memory to both
int** ary = new int*[N];
  for(int i = 0; i < N; ++i)
      ary[i] = new int[M];
//fill the arrays
//print them
//deallocate the memeory
```

Task # 3

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.

Task # 4

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.

Hint: You can use Auxiliary array in both tasks.

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**. ie. 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){
```

```
            cout<<"Boundary Error\n";
        }
        else{
            Array[pos] = val;
        }
}
```

Task # 5

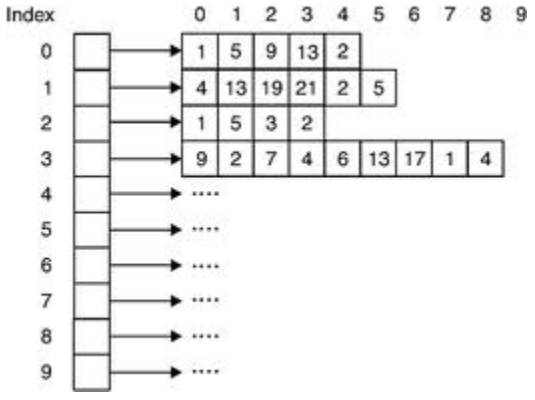
Write a program that creates a **2D array** of **5x5** values of **type boolean**. Suppose indices represent students and that the value at **row i**, **column j** of a **2D array** is true just in case **i and j** are **studying the same course** and **false otherwise**. Use the initializer list to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

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

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

Jagged Array:

Jagged array is nothing but it is **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
    // deallocate memory using delete[] operator
```

Task # 6

Write a program to calculate the cumulative distance in km covered by each airline. Assuming that each city may be at a max distance of 400 km from its next destination.

	Moscow	**Chicago**	**Astana**	**Edinburgh**	**Helsinki**
British Airways	366	333	400	300	266
Eastern Airways	333	300	366	300	---
Easy Jet	400	366	266	---	---
FlyBe	266	233	400	---	---
Ryanair	333	366	400	300	333

Rule of Three:

The Rule of Three is a rule of thumb in C++ (prior to C++ 11) which states that if a class defines one or more of the following, then it should probably explicitly define all three of these. The rule is also called Law of Big Three or The Big Three, three special functions which are:
1) Destructor
2) Copy Constructor
3) Overloaded/Copy Assignment Operator.

Example:

```
class Numbers{
//private members:
//public members
//copy constructor
/*
Numbers(const ClassName &amp;amp;obj)
{
Some code
}*/
~Numbers() //destructor
{
delete ptr;
```

```
}
};
int main(){
int arr[ 4 ] = { 11, 22, 33, 44 };
Numbers Num1( 4, arr );
// this creates problem (if not using the copy constructor)
Numbers Num2( Num1 );
return 0;
}
```

Task # 7

The task is to understand the concept of shallow and deep copies, and to understand the logic behind Rule of Three.

Write a code that create a Class named Numbers, with private size and pointer variables , a public function to assign values to a private member, and a destructor to destroy the pointer and memory. Understand what happens when you define an object by providing one object as an argument to another or simply assign one object to another.

Lab01: DynamicSafeArray with one/two dimensional pointers + Rule of Three		
Std Name:		**Std_ID:**
Section:		
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		

Instructor signature: _____