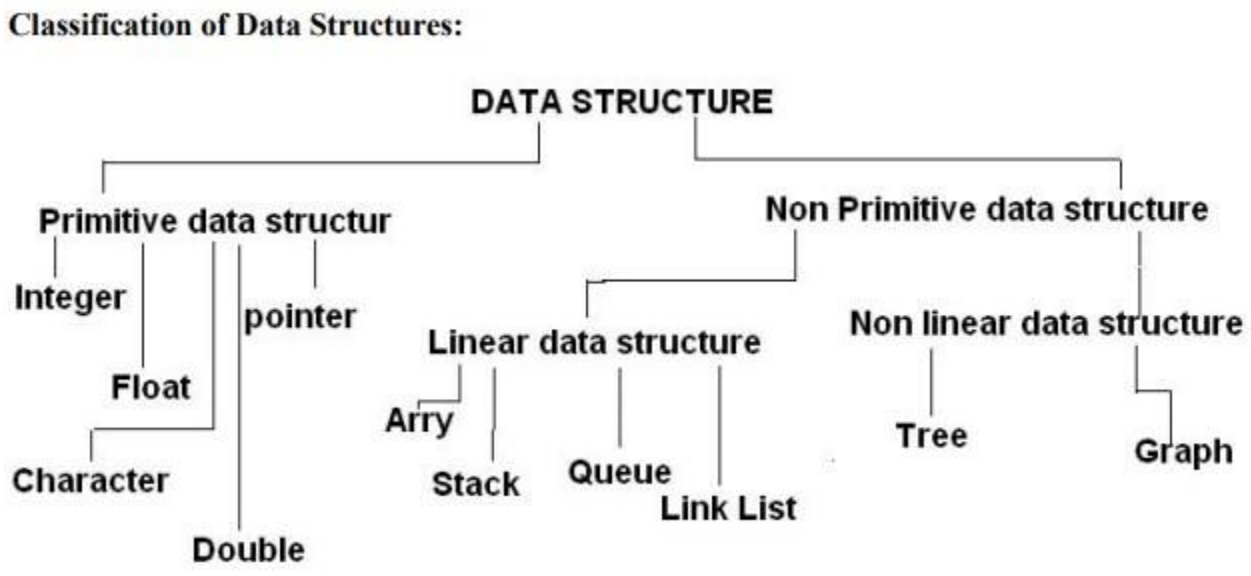
Data Structures Lab 02

Course: Data Structures (CL2001) **Semester**: Fall 2024
Instructor: Muhammad Khalid **T.A**: N/A

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

Data Structure:

Data may be organized in many different ways: the logical or mathematical model of a particular organization of data is called a data structure.



Linear Data Structures:

A data structure is said to be linear if its elements form a sequence or a linear list. In linear data structures, the data is arranged in a linear fashion. Example: Arrays, linked lists, stacks and queues.

Arrays:

The simplest type of data structure is a linear (or one-dimensional) array. By a linear array, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3,..., n.

If we choose the name A for the array, then the elements of A are denoted by subscript notation i.e. Al, A2, A3 or by the parenthesis notation i.e. A(1), A(2), A(3), …., A(N) or by the bracket notation A[1], A[2], A[3], ..., A[N]. Regardless of the notation, the number K in A[K] is called a subscript and A[K] is called a subscripted variable.

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};
```

→ Dynamic Arrays

To allocate an array dynamically we use array form of new and delete (new[ ] , delete[ ])

```
dynamic_array.cpp

// dynamic_array.cpp

#include<iostream>

using namespace std;

int main()
{
   int array[] = {1,2,3};
 cout << array[0];    cout << endl;

//   int* dArray = new int[] {1,2,3};    int* dArray = new int[3] {1,2,3};
cout << *dArray+1;    cout << endl;    cout << dArray[2];

   delete[] dArray;
}
```

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.

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.

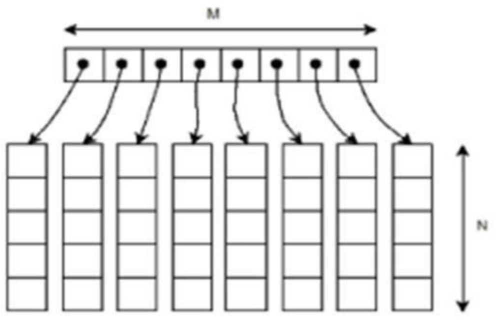
Example:

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;
  }
```

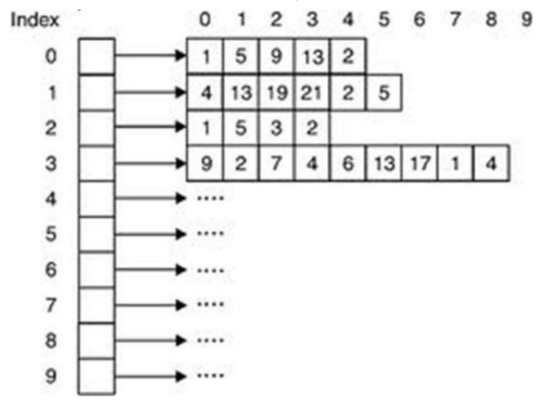
Safe Array:

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;
          }
  }
```

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
```

Lab Exercises:

1. Implement Jagged arrays from the above pseudocode by taking a size of 5 and resizing it to 10 in each index.
2. Create a header file called matrix_multuply.h that takes two arrays as input and multiplies them and outputs a multiplied array.
[HINT: Use 2D arrays to accomplish this]
3. 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. You can use initializer list to instantiate and initialize your array to represent the following configuration: (* means “friends”)

i/j	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.
4. You are tasked with developing a program to manage and display the Grade Point Average (GPA) for the core courses offered in the first semester of four departments: Software Engineering (SE), Artificial Intelligence (AI), Computer Science (CS), and Data Science (DS). Each department offers a distinct number of core courses for this semester: SE has 3 core courses, AI has 4 core courses, CS has 2 core courses, and DS has 1 core course. To efficiently store and present this data, which type of array structure would you employ? implement a solution using the chosen array structure to display the GPAs of the core courses for each department.
5. You're developing a program to manage a seating chart for a conference held in a hall with multiple rows of seats. Each row has a different seat capacity. To efficiently handle the seating arrangements, you decide to use a dynamic array. Implement a C++ code that allocates memory for the seating chart and allows attendees' names to be inputted for each seat. Choose and implement the appropriate type of dynamic array for this scenario.