CL2001 Data Structure Lab	Lab 2 1D and 2D Dynamic Safe Pointers and jagged array

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

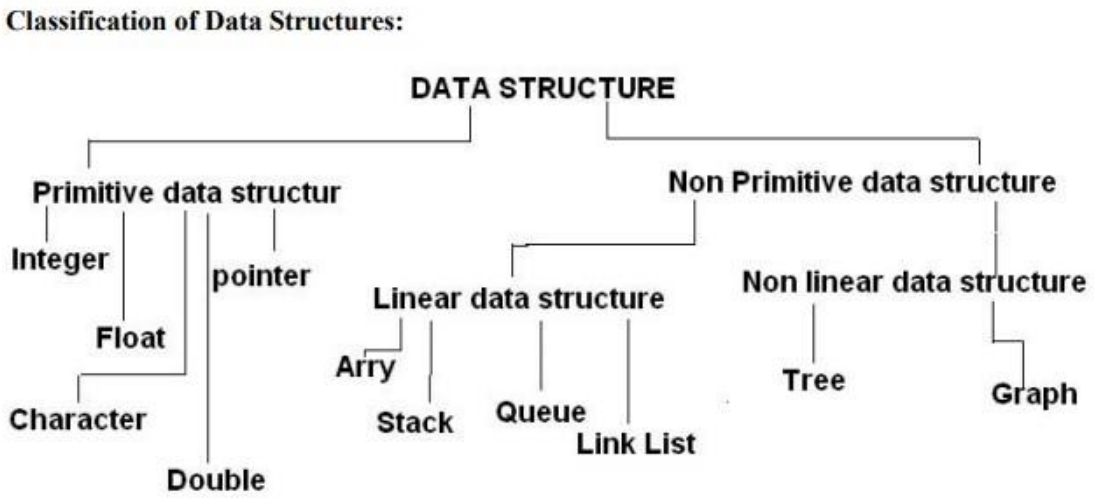
Fall 2025

Lab Content

1. 1D and 2D static Array
2. 1D and 2D Dynamic Arrays
3. Safe Arrays
4. Jagged Array

Data Structure

A data structure is a way of organizing and storing data in a computer so that it can be used efficiently.
Think of it as a container that holds data in a particular arrangement so that operations like searching, inserting, deleting, or sorting can be done quickly. **Below is the classification of data structure:**



Linear Data Structures

A Linear Data Structure is a type of data structure in which data elements are arranged sequentially (one after another). For Example: Arrays, linked lists, stacks and queues. In a linear data structure, every element has a unique predecessor and a unique successor (except the first and last element).

Characteristics of Linear Data Structures:

1. **Sequential organization** - elements are stored in a sequence.
2. **Single level** - data flows in one dimension (like a straight line).
3. **Traversal** - elements are visited one by one.
4. **Memory allocation** - can be contiguous (**arrays**) or non-contiguous (**linked lists**).

Arrays

An array is a collection of elements of the same data type, stored in contiguous (continuous) memory locations. It allows us to store and access multiple values using a single variable name with an index.

Key Features of Arrays

- Stores multiple values of the same type.
- Elements are stored in contiguous memory.
- Index-based access (fast lookup).
  - Index starts from 0 (first element).
  - Last element index = size – 1.
- Fixed size (once created, size cannot change).

1D and 2D arrays

1D array: a simple list of elements

int arr[5] = {10, 20, 30, 40, 50};

Memory:

Index: 0 1 2 3 4

Value: 10 20 30 40 50

2D array: Like a tabe (rows x columns)

```
int matrix[2][3] = {
  {1, 2, 3},
  {4, 5, 6}
};
```

Memory:

Row 0 → [1 2 3]

Row 1 → [4 5 6]

1D array Example

```
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    cout << "1D Array elements: ";
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

2D array Example

```
#include <iostream>
using namespace std;

int main() {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    cout << "2D Array elements:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Dynamic arrays

A 1D array is just a list of elements stored sequentially in memory. A dynamic array means the array size is not fixed at compile-time but is allocated at runtime using pointers and the new operator. Memory can be allocated and later freed using delete[].

Example 1D dynamic array

```cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Enter size of 1D array: ";
    cin >> n;

    // 1D Dynamic array
    int* arr = new int[n];   // allocate memory dynamically

    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "You entered: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;
    return 0;
}
```

2D Dynamic array

```cpp
#include <iostream>
using namespace std;

int main() {
    int rows, cols;
    cout << "Enter rows and cols: ";
    cin >> rows >> cols;

    // 2D Dynamic array
    int** arr = new int*[rows];    // array of row pointers
    for (int i = 0; i < rows; i++) {
        arr[i] = new int[cols];    // each row has 'cols' elements
    }

    cout << "Enter elements (" << rows << "x" << cols << "):\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> arr[i][j];
        }
    }

    cout << "Matrix:\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    // free memory
    for (int i = 0; i < rows; i++) {
        delete[] arr[i];
    }
    delete[] arr;

    return 0;
}
```

Resizing dynamic the arrays

```
int main() {

    int size = 5;
    int* arr = new int[size]{1, 2, 3, 4, 5};

    cout << "Original array: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    cout << "\n";

    int newSize = 8;
    arr = resizeArray(arr, size, newSize);

    // Fill new elements
//    for (int i = size; i < newSize; i++) {
//        arr[i] = (i + 1) * 10;  // 60, 70, 80
//    }

 for (int i = size; i < newSize; i++) {
    cin>>arr[i];
    }
    cout << "Resized to bigger array: ";
    for (int i = 0; i < newSize; i++) cout << arr[i] << " ";
    cout << "\n";

    size = newSize;
    newSize = 3;
    arr = resizeArray(arr, size, newSize);

    cout << "Resized to smaller array: ";
    for (int i = 0; i < newSize; i++) cout << arr[i] << " ";
    cout << "\n";

    delete[] arr;
    return 0;
}
```

```
#include <iostream>
using namespace std;

int* resizeArray(int* oldArr, int oldSize, int newSize) {

    if (newSize == oldSize) {
        return oldArr;
    }

    int* newArr = new int[newSize];

    int limit = (oldSize < newSize) ? oldSize : newSize;
    for (int i = 0; i < limit; i++) {
        newArr[i] = oldArr[i];
    }

    delete[] oldArr;

    return newArr;
}
```

Safe Array

In C++, when you create a normal array (static or dynamic), there is no **automatic bounds** checking.

```
int arr[5];
arr[10] = 100;   // ? Undefined behavior (out of range)
```

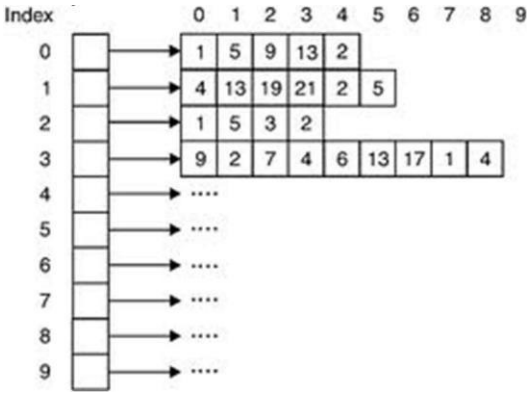
What will happen by executing above line?

Solution of above is safe array that Validates the index before accessing or modifying elements. Prevents writing outside the valid range. Optionally allows custom index ranges (like starting from -5 instead of 0). **See Example below:**

```
void set(int pos, Element val) {
    // check if position is valid
    if (pos < 0 || pos >= size) {
        cout << "Boundary Error\n";
    }
    else {
        Array[pos] = val;   // safe assignment
    }
}
```

Jagged Array

A Jagged Array is like a 2D array but each row can have a different number of columns.



Example:

```cpp
#include <iostream>
using namespace std;

int main() {
    int rows = 3;
    int **arr = new int*[rows];  // pointer to pointer (for jagged array)
    int Size[3];


    for (int i = 0; i < rows; i++) {
        cout << "Enter size of Row " << i + 1 << ": ";
        cin >> Size[i];
        arr[i] = new int[Size[i]]; // allocate each row with different size
    }
    for (int i = 0; i < rows; i++) {
        cout << "Enter " << Size[i] << " elements for Row " << i + 1 << ": ";
        for (int j = 0; j < Size[i]; j++) {
            cin >> arr[i][j];  // easier notation instead of pointer arithmetic
            //cin >> *(*(arr + i) + j);

        }
    }
    cout << "\nJagged Array Elements:\n";
    for (int i = 0; i < rows; i++) {
        cout << "Row " << i + 1 << ": ";
        for (int j = 0; j < Size[i]; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    for (int i = 0; i < rows; i++) {
        delete[] arr[i];  // free each row
    }
    delete[] arr;  // free row pointers

    return 0;
}
```

LAB TASKS

Task#1

You are tasked with implementing a dynamic matrix class in C++ that supports the following operations: Dynamic Matrix Creation: Create a dynamic 2D array (matrix) with rows and columns specified by the user. Matrix Resizing: Implement a method to resize the matrix. The new size should be provided as input (new rows and columns). If the new size is larger, initialize the new elements with a given value. If the new size is smaller, truncate the matrix. Matrix Transposition: Implement a method to transpose the matrix (rows become columns and vice versa). Matrix

Printing: Implement a method to print the matrix. After add 2 to each odd index then print the array. Memory Deallocation: Ensure proper memory management, including deallocation of the dynamic matrix when no longer needed.

Task#2

A school has conducted exams for 5 students in 4 subjects.
You are required to:

- Use a appropriate dynamic array to store marks.
- Calculate and display:
- Total marks of each student.
- Average marks of each subject.
- The topper student (highest total marks).

Task#3

A university is managing the marks of students in multiple subjects. Each department has a different number of students. Each student has marks for a fixed number of 5 subjects.

You need to store this data in a appropriate dynamic array where rows = number of departments and columns = number of students in each department. Then, calculate the highest, lowest and average marks of each department and display them.

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. 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.

Task#5

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.

Task#6

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.