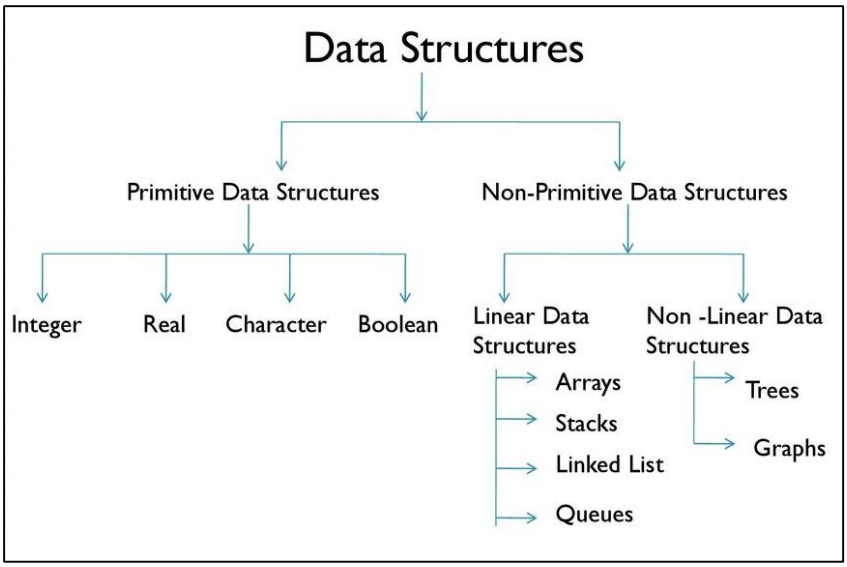
LECTURE: WEEK 01

INTRODUCTION TO DATA STRUCTURES

Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized

Types of Data Structures



Primitive Data Structures

Primitive Data Structures are the data structures consisting of the numbers and the characters that come in-built into programs.

Basic data types like Integer, Float, Character, and Boolean come under the Primitive Data Structures.

Non-Primitive Data Structures

Non-Primitive Data Structures are those data structures derived from Primitive Data Structures.

we can divide these data structures into two sub-categories -

1. Linear Data Structures
2. Non-Linear Data Structures

Linear Data Structures

In Linear Data Structure the arrangement of the data is done sequentially, where each element consists of the successors and predecessors except the first and the last data element. The Array, Linked Lists, Stacks, and Queues are examples of linear data structures.

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items. Trees and Graphs are examples of non-linear data structures.

Basic operation on Data Structure

1. Insert
2. Update
3. Delete
4. Search
5. Sort

RULE OF THREE (Related to C++)

The "Rule of Three" is a guideline in C++. It states that if a class needs to implement one of the following special member functions, it likely needs to implement all three:

1. Destructor: Responsible for releasing resources when an object goes out of scope or is explicitly deleted.
2. Copy Constructor: Defines how an object is copied when a new object is created as a copy of an existing one.
3. Copy Assignment Operator: Defines how an object is assigned a new value from an existing object.

Purpose

The Rule of Three ensures that your class handles copying and assignment properly, especially if it manages resources such as dynamic memory, file handles, or network connections. Without implementing all three, you might end up with resource leaks, double deletions, or shallow copies that lead to undefined behavior.

When to Apply

You should apply the Rule of Three if:

- Your class manages resources directly (e.g., it allocates memory with new, opens files, or acquires other system resources).
- You need to ensure deep copying of objects to avoid issues with shared resources.

Example

Consider a simple class that manages dynamic memory:

```
#include <iostream>
class MyClass {
public:
   MyClass(int size) : size_(size), data_(new int[size]) {
   ~MyClass() {
      delete[] data_; // Destructor
   }
   MyClass(const MyClass& other) : size_(other.size_), data_(new int[other.size_]) {
      std::copy(other.data_, other.data_ + other.size_, data_); // Copy Constructor
   }
   MyClass& operator=(const MyClass& other) {
      if (this == &other) return *this; // Self-assignment check
      delete[] data_; // Release old memory
      size_ = other.size_;
      data_ = new int[size_];
      std::copy(other.data_, other.data_ + size_, data_); // Copy Assignment Operator
      return *this;
   }
private:
   int size_;
   int* data_;
};
```

In this example:

- **Destructor:** Releases the dynamically allocated memory.
- **Copy Constructor:** Creates a deep copy of the object to ensure that each object has its own separate memory.

- **Copy Assignment Operator:** Handles self-assignment, releases old memory, and makes a deep copy of the assigned object's data.

ABSTRACT DATA TYPE (ADT)

An ADT is a collection of data and associated operations for manipulating that data ADTs support abstraction, encapsulation, and information hiding. They provide equal attention to data and operations Common examples of ADTs: – Built-in types: boolean, integer, real, array – User-defined types: stack, queue, tree, list

ARRAYS

Definition

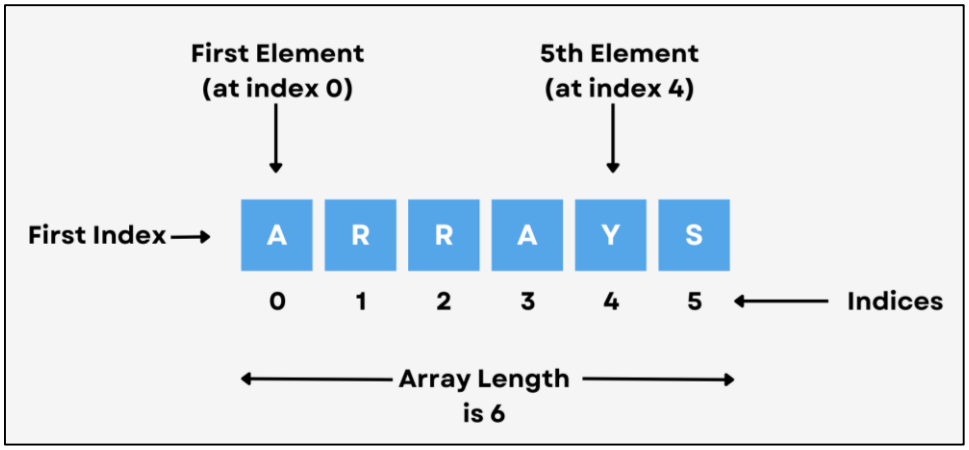
- Arrays in C++ are a fundamental data structure used to store multiple values of the same type in a contiguous block of memory.
- They provide a way to manage collections of data efficiently.

Types of Arrays

There are two types of arrays: **static** and **dynamic.**

STATIC ARRAYS

- A static array has a fixed size determined at compile time.

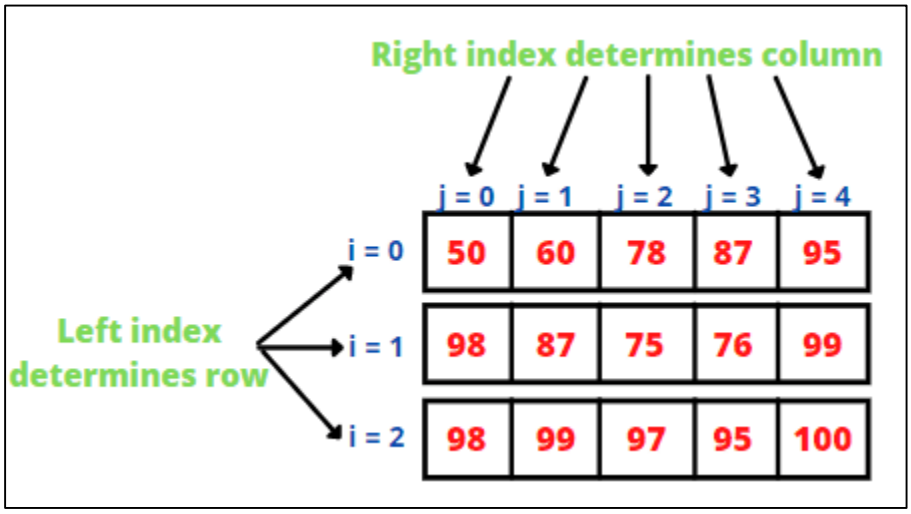


Example of 1d Static Array

```
#include <iostream>

int main() {
   int numbers[5]; // Declares an array of 5 integers
   // Initialize the array
   numbers[0] = 10;
   numbers[1] = 20;
   numbers[2] = 30;
   numbers[3] = 40;
   numbers[4] = 50;
   // Print array elements
   for (int i = 0; i < 5; ++i) {
      std::cout << numbers[i] << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Example of 2d Static Array




```
#include <iostream>

int main() {

   int matrix[2][3] = { // 2 rows and 3 columns

      {1, 2, 3},

      {4, 5, 6}

   };

   // Print the matrix

   for (int i = 0; i < 2; ++i) {

      for (int j = 0; j < 3; ++j) {

         std::cout << matrix[i][j] << " ";

      }

      std::cout << std::endl;

   }

   return 0;

}
```

DYNAMIC ARRAYS

Definition

- If the size of the array is not known at compile time, you can use dynamic memory allocation with pointers.

The new Keyword

- In C++, we can create a dynamic array using the new keyword.
- The new operator denotes a request for memory allocation on the Free Store.
- If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator

```
pointer-variable = new data-type;
```

delete operator

- Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.

Syntax to use delete operator

```
delete pointer-variable;
```

Example of 1D dynamic array

```
#include <iostream>

int main() {

   int size;

   std::cout << "Enter the size of the array: ";

   std::cin >> size;

   // Allocate memory

   int* array = new int[size];

   // Initialize and use the array

   for (int i = 0; i < size; ++i) {
```

```
        array[i] = i * 10;
    }
    // Print the array
    for (int i = 0; i < size; ++i) {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;
    // Deallocate memory
    delete[] array;
    return 0;
}
```

Example of 2D dynamic array

```
#include <iostream>
int main() {
    int rows = 3, cols = 4;
    // Allocate memory for rows
    int** array = new int*[rows];
    // Allocate memory for columns
    for (int i = 0; i < rows; i++) {
        array[i] = new int[cols];
    }
    // Initialize and use the array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            array[i][j] = i * cols + j;
        }
```

```
 }
   // Print the array
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         std::cout << array[i][j] << " ";
      }
      std::cout << std::endl;
   }
   // Free the allocated memory
   for (int i = 0; i < rows; i++) {
      delete[] array[i];
   }
   delete[] array;
   return 0;
}
```

DYNAMIC 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:

Example

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