UST
Faculty of Engineering
First Year [Sem2]

Data Structure and Algorithms

Lecture 4

Non Primitive Data Structures

Non Primitive data structures

Non-primitive data structures are user-defined data structures constructed by combining primitive types (int, float, char).

Unlike primitive types that store single values, non-primitive simple linear structures organize collections of items in a strict sequential order.

Classification of Non-Primitive Data Structures

- Non-primitive data structures can be divided into two major categories:

A. Linear Data Structures

- Data elements are arranged sequentially.

Examples:

- Array
- Linked List
- Stack
- Queue

10 → 20 → 30 → 40 → 50

- Each element has a logical relationship with the next element.

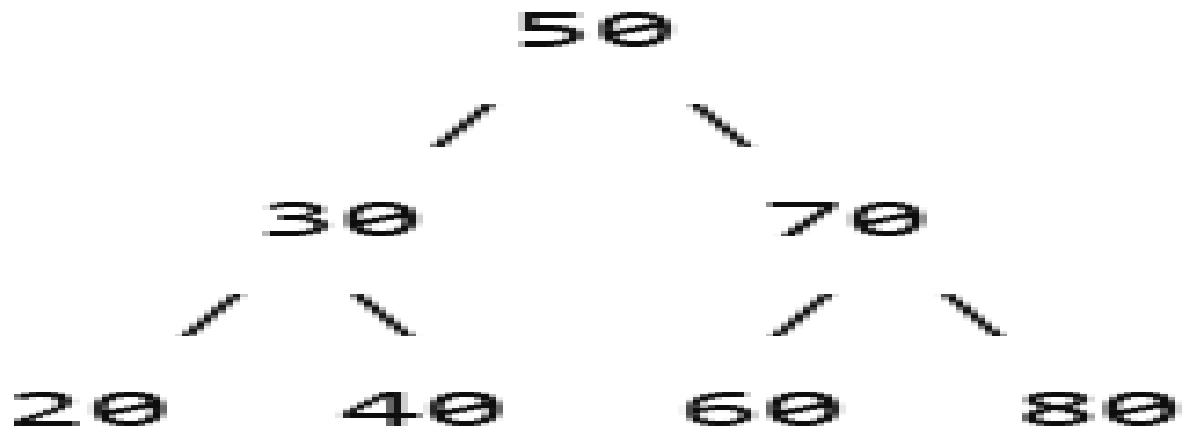
B. Non-Linear Data Structures

- Data elements are not arranged in a simple sequential order.

Examples:

- Trees
- Graphs

Example of a tree:



Simple Data Structures

we use simple structures to refer to basic non-primitive structures that are relatively easy to understand and implement.

Important examples are:

- Arrays
- Strings
- Structures (struct)
- Enumerations(enum)

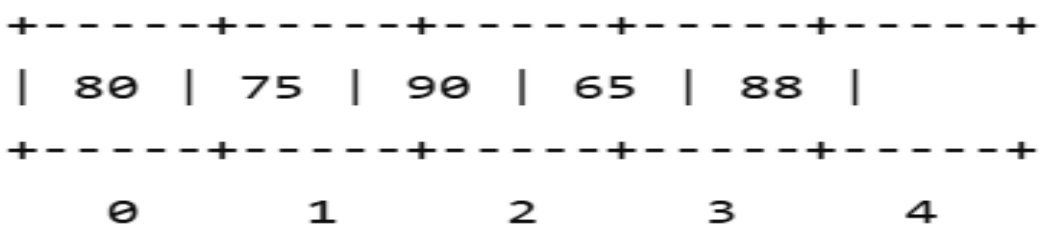
Arrays

- An array is a collection of elements of the same data type, stored under one variable name.

Example:

```
int marks[5];
```

- This creates an array capable of storing five integers.

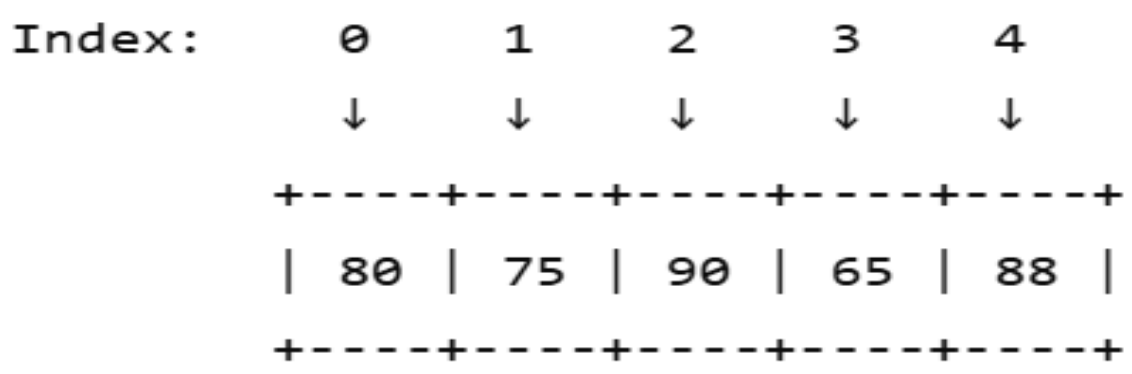


Array Indexing

- C++ arrays use zero-based indexing.
- This means the first element has index 0.For an array of five elements:

Therefore:

- marks[0] = 80;
- marks[1] = 75;
- marks[2] = 90;
- marks[3] = 65;
- marks[4] = 88;



Declaring an Array

The general syntax is:

dataType arrayName[size];

Example:

int marks[5];

Another example:

float temperatures[7];

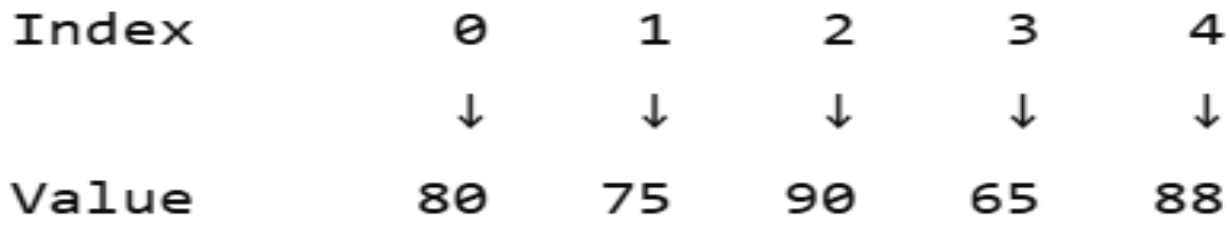
This creates an array containing seven floating-point values.

Initializing an Array

An array can be initialized when it is declared.

int marks[5] = {80, 75, 90, 65, 88};

The result is:



The compiler assigns the values to the corresponding positions.

Accessing Array Elements

- To access an individual element, use its index.

```
cout << marks[0];
```

Output:

80

Example:

```
cout << marks[3];
```

Output:

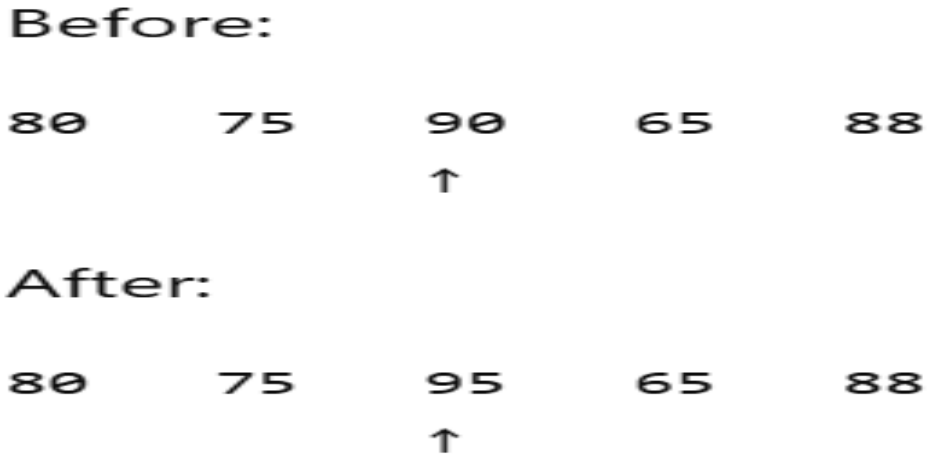
65

Modifying an Array Element

- An array element can be changed using its index.

For example:

- marks[2] = 95;



Complete Array Example

```
#include <iostream>
using namespace std;
int main() {
int marks[5] = {80, 75, 90, 65, 88};
cout << "First mark: " << marks[0] << endl;
cout << "Third mark: " << marks[2] << endl;
return 0;
}
```

Output

```
First mark: 80
Third mark: 90
```

Processing Arrays Using Loops

One of the major advantages of arrays is that loops can be used to process all elements.

Example:

```
#include <iostream>
using namespace std;
int main(){
int marks[5] = {80, 75, 90, 65, 88};
for(int i = 0; i < 5; i++)    {
 cout << marks[i] << endl;
}
return 0;
}
```

Output:

```
80
75
90
65
88
```

The loop changes the value of `i`:

```
i = 0 → marks[0]
i = 1 → marks[1]
i = 2 → marks[2]
i = 3 → marks[3]
i = 4 → marks[4]
```

Calculating the Sum of Array Elements

Example:

```
#include <iostream>
using namespace std;
int main(){
int marks[5] = {80, 75, 90, 65, 88};
int sum = 0;
for(int i = 0; i < 5; i++)    {
 sum = sum + marks[i];
   }
cout << "Total = " << sum;
return 0;
}
```

Output:

```
Total = 398
```

Calculating the Average

- The average can be calculated as:
- Average = Sum / Number of Elements

Example:

```
#include <iostream>
using namespace std;
int main(){
 int marks[5] = {80, 75, 90, 65, 88};
 int sum = 0;
 for(int i = 0; i < 5; i++)    {
    sum += marks[i];    }
  double average = sum / 5;
 cout << "Average = " << average;
  return 0;
}
```

Output:

```
Average = 79.6
```

Types of Array:

1. One Dimensional Array.
2. Two Dimensional Array

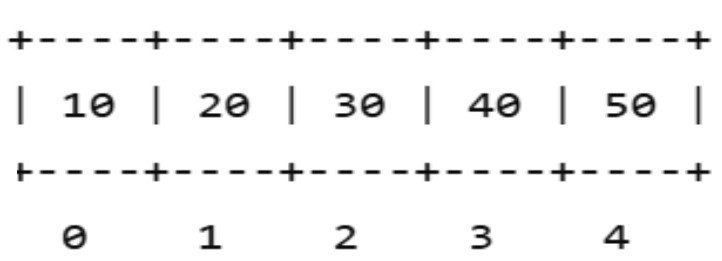
One-Dimensional Array

- A one-dimensional array contains elements in a single sequence.

Example:

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

Representation:



This is useful for:

- Student marks
- Temperatures
- Product prices
- Employee salaries
- Sensor readings

Two-Dimensional Arrays

A two-dimensional array can be viewed as a table containing rows and columns.

Example:

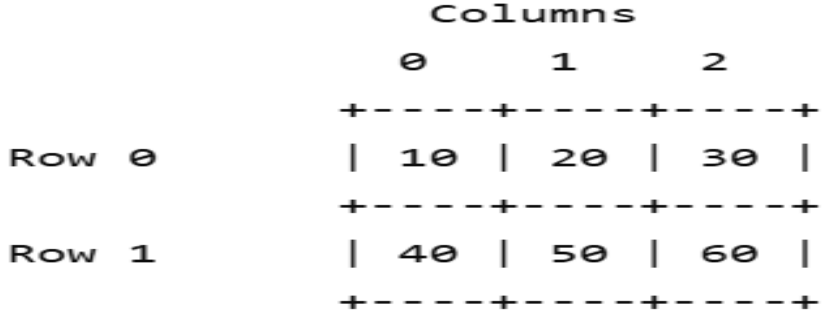
int matrix[2][3];

It contains:

2 rows

3 columns

Representation:



Initialization:

```
int matrix[2][3] =
{
     {10, 20, 30},
     {40, 50, 60}
};
```

Accessing a Two-Dimensional Array

To access an element:

matrix[row][column]

Example:

cout << matrix[0][1];

Output:20

Because:matrix[0][1]

row = 0

column = 1

Nested Loops with 2D Arrays

A nested loop can be used to process a matrix.

```
#include <iostream>
using namespace std;
int main(){
 int matrix[2][3] =    {       {10, 20, 30},       {40, 50, 60}    };
  for(int i = 0; i < 2; i++)    {
 for(int j = 0; j < 3; j++)       {
       cout << matrix[i][j] << " ";
   }       cout << endl;
  }
 return 0;
}
```

Output:

```
10 20 30
40 50 60
```

Applications of Arrays

Arrays are widely used in engineering and computer science.

Examples:

Engineering

- Sensor measurements
- Temperature readings
- Voltage measurements
- Simulation data
- Signal processing

Software

- Lists of values
- Search algorithms
- Sorting algorithms
- Matrices Tables

Advantages of Arrays

1. **Simple**

Arrays are easy to understand and implement.

2. Fast Access

An element can be accessed directly using its index.

marks[3]

3. Efficient Storage

Elements are stored in an organized manner.

4. Easy Processing

Loops can process all elements efficiently.

Limitations of Arrays

1. Fixed Size

A traditional C++ array has a fixed size.

int numbers[5];

It cannot normally be expanded to six elements during its lifetime.

2. Same Data Type

All elements normally have the same type.

For example:

int numbers[5];

stores integers.

3. Insertion and Deletion

Adding or removing elements from the middle may require shifting other elements.

Strings

- A string is a sequence of characters used to represent text.

Example:

"Engineering“

- A string consists of characters:

E n g i n e e r i n g

- In C++, strings can be represented using the string class.

```
string name = "Ahmed";
```

Character Arrays

- A string can also be represented using a character array.

char name[] = "Ahmed";

Conceptually:

The **\0** is the null character, which indicates the end of a C-style string.

Using the C++ String Class

- Modern C++ programs commonly use:

```
#include <string>
```

Example:

```
#include <iostream>
#include <string>
using namespace std;
int main(){
string name = "Ahmed";
 cout << "Name: " << name;
 return 0;
}
```

Basic String Operations

Common operations include:

- Concatenation
- Finding length
- Comparing strings
- Accessing characters

Example:

```
string firstName = "Ahmed";
string lastName = "Ali";
string fullName = firstName + " " + lastName;
cout << fullName;
```

Output:Ahmed Ali

Finding String Length

- The length() function can be used.
- string name = "Ahmed";
- cout << name.length();

```
Output:

5

The characters are:

A h m e d
1 2 3 4 5
```

Accessing Characters in a String

- Strings can be accessed using indices.

```
string name = "Ahmed";
```

```
cout << name[0];
```

Output:A

And:

```
cout << name[2];
```

Output:m

- Therefore, strings behave similarly to character arrays in terms of indexing.