UST
Faculty of Engineering
First Year [Sem2]

Data Structure and Algorithms

Lecture 5

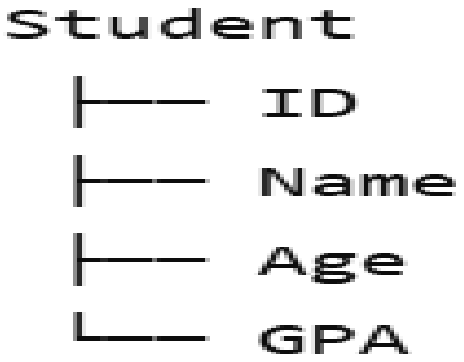
Non-Primitive data Structures (Simple Structures)

Structures (struct)

An important simple non-primitive structure in C++ is the **structure**.

A structure allows us to group variables of different data types under one name.

For example, a student may have:



These values have different data types.

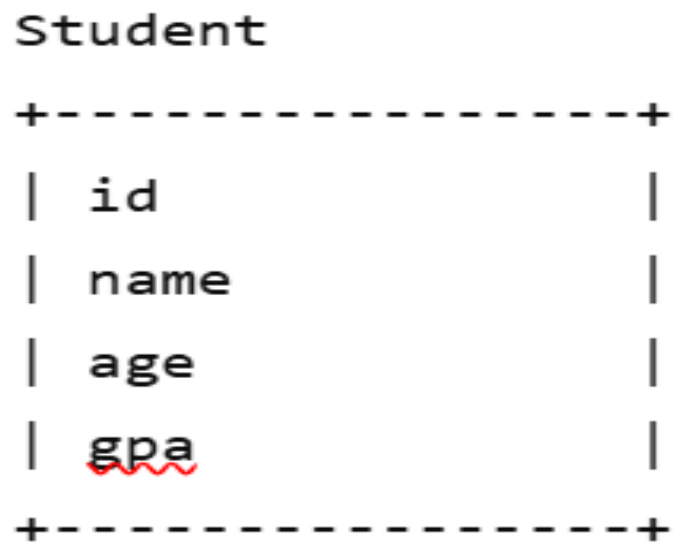
Why Do We Need Structures?

Consider:

- int id;
- string name;
- int age;
- double gpa;
- These variables describe one student, but they are independent variables.

- A structure allows us to combine them:

```
struct Student
{
int id;
 string name;
int age;
double gpa;
};
```



- Now all information belongs to one logical entity:

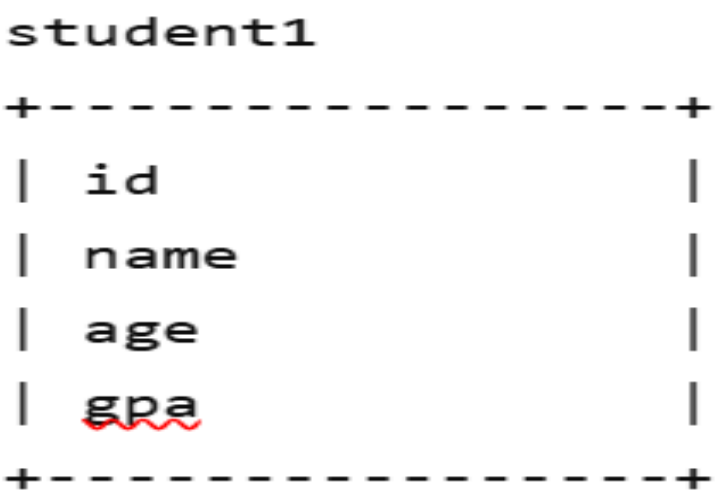
Defining a Structure

General syntax:

```
struct StructureName{
dataType member1;
 dataType member2;
 dataType member3;
};
```

Creating a Structure Variable

- After defining the structure:
- Student student1;
- Now student1 contains:



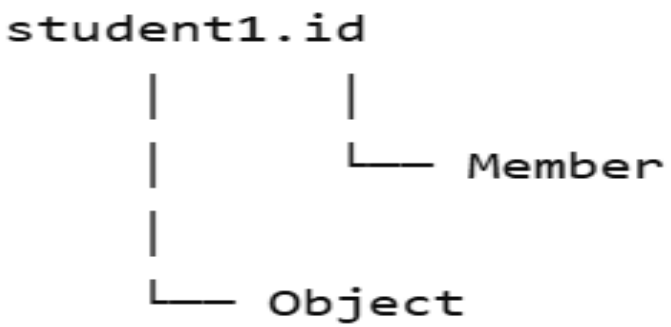
Assigning Values to Structure Members

- The dot operator **.** is used to access members.

Example:

- student1.id = 101;
- student1.name = "Ahmed";
- student1.age = 20;
- student1.gpa = 3.5;

The dot means:



Complete Structure Example

```
#include <iostream>
#include <string>
using namespace std;
struct Student{
 int id;
 string name;
int age;
};
int main(){
Student student1;
student1.id = 101;
student1.name = "Ahmed";
student1.age = 20;
cout << "ID: " << student1.id << endl;
  cout << "Name: " << student1.name << endl;
  cout << "Age: " << student1.age << endl;
return 0;
}
```

```
Output:

ID:  101
Name:  Ahmed
Age:  20
```

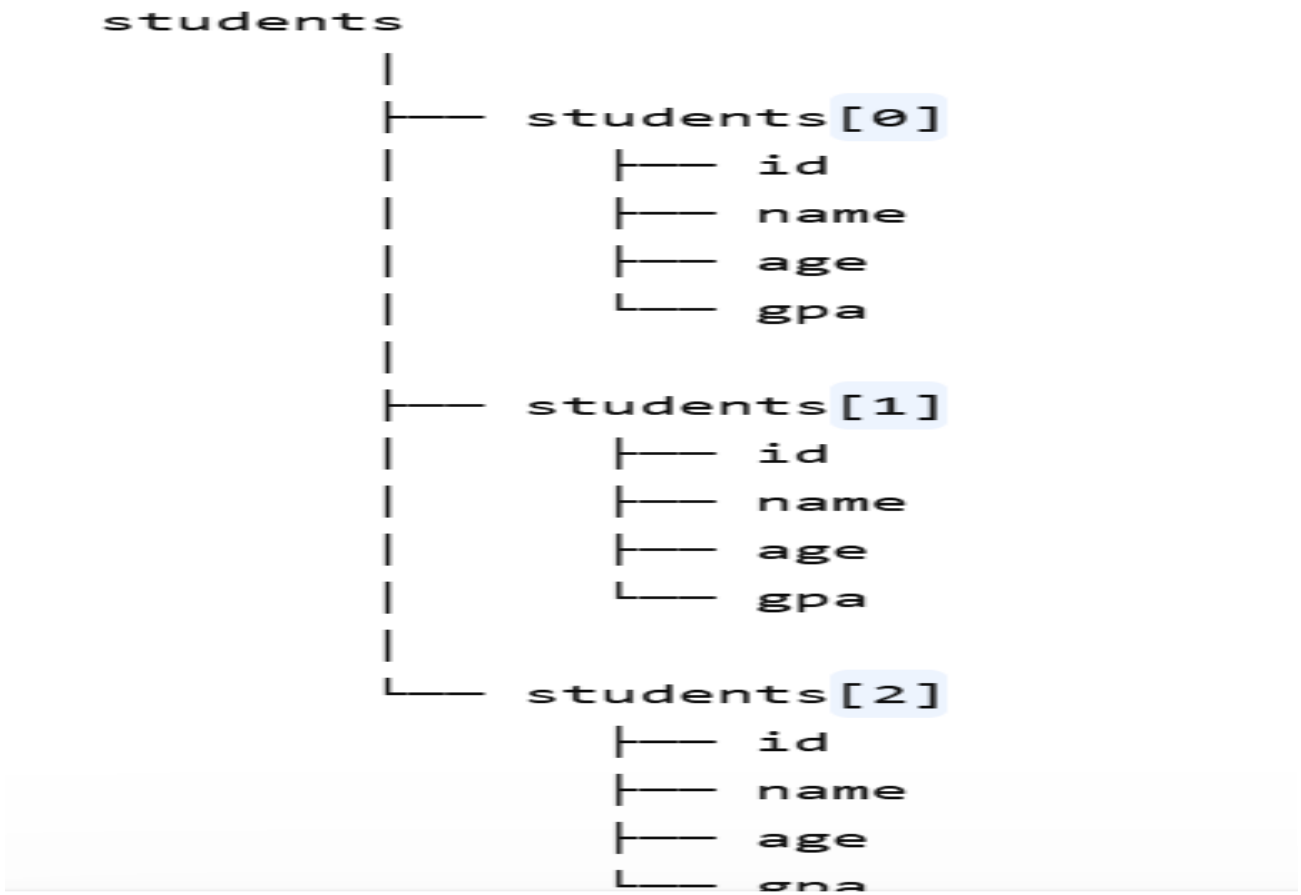
Array of Structures

Structures become especially useful when combined with arrays.

Suppose we have several students:

Student students[3];

Example




```
Student students[3] =
{
    {101, "Ahmed", 20, 3.5},
    {102, "Sara", 21, 3.8},
    {103, "Omar", 19, 3.2}
};
```

Processing an Array of Structures

- A loop can be used:

```
for(int i = 0; i < 3; i++){
cout << students[i].name << endl;
}
```

Output:

```
Ahmed
Sara
Omar
```

- This is a very important concept because real-world applications frequently manage collections of records.

Enumeration (enum)

An enumeration is a user-defined type containing a set of named constant values.

Example:

```
enum Day{
Monday,
Tuesday,
Wednesday,
Thursday,
Friday
};
```

We can create a variable:

```
Day today = Monday;
```

Why Use Enumeration?

Suppose we write:

int status = 1;

It may not be obvious what 1 means.

Instead:

```
enum Status{
 Pending,
Approved,
Rejected};
```

Then:

Status applicationStatus = Approved;

This makes the program easier to understand.

Simple Structures Summary

- The main structures introduced in this lecture can be summarized as follows:

Structure	Purpose	Example
Array	Store multiple values of the same type	`int marks[5]`
String	Store text	`string name`
Structure	Group different types	`Student`
Enumeration	Represent a fixed set of named values	`Day`

Array vs Structure

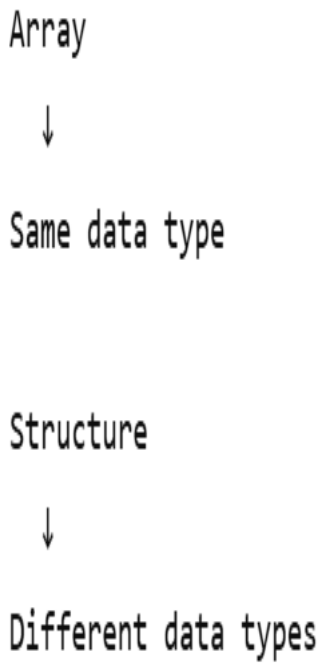
An important difference is:

Array

- Stores multiple values of the **same type**.

Structure

Can store values of different types.



Array of Structures vs Structure of Arrays

These two concepts are useful for engineering students.

Array of Structures

Student students[100];

Conceptually:

Student 1 → ID, Name, GPA

Student 2 → ID, Name, GPA

Student 3 → ID, Name, GPA

Structure of Arrays

Another approach is:

```
struct Students{
  int id[100];
string name[100];
 double gpa[100];
};
```

Conceptually:

- IDs → 101 102 103 ...
- Names → Ali Sara Omar ...
- GPA → 3.2 3.8 3.5 ...
- Both approaches can be useful depending on the application.

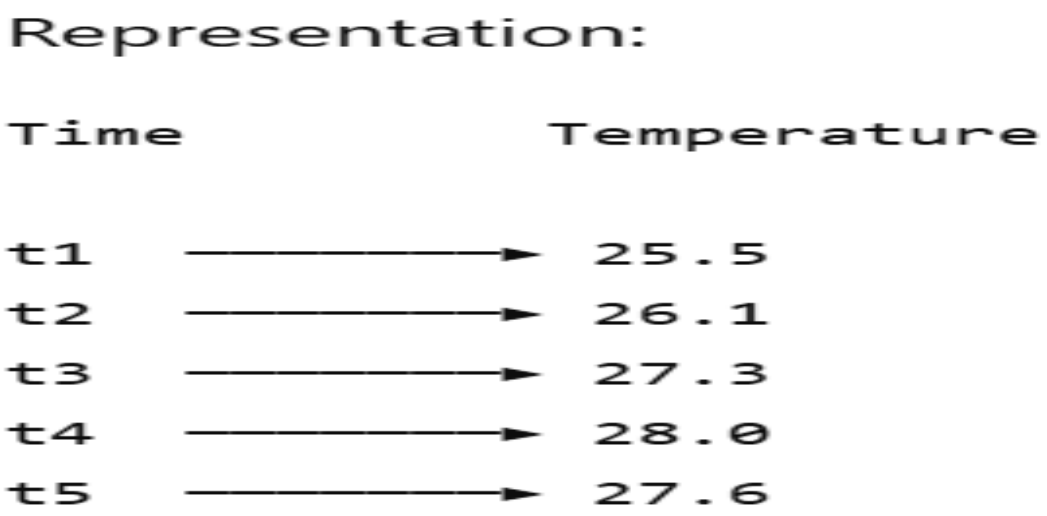
Engineering Sensor System

Imagine an engineering system that records temperature readings.

An array can store the measurements:

double temperature

[5] ={ 25.5, 26.1, 27.3, 28.0, 27.6};



Real-World Example: Sensor Record

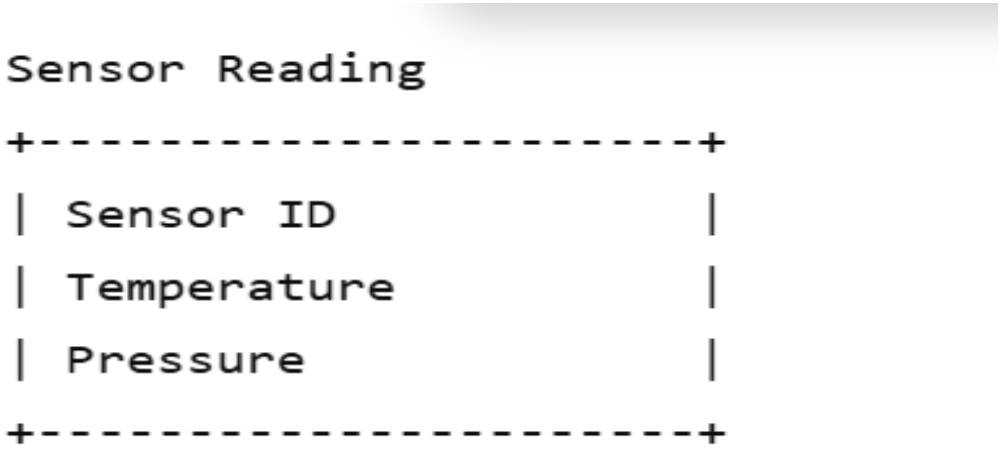
If every reading contains several properties, a structure is more appropriate.

```
struct SensorReading{
 int sensorID;
double temperature;
double pressure;
};
```

Now:

SensorReading reading1;

can contain:



Searching an Array

A common operation is searching for a particular value.

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
int target = 30;
for(int i = 0; i < 5; i++){
 if(numbers[i] == target)    {
 cout << "Found at index " << i;
 }
}
```

Output:

```
Found at index 2
```

This is a simple **linear search.**

Finding the Largest Element

Example:

```
int numbers[5] = {10, 45, 20, 80, 30};
int largest = numbers[0];
for(int i = 1; i < 5; i++){
 if(numbers[i] > largest)
  {
 largest = numbers[i];
  }
}
cout << "Largest = " << largest;
```

Output:

```
Largest = 80
```

Finding the Smallest Element

```
int numbers[5] = {10, 45, 20, 80, 30};
int smallest = numbers[0];
for(int i = 1; i < 5; i++){
 if(numbers[i] < smallest)    {
 smallest = numbers[i];
 }
}
cout << "Smallest = " << smallest;
```

Output:

```
Smallest = 10
```

Common Array Errors

Error 1: Invalid Index

If:

int numbers[5];

valid indices are:0 1 2 3 4

This is invalid:

numbers[5]

because index 5 is outside the array.

Error 2: Wrong Loop Condition

Incorrect:

- for(int i = 0; i <= 5; i++)

Correct:

- for(int i = 0; i < 5; i++)

Memory Representation

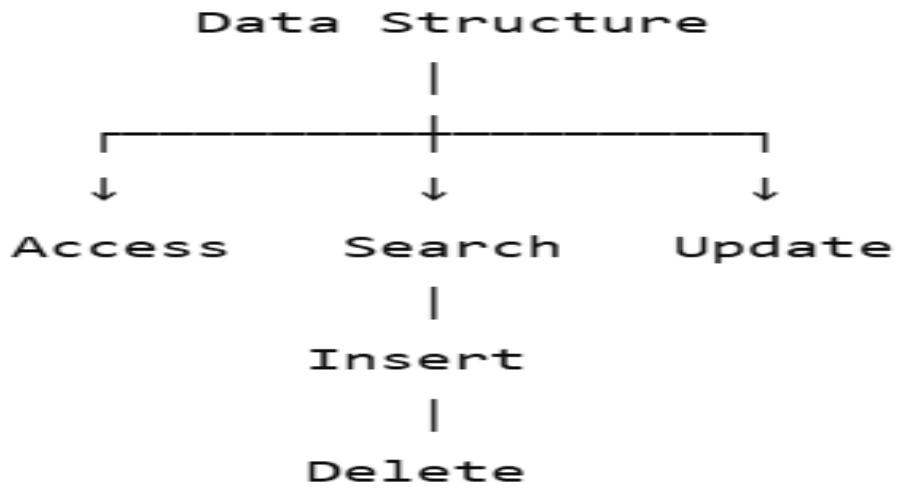
- One important characteristic of an array is that its elements are stored in contiguous memory locations.

Conceptually:

- Memory1000 → 10
- 1004 → 20
- 1008 → 30
- 1012 → 40
- 1016 → 50
- For an integer array, the exact number of bytes per element depends on the implementation, but the important idea is that array elements are stored sequentially.
- This helps explain why indexed access is efficient.

Simple Data Structures and Algorithm Operations

- When studying data structures, students should consider common operations:



For arrays:

- Access → direct usi
- Search → often sequential
- Update → direct using index
- Insertion → may require shifting
- Deletion → may require shifting

Choosing the Appropriate Structure

A programmer should ask:

Question 1: Do I need to store many values of the same type?

Use: Array

Question 2: Do I need to store text?

Use: String

Question 3: Do I need to group different types describing one object?

Use: Structure

Question 4: Do I have a small fixed set of named choices?

Use: Enumeration

Example: Student Management System

Suppose we need to store:

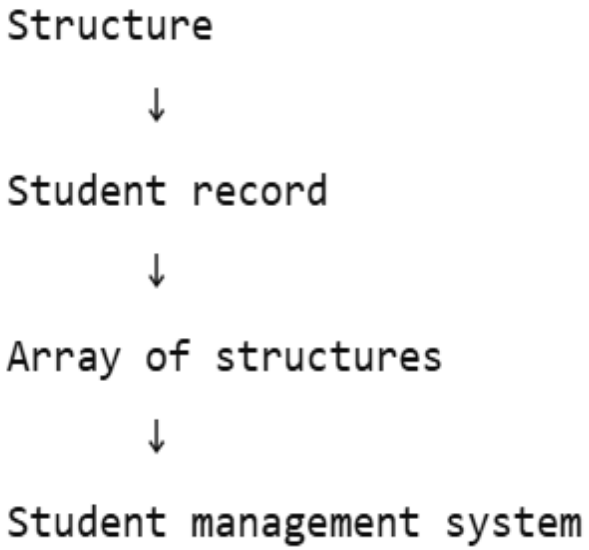
Student ID, Name, Age and GPA.

A suitable structure is:

```
struct Student{
 int id;
 string name;
int age;
double gpa;
};
```

For multiple students:

```
Student students[100];
```



Example: Matrix in Engineering

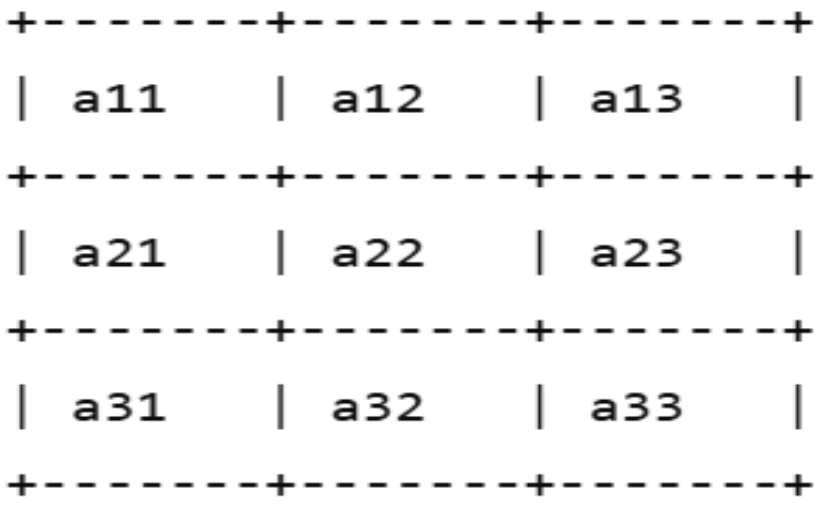
- Matrices are commonly used in engineering.

Example:

- double matrix[3][3];

Applications include:

- Electrical engineering calculations
- Mechanical engineering calculations
- Computer graphics
- Image processing
- Numerical methods
- Machine learning



Important Terminology

1. **Element:** An individual value stored in a data structure.
2. **Index:** The position used to access an array element.
3. **Dimension:** The number of indexing levels in an array.
4. **Member:** A variable contained inside a structure.
5. **Record:** A collection of related fields describing one entity.
6. **Data Type:** Defines the type of data that can be stored.

Key Differences

Feature	Array	String	Structure
Main purpose	Store multiple values	Store text	Group related data
Data types	Usually same	Characters/text	Can be different
Access	Index	Index/functions	Member name
Example	`int a[5]`	`string name`	`Student s`
Example application	Marks	Student name	Student record

Key Points

1. Data structures organize data for efficient processing.
2. Primitive data types include int, char, float, double, and bool.
3. Non-primitive structures are built to organize collections or relationships among data.
4. Arrays store multiple elements of the same type.
5. C++ arrays use zero-based indexing.
6. A one-dimensional array is a sequence of elements.
7. A two-dimensional array represents rows and columns.

8. Strings represent sequences of characters.

9. Structures group related variables, potentially of different types.

10. Enumerations represent a fixed set of named constants.

11.Arrays and structures can be combined.

12.The choice of data structure depends on the problem being solved.