Module – 3

Introduction to OOPS Programing

Q1. What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

1. Basic Concept

- **Procedural Programming:**
  Based on the concept of **procedures or routines (functions)**. Programs are organized as a sequence of instructions and function calls.
- **OOP (Object-Oriented Programming):**
  Based on the concept of **objects and classes**. Programs are organized around objects that combine data and behavior

2. Program Structure

- **Procedural:**
  Code is structured in **functions** and **procedures** that operate on data.
- **OOP:**
  Code is structured using **classes and objects**. Data and methods are encapsulated together.

3. Data Handling

- **Procedural:**
  Data is typically **global or passed explicitly** to functions.
  Emphasis on **functions** acting on data.
- **OOP:**
  Data is kept **private or protected** inside objects.
  Objects control access via **methods**.

4. Key Features

- **Procedural:**
  - Sequential execution
  - Top-down approach
  - Function reuse
- **OOP:**
  - **Encapsulation**
  - **Inheritance**
  - **Polymorphism**
  - **Abstraction**

5. Reusability

- **Procedural:**
  Reusability through functions or procedures.
- **OOP:**
  Reusability through **inheritance** and **polymorphism**, which allows creating new classes based on existing ones.

6. Example Languages

- **Procedural:**
  C, Pascal, Fortran
- **OOP:**
  C++, Java, Python, C#

7. Modifiability and Maintainability

- **Procedural:**
  Harder to modify large programs as changes in one function may affect others.
- **OOP:**
  Easier to modify and maintain code due to **encapsulation** and **modular design**.

8. Real-World Modeling

- **Procedural:**
  Less natural for modeling real-world problems directly.
- **OOP:**
  Better suited for modeling **real-world entities** as objects with state and behavior.

Q2. List and explain the main advantages of OOP over POP.

1. Modularity

- **OOP:** Code is divided into **classes and objects**, each with a clear responsibility. This modular structure makes it easier to manage, develop, and debug.
- **Advantage:** Changes in one module (class) typically don't affect others.

2. Reusability

- **OOP:** Supports **code reuse** through **inheritance**—a class can inherit properties and methods from another class.
- **Advantage:** You don't need to rewrite code; common features can be defined once and reused.

3. Encapsulation

- **OOP:** Bundles data and methods together and **hides internal details** from the outside world using access modifiers (`private`, `public`, `protected`).
- **Advantage:** Protects data from unintended changes, improving security and reliability.

4. Abstraction

- **OOP:** Allows programmers to focus on **what an object does**, rather than **how it does it**.
- **Advantage:** Simplifies complex systems by exposing only essential features.

5. Inheritance

- **OOP:** Enables new classes to **derive** from existing ones, inheriting their behavior and adding new features.
- **Advantage:** Promotes **code reuse** and reduces redundancy

6. Polymorphism

- **OOP:** Allows **the same interface to behave differently** depending on the object.
- **Advantage:** Increases flexibility and scalability. For example, the same method name can work on different types of objects.

7. Improved Maintainability

- **OOP:** With a modular structure and encapsulated design, it's easier to **modify, extend, and maintain** code.
- **Advantage:** Bugs can be fixed or features added with minimal impact on the rest of the system.

8. Better Modeling of Real-World Problems

- **OOP:** Objects represent **real-world entities**, making it intuitive to design and understand software.
- **Advantage:** More natural alignment with the problem domain.

9. Scalability

- **OOP:** Easier to scale due to its modular and hierarchical structure.
- **Advantage:** Large projects can be divided into manageable components.

10. Team Collaboration

- **OOP:** Different team members can work on different classes or modules independently.
- **Advantage:** Improves development efficiency in large projects.

Q3. Explain the steps involved in setting up a C++ development environment.

1. Install a C++ Compiler

A compiler is needed to convert C++ code into machine code. Common compilers include:

- **GCC (GNU Compiler Collection)** – for Linux and Windows (via MinGW or WSL)
- **Clang** – often used on macOS
- **MSVC** – Microsoft C++ compiler for Windows

◈ For Windows:

- Install **MinGW-w64** (GCC for Windows):
  - Download from: https://www.mingw-w64.org/
  - During installation, select architecture (usually `x86_64`) and threads (`posix`) with `seh` exception.

◈ For macOS:

- Install Xcode Command Line Tools:

  ```
  xcode-select --install
  ```

◈ For Linux (Ubuntu/Debian):

- Install GCC:

  ```
  sudo apt update
  sudo apt install g++
  ```

---

2. Choose a Text Editor or IDE

You need an editor or IDE to write and manage your code.

◆ Popular IDEs:

- **Code::Blocks** – Lightweight C++ IDE (https://www.codeblocks.org/)
- **Dev C++** – Simple and beginner-friendly
- **Visual Studio** (not Visual Studio Code) – Full-featured for Windows
- **CLion** – Advanced C++ IDE from JetBrains (paid, with free trial/student license)

◆ Popular Editors:

- **Visual Studio Code (VS Code)** – Lightweight and extensible
- **Sublime Text**
- **Atom**

For beginners, **VS Code with MinGW** is a popular choice.

---

3. Install and Configure the IDE or Editor

◆ For Visual Studio Code:

1. **Download and install VS Code**: https://code.visualstudio.com/
2. Install the **C++ Extension** by Microsoft:
   - Go to Extensions (Ctrl+Shift+X), search for "C++", and install it.
3. Add your **compiler to PATH** (especially for MinGW):
   - Add `C:\MinGW\bin` or `C:\Program Files\mingw-w64\...\bin` to system environment variables.
4. Create a basic file like `hello.cpp` and compile it:

   ```
   g++ hello.cpp -o hello
   ./hello
   ```

---

4. Verify Installation

Open a terminal or command prompt and run:

```
g++ --version
```

You should see version details, which confirms the compiler is installed correctly.

---

5. Write and Run Your First Program

Example:

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

 main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

To compile and run:

```
g++ hello.cpp -o hello
./hello   # (Linux/macOS)
hello.exe # (Windows)
```

Q4 **What are the main input/output operations in C++? Provide examples.**

🔹 Main I/O Operations in C++

1. Input Operation – `cin`

- Used to **take input** from the user.
- Comes from the **standard input stream** (keyboard).

2. Output Operation – `cout`

- Used to **display output** to the user.
- Sends data to the **standard output stream** (console).

Syntax

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

 main() {
    int number;
    cout << "Enter a number: ";  // Output to the console
    cin >> number;               // Input from the user
    cout << "You entered: " << number << endl;
    return 0;
}
```

🔹 Explanation

Operator	Description	Example
`<<`	**Insertion operator** (used with `cout`)	`cout << "Hello";`
`>>`	**Extraction operator** (used with `cin`)	`cin >> variable;`

🔹 More Examples

◆ Input and Output of Different Data Types

```
#include <iostream>
```

```
using namespace std;

 main() {
    string name;
    int age;

    cout << "Enter your name: ";
    cin >> name;

    cout << "Enter your age: ";
    cin >> age;

    cout << "Hello, " << name << ". You are " << age << " years old." <<
endl;

    return 0;
}
```

🔹 Note: `cin` reads only a single word for `string`. To read full lines, use `getline()`.

◆ Reading Full Line of Input

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

int main() {
    string sentence;

    cout << "Enter a sentence: ";
    getline(cin, sentence);  // Reads full line including spaces

    cout << "You entered: " << sentence << endl;

    return 0;
}
```

🔹 Common I/O Functions

Function	Purpose
`cin >>`	Input single word or value
`getline(cin, s)`	Input full line including spaces
`cout <<`	Display output
`endl`	Ends the line and flushes the output

Q5. **What are the different data types available in C++? Explain with examples**

1. Basic (Built-in) Data Types

Type	Description	Example
`int`	Integer values	`int age = 25;`
`float`	Single-precision floating-point number	`float temp = 36.6f;`
`double`	Double-precision floating-point number	`double pi = 3.14159;`
`char`	Single character	`char grade = 'A';`
`bool`	Boolean (true/false)	`bool isPassed = true;`
`void`	No value / no return type	`void showMessage();`

---

2. Derived Data Types

These are built from fundamental types:

Type	Description	Example
Array	Collection of elements of same type	`int marks[5] = {90, 85, 75, 88, 92};`
Pointer	Stores address of another variable	`int *ptr = &age;`
Function	A block of reusable code	`int add(int a, int b);`
Reference	Alias for another variable	`int &ref = age;`

---

3. User-defined Data Types

Created by the programmer to represent complex data:

Type	Description	Example
`struct`	Group of variables under one name	`struct Student { int id; string name; };`
`class`	Blueprint for objects in OOP	`class Car { public: int speed; };`

Type	Description	Example
`union`	Stores one value at a time from members	`union Data { int i; float f; };`
`enum`	User-defined constant set	`enum Color { RED, GREEN, BLUE };`

---

4. Constants / Literal Types

Type	Description	Example
Integer	Whole numbers	`100`, `-20`
Floating-point	Decimal numbers	`3.14`, `-0.001`
Character	Single character in quotes	`'a'`, `'Z'`
String literal	Sequence of characters	`"Hello"`
Boolean	`true` or `false`	`bool active = true;`

---

5. Type Modifiers

Used to change the size or sign of basic types:

Modifier	Description	Example
`signed`	Allows negative and positive values	`signed int a = -10;`
`unsigned`	Only positive values (more range)	`unsigned int u = 10;`
`short`	Smaller range than `int`	`short s = 32767;`
`long`	Larger range than `int`	`long l = 100000L;`
`long long`	Even larger integer range	`long long ll = 1e18;`

Example: `unsigned long long int population = 7800000000;`

🎯 Example Code

```
#include <iostream>
```

```
using namespace std;

 main() {
    int age = 25;
    float height = 5.9f;
    double pi = 3.1415926535;
    char grade = 'A';
    bool passed = true;

    cout << "Age: " << age << endl;
    cout << "Height: " << height << " ft" << endl;
    cout << "Value of Pi: " << pi << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << passed << endl;

    return 0;
}
```

Q6. **Explain the difference between implicit and explicit type conversion in C++.**

1. Implicit Type Conversion (Type Promotion)

◈ Definition:

Implicit conversion is **automatically performed by the compiler** when operands of different types are used in an expression.

◈ Characteristics:

- Happens **without programmer intervention**.
- Also called **automatic type conversion**.
- Generally converts to the **higher or more precise** data type to avoid data loss.

2. Explicit Type Conversion (Type Casting)

◈ Definition:

Explicit conversion is **manually specified by the programmer** to convert a value from one type to another.

◈ Characteristics:

- Requires **type casting syntax**.
- Used when you want to **override the default type conversion**.
- Can lead to **data loss** if done incorrectly (e.g., converting `float` to `int`).

Q-7 **What are the different types of operators in C++? Provide examples of each.**

✓ 1. Arithmetic Operators

Used to perform **basic mathematical operations**.

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (remainder)	`a % b`

```
cpp
CopyEdit
int a = 10, b = 3;
cout << a + b << " " << a % b;
```

✓ 2. Relational (Comparison) Operators

Used to **compare** two values and return a boolean result (`true` or `false`).

Operator	Description	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater or equal	`a >= b`
`<=`	Less or equal	`a <= b`

```
cpp
CopyEdit
cout << (a > b);  // Output: 1 (true)
```

✅ 3. Logical Operators

Used to combine or invert **boolean expressions**.

Operator	Description	Example
`&&`	Logical AND	`(a > 5 && b < 5)`
`		`
`!`	Logical NOT	`!(a > b)`

```
cpp
CopyEdit
bool result = (a > 5 && b < 5);
```

---

4. Assignment Operators

Used to **assign values** to variables.

Operator	Description	Example
`=`	Assign	`a = 10`
`+=`	Add and assign	`a += 5`
`-=`	Subtract and assign	`a -= 3`
`*=`	Multiply and assign	`a *= 2`
`/=`	Divide and assign	`a /= 2`
`%=`	Modulus and assign	`a %= 2`

---

5. Increment and Decrement Operators

Used to **increase or decrease** a variable's value by 1.

Operator	Description	Example
`++`	Increment	`a++`, `++a`
`--`	Decrement	`a--`, `--a`

```
int x = 5;
cout << ++x;   // Pre-increment
cout << x--;  // Post-decrement
```

---

6. Bitwise Operators

Operate on **individual bits** of data.

Operator	Description	Example
`&`	AND	`a & b`
`	`	OR
`^`	XOR	`a ^ b`
`~`	NOT	`~a`
`<<`	Left shift	`a << 1`
`>>`	Right shift	`a >> 1`

---

7. Conditional (Ternary) Operator

Short form of `if-else`.

Syntax	**`condition ? expr1 : expr2`**
Example	`int result = (a > b) ? a : b;`

---

8. size of Operator

Returns the **size (in bytes)** of a data type or variable.

```
cpp
CopyEdit
cout << sizeof(int);  // Output: 4 (usually)
```

---

9. Type Cast Operator

Used to convert data from one type to another.

```
float f = 3.14;
int i = (int)f;  // C-style
int j = static_cast<int>(f); // C++ style
```

10. Scope Resolution Operator

Used to access a **global variable** or class member scope.

```
int x = 10;

void show() {
    int x = 5;
    cout << ::x;  // Access global x
}
```

Q8 **Explain the purpose and use of constants and literals in C++.**

1. Constants in C++

Constants are used to **define values that must not change** once assigned. This ensures data integrity and prevents accidental modification.

2. Literals in C++

Literals are **fixed values written directly into the code**. They're the actual data values assigned to constants or variables.

Q9 **What are conditional statements in C++? Explain the `if-else` and `switch` statements**

In C, programs can choose which part of the code to execute based on some condition. This ability is called **decision making** and the statements used for it are called **conditional statements.** These statements evaluate one or more conditions and make the decision whether to execute a block of code or not.

1)If.. statement
The if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

2)If….else statement
The **if** statement alone tells us that if a condition is true, it will execute a block of statements and if the condition is false, it won't. But what if we want to do something else when the condition is false? Here comes the C **else** statement. We can use the **else** statement with the **if** statement to execute a block of code when the condition is false. The if-else statement consists of two blocks, one for false expression and one for true expression.

3) Switch… statement
The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Q10. **What is the difference between `for`, `while`, and `do-while` loops in C++?**

1) For Loop

- The `for` loop is used when you know in advance how many times you want to execute the block of code.
- It iterates over a sequence (e.g., a list, tuple, string, or range) and executes the block of code for each item in the sequence.
- The loop variable (`variable`) takes the value of each item in the sequence during each iteration.

2) While Loop

- The `while` loop is used when you don't know in advance how many times you want to execute the block of code. It continues to execute as long as the specified condition is true.
- It's important to make sure that the condition eventually becomes false; otherwise, the loop will run indefinitely, resulting in an infinite loop.

3) do-while Loop

- The do-while loop is similar to the while loop, but with one key difference: it guarantees that the block of code will execute at least once before checking the condition.
- This makes it useful when you want to ensure that a certain task is performed before evaluating a condition for continuation.
- The loop continues to execute as long as the specified condition is true after the first execution. It's crucial to ensure that the condition eventually becomes false to prevent the loop from running indefinitely, leading to an infinite loop.

Q11 **How are `break` and `continue` statements used in loops? Provide examples.**

1) Break Statement

A break statement is used when we want to terminate the running loop whenever any particular condition occurs. Whenever a break statement occurs loop breaks and stops executing.
For example: If we have a **variable curr=0** and in a **while loop** we are incrementing it by one each time and printing it. Now we want to stop whenever we get **curr=5.** So we can write a break statement when we get **curr=5.**

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

main()
{
    int curr = 0;
      // Loop till curr < 10
    while (curr < 10) {
        cout << curr << endl;
        curr++;
          // If curr = 5, break out of the loop
        if (curr == 5) {
            break;
        }
    }
    return 0;
}
```

Output:

```
0

1

2

3

4
```

2) Continue Statement

On the other side continue statement is used when we have to skip a particular iteration. Whenever we write continue statement the whole code after that statement is skipped and loop will go for next iteration.
For example: We have to print numbers form **1 to 5** but skip **3**.So we can use a **for loop** which will print the number and when i=3 we will continue.

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

main()
{
    // Loop till i <= 5
    for (int i = 0; i <= 5; ++i) {
        // If i = 3, then continue to the next iteration
        if (i == 3) {
            continue;
        }
        cout << i << endl;
    }
    return 0;
}
```

Output:

```
0

1

2

4

5
```

Q12. **Explain nested control structures with an example.**

Nested control structures refer to **placing one control structure inside another**, such as:

- An `if` inside another `if`
- A `loop` inside another `loop`
- An `if` inside a `loop`, or vice versa

This allows you to **handle more complex logic**, such as multiple levels of conditions or repetitions.

🔹 Types of Nested Control Structures

1. **Nested if-else**
2. **Nested loops (`for`, `while`, `do-while`)**
3. **Loops inside conditionals or vice versa**

Example 1: Nested `if-else`

```
#include <iostream>
```

```
using namespace std;

int main() {
    int number = 15;

    if (number > 0) {
        if (number % 2 == 0) {
            cout << "Positive even number";
        } else {
            cout << "Positive odd number";
        }
    } else {
        cout << "Non-positive number";
    }

    return 0;
}
```

Output:

Positive `odd` number

Example 2: Nested `for` Loop

Commonly used in **matrix** or **pattern printing**.

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

int main() {
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 5; j++) {
            cout << "* ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
* * * * *
* * * * *
* * * * *
```

Example 3: `if` Inside a `for` Loop

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

```
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            cout << i << " is even" << endl;
        } else {
            cout << i << " is odd" << endl;
        }
    }

    return 0;
}
```

Q13. **What is a function in C++? Explain the concept of function declaration, definition, and calling.**

A Function is a reusable block of code designed to perform a specific task. It helps break large programs into smaller, logical parts. Functions make code cleaner, easier to understand, and more maintainable.

Function Declaration introduces the function to the compiler. It tells the compiler the return type, name, and parameters but does not include the body. It ends with a semicolon

```
// Declaration
int add(int, int);
```

Function Definition provides the actual implementation of the function.

```
int add(int a, int b) {
    return a + b;
}
```

Once a function is defined, you can use it by simply **calling** its name followed by parentheses. This tells the program to execute the code inside that function.

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

// Function with no parameters and no return value
void greet() {
    cout << "Welcome to C++ Programming!" << endl;
}

// Function with parameters and a return value
int multiply(int a, int b) {
    return a * b;
}

int main() {
    // Calling the greet function
    greet();

    // Calling the multiply function
    int result = multiply(4, 5);
    cout << "Multiplication result: " << result << endl;

    return 0;
}
```

Output:

```
Welcome to C++ Programming!

Multiplication result: 20
```

Q14. What is the scope of variables in C++? Differentiate between local and global scope.

Now that you understand how functions work, it is important to learn how variables act inside and outside of functions.

In C++, variables are only accessible inside the region they are created. This is called **scope**.

Local variables are declared within a specific block of code, such as a function or method, and have limited scope and lifetime, existing only within that block. Global variables, on the other hand, are declared outside of any function and can be accessed from any part of the program, persisting throughout its execution.

Local Variables:

- Local variables are declared within a specific block of code, such as within a function or a loop.
- They are only accessible within the block in which they are declared.
- Once the block of code in which they are declared exits, the memory allocated to these variables is released, and they are no longer accessible.
- Local variables can have the same name as variables in other blocks without conflict because their scope is limited to the block in which they are declared.
- They are typically used for temporary storage or only relevant data within a specific context.

Global Variables:

- Global variables are declared outside of any function or block of code, usually at the top of a program or in a separate file.
- They are accessible from any part of the program, including within functions, loops, or other blocks of code.
- Global variables retain their value throughout the lifetime of the program unless explicitly modified or reset.

- Due to their accessibility from anywhere in the program, global variables can introduce unintended side effects and make it harder to understand and debug code, especially in larger programs.
- They are typically used for values that need to be accessed and modified by multiple parts of the program.

Difference between Local Variable and Global variables:

Aspect	Local Variables	Global Variables
Scope	Limited to the block of code	Accessible throughout the program
Declaration	Typically within functions or specific blocks	Outside of any function or block
Access	Accessible only within the block where they are declared	Accessible from any part of the program
Lifetime	Created when the block is entered and destroyed when it exits	Retain their value throughout the lifetime of the program
Name conflicts	Can have the same name as variables in other blocks	Should be used carefully to avoid unintended side effects
Usage	Temporary storage, specific to a block of code	Values that need to be accessed and modified by multiple parts of the program

Q15. **Explain recursion in C++ with an example.**

Recursion is a programming technique where a function calls itself repeatedly until a specific base condition is met. A function that performs such self-calling behavior is known as a **recursive function**, and each instance of the function calling itself is called a **recursive call**.

Example: Factorial Using Recursion

The **factorial** of a number `n` (written as `n!`) is:

```
n! = n × (n-1) × (n-2) × ... × 1
```

With:

0! = 1 (Base case)

C++ Code:

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

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0) {
        return 1; // Base case
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}

 main() {
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;

    int result = factorial(number);
    cout << "Factorial of " << number << " is " << result << endl;

    return 0;
}
```

How it works (for input 4):

```
factorial(4)
→ 4 * factorial(3)
→ 4 * (3 * factorial(2))
→ 4 * (3 * (2 * factorial(1)))
→ 4 * (3 * (2 * (1 * factorial(0))))
→ 4 * (3 * (2 * (1 * 1)))
→ 24
```

Q16. **What are function prototypes in C++? Why are they used?**

In C++, **function prototype** is a function declaration that tells the compiler about the name of the function, its return type and the number and type of parameters. With this information, the compiler ensures that function calls can be linked to the function definition, even if the definition is not available at the time of call.

✅ **Allows calling functions before they are defined**

- Without a prototype, the compiler won't know the function signature when it's called.

✅ **Supports modular programming**

- Functions can be declared in headers and defined in source files.

✅ **Enables type checking**

- The compiler checks if the function is called with correct number and types of arguments.

✅ **Improves readability and organization**

- Keeps code clean by separating interface (prototype) from implementation (definition).

Q17. **What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.**

array is a collection of elements of the same type placed in contiguous memory locations. It allows you to store multiple values under a single name and access them using an index.

Basis	**One Dimension Array**	**Two Dimension Array**
Definition	Store a single list of the element of a similar data type.	Store a 'list of lists' of the element of a similar data type.
Representation	Represent multiple data items as a list.	Represent multiple data items as a table consisting of rows and columns.

Declaration	The declaration varies for different programming language: 1. For C++, ***datatype variable_name[row]*** 2. For Java, ***datatype [] variable_name= new datatype[row]***	The declaration varies for different programming language: 1. For C++, ***datatype variable_name[row][column]*** 2. For Java, ***datatype [][] variable_name= new datatype[row][column]***
Dimension	One	Two
Size(bytes)	size of(datatype of the variable of the array) * size of the array	size of(datatype of the variable of the array)* the number of rows* the number of columns.
Address calculation.	Address of a[index] is equal to (base Address+ Size of each element of array * index).	Address of a[i][j] can be calculated in two ways row-major and column-major 1. **Column Major:** Base Address + Size of each element (number of rows(j-lower bound of the column)+(i-lower bound of the rows)) 2. **Row Major:** Base Address + Size of each element (number of columns(i-lower bound of the row)+(j-lower bound of the column))

Example	int arr[5]; //an array with one row and five columns will be created.
	{a , b , c , d , e}
		int arr[2][5]; //an array with two rows and five columns will be created.
		a b c d e
		f g h i j

Q18. **Explain string handling in C++ with examples**

In C++, **strings** are sequences of characters that are used to store words and text. They are also used to store data, such as numbers and other types of information in the form of text. Strings are provided by **<string>** header file in the form of **std::string** class.

C++ is a superset of C language, so it also inherits the way in which we used to create strings in C. In C, strings were nothing, but an array of characters terminated by a NULL character **'\0'**. They were created as:

```
char str[] = "Hello";
```

Function	**Description**
`strcpy()`	Copy one string to another
`strcat()`	Concatenate two strings
`strlen()`	Find length of a string
`strcmp()`	Compare two strings