Module-3 Introduction to OOPS Programming

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

1. Structure & Organization

- **Procedural**: Divides programs into **functions or procedures** that operate on external data—typically following a top-down, linear flow.
- **OOP**: Organizes code around **objects**, each encapsulating both state (data) and behavior (methods). Objects are instances of classes forming a bottom-up design.

2. Data Access & Encapsulation

- **Procedural**: Data is often global or passed around; little to no protection/enforcement.
- **OOP**: Enforces **encapsulation**—objects hide internal data and expose only controlled interfaces via methods, supporting private, public, and protected access.

3. Reusability & Modularity

- **Procedural**: Offers modularity via functions but often lacks fine-grained reuse across different contexts.
- **OOP**: Promotes reusability through **inheritance**, **polymorphism**, and composition—allowing new classes to easily extend or modify behavior

4. Data Security

- **Procedural**: States are more exposed; change in one place can unintentionally affect code elsewhere.
- **OOP**: With encapsulation and access controls, OOP ensures stronger data protection and controlled modification paths.

5. Flexibility & Extensibility

- **Procedural**: As programs grow, adapting or extending them typically means modifying multiple procedural blocks—can get unwieldy .
- **OOP**: Easier to extend; you can subclass or override behaviors without touching existing code, which improves maintainability .

6. Polymorphism & Inheritance

- **Procedural**: No built-in support for these concepts.

- **OOP**: Offers **inheritance** (sharing behavior across classes) and **polymorphism** (treating different types uniformly)

7. Mindset & Ideal Use Cases

- **Procedural**: Think in terms of "steps to perform tasks" — good for simple, linear, small-to-medium scripts.
- **OOP**: Think in terms of "entities that own data and behavior" — ideal for modeling complex systems, GUIs, simulations, etc.

8. Performance Considerations

- **Procedural**: Usually faster, with minimal overhead—suitable for smaller, performance-critical tasks.
- **OOP**: Adds some overhead (object creation, method dispatch), but it's negligible for large systems where maintainability wins.

List and explain the main advantages of OOP over POP.

1.Modularity & Reusability

- OOP bundles data and methods into **self-contained objects**, making code modular and reusable across components or even different projects.
- **Inheritance** allows new classes to build upon existing ones, reducing redundant code and enhancing efficiency

2. Encapsulation & Data Hiding

- Encapsulation keeps an object's internal state private and exposes behavior only through methods, preventing unintended misuse and enhancing integrity.
- This ensures controlled access and reduces potential for bugs caused by global data manipulation

3. Abstraction

- OOP allows developers to work at a higher level by hiding internal complexity of classes and exposing only necessary interfaces.
- This simplifies design of complex systems by focusing on essential features and relationships

4. Polymorphism & Flexibility

- With polymorphism, objects of different classes can be treated uniformly via a common interface, enabling flexible, extensible code design.
- This allows you to swap object implementations seamlessly without altering calling code.

5. Improved Maintainability & Collaboration

- OOP's modular structure means changes made to one class typically don't affect others, making maintenance and debugging easier.
- Teams can work independently on different classes/modules without stepping on each other's code .

6. Scalability & Extensibility

- As systems grow, OOP supports scaling more naturally—new features can be added via subclassing or composition without major refactoring.
- OOP keeps large codebases organized and easier to evolve over time .

7. Real-World Modeling

- By using classes that mirror real-world entities (e.g. Customer, Order, Motor), OOP provides a more intuitive and maintainable design.
- This approach eases reasoning and aligns code with domain concepts.

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

1. Install a Compiler 🛠️

Windows

- **MinGW-w64**: Download from the official MinGW-w64 installer site, select GCC/G++ packages, and add the ...\bin folder to your system **PATH**.
- **MSVC (Microsoft Visual C++)**: Install Visual Studio Community edition and choose the "Desktop development with C++" workload.

macOS

- Install Xcode Command Line Tools via:

xcode-select –install

which includes the Clang compiler

Optionally, use **Homebrew**:

brew install gcc

Linux

- Use the distro package manager:

sudo apt update && sudo apt install build-essential g++

for Debian/Ubuntu, or:

sudo dnf install gcc-c++

2. Verify Compiler Installation

Run in your terminal or command prompt:

g++ --version

or clang++ --version / cl (for MSVC)

Ensure you see version output or the command recognises.

3. Choose an IDE or Editor

Here are popular options:

- **Visual Studio** (Windows) – powerhouse IDE with advanced IntelliSense and robust debugging.
- **Visual Studio Code** (all platforms) – lightweight, extensible, with Microsoft's "C/C++" extension and Code Runner.
- **CLion**, **Code::Blocks**, **Qt Creator**, **Dev-C++**, **CodeLite**, **Eclipse CDT**.

Reddit insight:

"For Windows, I'm aware there's WSL2… MSYS2… vim with language server… cmake… packages: vcpkg".

4. (Optional) Install Package Manager & Build Tools

- **CMake**: Essential for cross-platform building; integrates with IDEs.
- **vcpkg** (Microsoft) or **Conan**: Simplify dependency management.
- On Windows, consider **MSYS2** + pacman for a Unix-like toolchain.

5. Configure Your Editor

- **VS Code**:
    - Install the **C/C++** extension by Microsoft.
    - (Optional) Add **CMake Tools**, **clangd**, **clang-format**.
    - Configure tasks via tasks.json to automate g++ "${file}" -o "${fileBasenameNoExtension}".
    - Use integrated terminal to compile/run.

6. Test with "Hello, World!"

Create hello.cpp containing:

cpp

```
#include <iostream>

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

Build and run:

g++ hello.cpp -o hello

./hello

—or use appropriate commands for MSVC or another compiler.

7. (Optional) Setup Version Control

Install **Git** for source control:

- Windows: from git-scm.com
- Most systems: preinstalled or via package manager
  Integrate Git in your IDE (VS / VS Code)

8. (Optional) Add Debugger & Linter

- **GDB** or **LLDB** for Linux/macOS debugging.
- **MSVC debugger** on Windows.
- Add linting/formatting tools:

- clang-format, clang-tidy, clangd — especially useful in VS Code or Vim

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

1. Console Input & Output

std::cout – Standard Output

Usage: Displays text/numbers on screen using the insertion operator <<.

Example:

```
#include <iostream>

int main() {

   int age = 30;

   std::cout << "Age: " << age << std::endl;

   return 0;

}
```

2. Error & Log Streams

std::cerr: Unbuffered output for immediate error reporting.

```
std::cerr << "Error occurred\n";
```

3. File Input & Output

Include <fstream> and use:

- **std::ifstream** for reading files,
- **std::ofstream** for writing files,
- **std::fstream** for both.

Writing to a file:

```
#include <fstream>
std::ofstream f("output.txt");
if (f.is_open()) {
   f << "Hello, file!" << std::endl;
   f.close();
}
```

Reading from a file:

```
#include <fstream>
#include <string>
std::ifstream in("input.txt");
std::string line;
while (std::getline(in, line)) {
   std::cout << line << "\n";
}
in.close();
```

4. Low-Level Stream Methods

These apply to console or file streams:

- cin.get(char&), cin.getline(...) – read single chars or lines.
- cin.read(buffer, n), cout.write(buffer, n) – binary I/O.
- cin.ignore(n), cin.putback(ch) – control buffer behavior.
- cin.gcount() – bytes read by get/read.

5. Formatting I/O with Manipulators

Use <iomanip> for formatting control methods:

- std::setw(n), std::setprecision(n), std::fixed, std::hex, std::dec, std::endl, and more.

Example:

```
#include <iostream>
#include <iomanip>
std::cout << std::fixed << std::setprecision(2)
        << "PI = " << 3.14159 << "\n";
```

6. String Streams (<sstream>)

- **std::ostringstream**: Build formatted strings.
- **std::istringstream**: Parse strings as if they were input streams.

Example:

```
#include <sstream>
std::ostringstream oss;
oss << "Result: " << 42;
std::string s = oss.str();
```

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

Here are the **basic built-in data types in C++** with **simple usage examples**:

🧱 1. Integer Types (int, short, long, …)

Store whole numbers (no decimals).

```
int age = 25;

short s = 32000;
```

long population = 8000000000L;

unsigned int u = 100u; // only non-negative

int typically 4 bytes, range –2.1B to +2.1 B

unsigned int 0 to ~4.3 B

2. Floating-Point Types (float, double, long double)

Store decimal values.

float pi = 3.14f;

double salary = 12345.67;

long double big = 1.234567890123456789L;

- float ≈ 6–7 decimal digits, 4 bytes
- double ≈ 15 digits, 8 bytes

3. Character Type (char)

- Stores a single character.

char grade = 'A';

char digit = '7';

Typically 1 byte, holds ASCII

4. Wide Character (wchar_t)

Used for larger character sets (e.g., Unicode).

wchar_t w = L'Ω';

Size 2–4 bytes

5. Boolean (bool)

Stores true or false.

bool isLoggedIn = false;

Typically 1 byte

6. Void (void)

Represents “no value”; used mainly for functions:

```
void doSomething() {
   // does not return anything
}
```

Example Program

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

int main() {
   int age = 30;
   float height = 5.9f;
   double weight = 70.5;
   char symbol = '#';
   wchar_t omega = L'Ω';
   bool isAdult = (age >= 18);

   cout << "Age: " << age << "\n"
        << "Height: " << height << "\n"
        << "Symbol: " << symbol << "\n"
        << "Omega: " << omega << "\n"
        << "Is adult? " << boolalpha << isAdult << "\n";

   return 0;
}
```

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

In C++, **type conversion** can be either **implicit** (automatic) or **explicit** (manual). Here's a breakdown:

Implicit Type Conversion (Coercion)

- **What it is**: The compiler automatically converts one type to another when it deems it safe or necessary.
- **When it happens**:
    1. Mixed-type expressions, e.g., int + double
    2. Assignments/initializations to a compatible type
    3. Passing arguments to functions expecting a different but compatible type

Explicit Type Conversion (Casting)

- **What it is**: Conversion performed manually by the developer.
- **Why use it**: To avoid ambiguity, data loss, or unintended behavior.
- **Forms in C++**:
    1. **C-style cast**: (int)someDouble
    2. **Function-style cast**: int(someDouble)
    3. **C++ named casts**:
        - static_cast<int>(x)
        - dynamic_cast<T*>(p)
        - const_cast<T>(expr)
        - reinterpret_cast<T>(ptr)

What are the different types of operators in C++? Provide examples of each.

1. Arithmetic Operators

Operate on numeric types.

Arithmetic

+ - * / % ++ --

```
int a = 8, b = 3;

std::cout << a + b << "\n";  // 11
```

```
std::cout << a - b << "\n";  // 5

std::cout << a * b << "\n";  // 24

std::cout << a / b << "\n";  // 2 (integer division)

std::cout << a % b << "\n";  // 2 (remainder)


a++;  // increment a → 9

b--;  // decrement b → 2
```

2. Relational (Comparison) Operators

Compare values; return true or false.

Relational

== != < > <= >=

```
int x = 10, y = 5;

bool eq  = (x == y);  // false

bool ne  = (x != y);  // true

bool lt  = (x < y);   // false

bool gt  = (x > y);   // true

bool le  = (x <= y);  // false

bool ge  = (x >= y);  // true
```

3. Logical Operators

Operate on boolean expressions with short-circuit behavior.

Logical

`&&

```
bool a = true, b = false;
```

```
bool and_op = a && b;  // false

bool or_op  = a || b;  // true

bool not_a  = !a;      // false
```

4. Bitwise Operators

Work on individual bits of integer types.

```
unsigned int m = 6;  // 0b0110

unsigned int n = 3;  // 0b0011

std::cout << (m & n);   // 2 (0b0010)

std::cout << (m | n);   // 7 (0b0111)

std::cout << (m ^ n);   // 5 (0b0101)

std::cout << (~m);      // bitwise NOT

std::cout << (m << 1);  // left shift → 12

std::cout << (n >> 1);  // right shift → 1
```

5. Assignment Operators

Assign and optionally combine.

```
int v = 10;

v += 5;  // v = 15

v -= 3;  // v = 12

v *= 2;  // v = 24

v /= 4;  // v = 6

v %= 4;  // v = 2

v <<= 1; // left shift: v = 4

v &= 3;  // bitwise AND assign: v = 0
```

6. Ternary (Conditional) Operator

Compact if-else in one line.

```
int a = 10, b = 20;
```

int mx = (a > b) ? a : b; // mx = 20

7. Miscellaneous Operators

int i = (1, 2, 3); // i = 3

- **Comma (,)**: Evaluates multiple expressions, returns the last.

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

C++, **constants** and **literals** are both about values that don't change, but they serve different purposes:

1. Literals

Literals are fixed values **written directly in your code**, like 42, 'A', or "Hello". They come in several types:

- **Integer literals**:

  int a = 42;   // decimal

  int b = 052;   // octal (leading 0)

  int c = 0x2A;   // hexadecimal prefix 0x :contentReference[oaicite:2]{index=2}

  Floating-point literals:

  double pi = 3.14;

  float f = 3.14f;

  long double ld = 3.14L;

  double e = 1.23e4; // scientific notation :contentReference[oaicite:3]{index=3}

  Character literals:

  char ch = 'A';

  char nl = '\n';   // escape sequence :contentReference[oaicite:4]{index=4}

  String literals:

const char* msg = "Hello, World!";

Boolean literals: true, false

2. Constants (const and #define)

Constants are **named values** whose contents cannot change after initialization:

- **Using const keyword**:

```
const int MAX = 100;

const double PI = 3.14159;
```

Benefits:

- Improves code readability
- Prevents accidental reassignment
- Enables compiler optimizations (e.g., constant folding)

What are conditional statements in C++? Explain the if-else and switch statements.

C++, *conditional statements* allow your program to make decisions and choose different execution paths based on conditions. The two primary forms are if-else and switch.

if-else Statements

Used to execute code based on boolean conditions, including complex expressions.

Syntax

```
if (condition1) {
   // executed if condition1 true
}
else if (condition2) {
   // executed if condition1 false and condition2 true
}
else {
   // executed if all above are false
}
```

Example

```
int x = 20;

if (x == 10)
```

```
    std::cout << "x is 10";
else if (x == 15)
    std::cout << "x is 15";
else if (x == 20)
    std::cout << "x is 20";
else
    std::cout << "x is something else";
```

Here, each condition is checked in turn; the first matching block runs, then the rest are skipped.

Features:

- Supports any expression: logical (&&, ||), comparisons, function calls, etc.
- Ideal for complex logic or range checks.
- Clean behavior—no fall-through as in switch.

switch Statements

- Optimized for checking *one variable/expression* against multiple **constant** cases.

Syntax

```
switch (expr) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // fallback code
```

```
}
```

Example

```
int day = 3;

switch (day) {
  case 1: std::cout << "Monday"; break;
  case 2: std::cout << "Tuesday"; break;
  case 3: std::cout << "Wednesday"; break;
  default: std::cout << "Invalid day";
}
```

Outputs "Wednesday".

Features:

- Checks one expression exactly once.
- Supports fall-through (cases execute sequentially if no break).
- MoreReadable and maintainable for multiple fixed options.
- Compilers can optimize to jump tables for speed.

When to Use Which?

- Use **if-else** when:
  - Evaluating ranges or logical conditions.
  - Working with multiple variables or complex logic.
- Use **switch** when:
  - Checking a single variable against many constant values.
  - You need clarity, brevity, and potential performance benefits.

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

In C++, three primary loop types allow you to execute a block of code repeatedly: **for**, **while**, and **do-while**. Here's how they differ and where each shines:

1. for Loop

- **Best when** you know the number of iterations in advance.
- **Syntax**:

```
for (init; condition; update) {
```

```
    // loop body
}
```

Structure: initialization, condition-check, then body, followed by update each iteration.

Example:

```
for (int i = 0; i < 5; ++i) {
    std::cout << i << "\n";  // prints 0–4
}
```

- **Use case**: Simple counting loops, indexing arrays. Combines control logic in one line.
- **Scope note**: i is only visible inside the loop.

2. while Loop

- **Best when** you don't know how many times you'll loop ahead of time.
- **Syntax**:

```
while (condition) {
    // loop body
}
```

Flow: evaluates condition *before* executing the body (entry-controlled).

Example:

```
int x = 0;
while (x < 5) {
    std::cout << x << "\n";
    ++x;
}
```

3. do-while Loop

- **Best when** you need the loop body to run at least once.

Syntax:

```
do {
   // loop body
} while (condition);
```

Flow: executes the body *first*, then checks the condition (exit-controlled).

Example:

```
int count = 5;
do {
   std::cout << "Count: " << count << "\n";
   ++count;
} while (count < 5);
```

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

C++, **break** and **continue** are control statements used within loops (and break also in switch) to alter the normal loop flow:

break

- **Purpose**: Immediately exits the **innermost loop** (or switch), skipping any remaining iterations.
- **Usage examples**:

for loop example

```
for (int i = 0; i < 10; ++i) {
   if (i == 3) {
      break;
   }
   std::cout << i << " ";  // prints 0 1 2
}
```

Stops the loop entirely when i == 3.

while loop example

```
int i = 0;

while (i < 10) {

   if (i == 5) {

      break;

   }

   std::cout << i << " ";  // prints 0 1 2 3 4

   ++i;

}
```

Statement	Effect	Use Case
break	Exit the entire loop immediately	Stop processing once a condition is met
continue	Skip to next iteration, not exit loop	Filter or skip specific cases but continue looping

Explain nested control structures with an example.

C++, **nested control structures** mean placing one control statement *inside* another. This can be nested if statements, loops, or even a mix of both. Let's explore two common scenarios:

1. Nested if-else Statements

You can have conditions inside conditions. For instance, check if a number is even, and if so, whether it's also divisible by 3:

```
#include <iostream>

using namespace std;



int main() {

   int n = /* some integer */;

   if (n % 2 == 0) {

      // Outer condition: is it even?
```

```
        if (n % 3 == 0) {
            cout << "Divisible by 2 and 3";
        } else {
            cout << "Divisible by 2 but not 3";
        }
    } else {
        cout << "Not divisible by 2";
    }
    return 0;
}
```

Here, the **inner if runs only if the outer condition is true**. This structure is especially useful for multilayered decisions

2. Nested Loops

Loops can be placed inside other loops, creating multi-level iteration — often used for matrices, tables, patterns, etc.

a) Nested for Loops:

Print a 3×3 multiplication table:

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

int main() {
    for (int i = 1; i <= 3; ++i) {          // outer loop: rows
        for (int j = 1; j <= 3; ++j) {      // inner loop: columns
            cout << (i * j) << " ";
        }
```

```
        cout << endl;

    }

    return 0;

}
```

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

What is a Function?

A **function** in C++ is a **named block of code** that performs a specific task and can be reused. Think of it as a mini-program within your program. You give it inputs (if needed), it runs, and may return a result.

1. Function Declaration (Prototype)

- **Purpose:** Informs the compiler about the function's **name**, **return type**, and **parameter types**.
- **Where it's used:** Before main() or in header files, especially if the definition appears later or in another file.

Syntax:

```
return_type functionName(paramType1 param1, paramType2 param2, ...);
```

2. Function Definition

- **Purpose:** Provides the actual **body**—the implementation—of the function.
- **Must match** the declaration (same return type, name, and parameters).

Syntax:

```
return_type functionName(paramType1 param1, paramType2 param2) {

    // function body

}
```

3. Function Call

- **Purpose:** Executes the function's body.
- Use it by writing the function's name followed by **arguments** in parentheses.

Syntax:

result = functionName(arg1, arg2);

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

In C++, the **scope** of a variable determines where in the code that variable can be accessed. Here's a clear breakdown of **local** vs. **global scope**:

Local Scope

- Variables declared **inside a function or block** (between {}) are **local variables**.
- They exist only within that block and are destroyed when the block ends.
- They **cannot be accessed** outside their defining block.

Example:

```
#include <iostream>

using namespace std;



int main() {

   int x = 10;  // local to main()

   {

      int y = 20;  // local to inner block

      cout << x << ", " << y << "\n";  // OK: prints 10, 20

   }

   // cout << y;  // Error: y is out of scope

   return 0;

}
```

Global Scope

- Variables declared **outside of all functions**, typically at the top, have **global scope**.
- They are accessible **anywhere** in the same file (and even across files if using extern).
- **Lifetime**: the entire runtime of the program.

Example:

```
#include <iostream>

using namespace std;


int g = 100;  // global variable


void show() {

   cout << g << "\n";  // can access global g

}


int main() {

   show();          // prints 100

   cout << g << "\n";  // prints 100

   return 0;

}
```

Explain recursion in C++ with an example.

Recursion in C++ is a powerful technique where a function **calls itself** to solve a larger problem by breaking it down into smaller, similar subproblems. It requires two essential parts:

1. Understanding Recursion

A **recursive function** includes:

1. **Base Case**: A condition that stops recursion—prevents infinite calls and stack overflow.
2. **Recursive Case**: The part where the function calls itself with a smaller or simpler input.

This pattern eventually leads to reaching the base case, after which the recursive calls **unwind** to produce the final result

2. Example: Factorial Using Recursion

Mathematical definition:

$n! = n \times (n - 1)!$

$0! = 1, 1! = 1$

Sample C++ Implementation:

```
#include <iostream>

using namespace std;


unsigned long factorial(int n) {

   if (n <= 1)              // Base case: 0 or 1

      return 1;

   else                   // Recursive case

      return n * factorial(n - 1);

}


int main() {

   int num = 5;

   cout << "Factorial of " << num << " = "

        << factorial(num) << endl;

   return 0;

}
```

For n = 5, it computes:
5 * factorial(4)
→ 5 * (4 * factorial(3))
…until factorial(1), which returns 1. Then the recursive stack unwinds, multiplying back to produce 120

3. How Recursion Works Internally

- Each recursive call allocates a new stack frame with its own parameters and local variables.
- Calls continue until reaching the **base case**, then results are returned step-by-step back through the call stack ("unwinding") .
- Avoid missing the base case to prevent infinite recursion and eventual program crash.

🔍 4. Key Characteristics & Trade-offs

Feature	Details
Clarity	Recursion often simplifies complex logic (e.g., tree traversal)
Overhead	Each call increases stack usage—O(n) space/time for factorial
Use Cases	Ideal for divide-and-conquer problems (factorials, Fibonacci, QuickSort, tree algorithms)

What are function prototypes in C++? Why are they used?

C++, a **function prototype** (or declaration) simply tells the compiler **what** a function looks like—its **name**, **return type**, and the **number and types of its parameters**—without providing the body. It ends with a semicolon and can appear before main() in a source file or inside a header file. For example:

```
int add(int a, int b);  // Function prototype
```

Why Use Function Prototypes?

1. **Enable calling functions before their definitions**
   By declaring the prototype early, you can call that function—such as add(5, 3)—in main(), even if the full implementation is written later in the file or in a separate source file.
2. **Type safety and early error checking**
   The compiler uses the prototype to verify at compile-time that calls to the function have the correct number and types of arguments, and that the return value is used correctly.
3. **Modular code organization & separate compilation**
   In multi-file projects, prototypes are placed in header files (.h/.hpp) and included where needed, enabling each translation unit to know about external functions without needing the full definitions.
4. **Support for recursion and mutual function calls**
   When two functions call each other, forward declarations (prototypes) ensure the compiler recognizes each before use.

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

C++, an **array** is a fixed-size collection of **elements** of the same data type stored in **contiguous memory** (i.e., one right after the other). You access elements using indices, starting from zero

Single-Dimensional Arrays

A **single-dimensional (1D) array** is essentially a linear list of items.

Declaration & Initialization:

```
int nums[5];              // declares an array of 5 ints

int nums2[5] = {10,20,30,40,50}; // initializes at declaration
```

Access & Modify:

```
int x = nums2[2];       // x = 30

nums2[3] = 60;          // third index updated to 60
```

Use Case:

- Ideal when you need to handle a simple sequence, like a list of scores or IDs.
- Single for or while loops suffice for iteration

Multi-Dimensional Arrays

- A **multi-dimensional array** is essentially an "array of arrays." The most common form is a **two-dimensional (2D) array**, which you can visualize as a table.
- **Two-Dimensional Declaration & Initialization:**
  ```
  int matrix[2][3]; // 2 rows, 3 columns
  int matrix2[2][2] = { {1, 2}, {3, 4} };
  ```

Common Use Case:

Like images, tables, or grids, often used in scientific or matrix computations.

Access & Modify:

```
int val = matrix2[1][1];  // second row, second column → 4

matrix2[0][2] = 7;       // first row, third column
```

Explain string handling in C++ with examples.

Below is a comprehensive overview of **string handling in C++**, covering both C-style strings and the modern, safer std::string (from the C++ Standard Library).

1. C-style strings (char arrays, null-terminated)

These are inherited from C: arrays of char ending with '\0'.

```
char str1[] = "Hello";  // 6 chars: 'H','e','l','l','o','\0'

char str2[10];

cin >> str2;          // read up to first space

cin.get(str2, 10);     // read with spaces
```

Common functions (from <cstring> / <string.h>):

- strcpy, strncpy — copy strings
- strcat, strncat — append strings
- strlen — length (not including \0)
- strcmp — compare strings
  Be careful: they don't check bounds—buffer overflow is possible

3. std::string (preferred in C++)

Defined in <string>, a class type with automatic memory management and rich functionality

2.1 Initialization

```
std::string s1 = "Hello";

std::string s2("World!");

std::string s3 = s1 + " " + s2; // concatenation
```

2.2 Accessing characters

```
char c0 = s3[0];     // no bounds check

char c1 = s3.at(1);  // throws if out-of-range

char f = s3.front(), b = s3.back();
```

2.3 Size and emptiness

```
size_t len = s3.length();

size_t sz = s3.size();       // same as length()

bool isEmpty = s3.empty();
```

2.4 Concatenation & appending

s3 += "!!!";

s3.append(" More");

2.5 Substrings and search

size_t pos = s3.find("lo");

if (pos != std::string::npos) { ... }

std::string sub = s3.substr(1, 3);

Reddit uses it all the time:

“if (str.find("wallet") != string::npos)” means “if substring ‘wallet’ is found”

2.6 Replace, insert, erase

s3.insert(5, "INSERTED");

s3.erase(2, 4);

s3.replace(pos, 6, "text");

2.7 Comparison

if (s1 == s2) { ... }

int cmp = s1.compare(s2); // 0 = equal, <0 or >0 otherwise

2.8 Conversion to C-string

const char* cstr = s3.c_str();

How are arrays initialized in C++? Provide examples of both 1D and 2D arrays

Here’s a well-structured guide on how arrays are initialized in C++, both for 1D and 2D cases:

1. One-Dimensional Arrays

✅ **Declaration & Full Initialization**

Specify size and initializer list:

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

- All elements explicitly set.
- Access with arr[i] (i from 0 to 4).

⏸️ Partial Initialization

Omitting some values automatically zero-initializes the rest:

int arr2[5] = {4, 7}; // equivalent to {4, 7, 0, 0, 0}

And you can zero-initialize the entire array concisely:

int arr3[5] = {0};   // all elements become 0

2. Two-Dimensional Arrays

🔢 Declaration

int mat[2][3];

Defines a matrix with 2 rows and 3 columns.

A) Full Nested Initialization

Use nested braces to match rows:

int mat1[2][3] = {

  {1, 2, 3},

  {4, 5, 6}

};

Here, mat1[0][*] = {1,2,3} and mat1[1][*] = {4,5,6}

B) Flat List Initialization

You can skip inner braces — elements fill in row-major order:

int mat2[2][3] = {1, 2, 3, 4, 5, 6};

C) Partial Initialization & Zero-Fill

Omit values to default the rest to zero:

int mat3[2][3] = { {5}, {7,8} };

// => mat3 = {{5,0,0}, {7,8,0}}

Explain string operations and functions in C++.

Here's a detailed guide to **string operations and functions in C++**, focusing on the modern std::string class and a few key C-style string utilities.

1. Overview of std::string Operations

The std::string class (in <string>) provides a rich set of **member functions**, **non-member utilities**, and stream/numeric conversion tools. Below are the most commonly used ones, with brief descriptions and usage examples. Most are documented in C++ references.

✅ **Basic Access & Inspection**

Basic Access & Inspection

- .size() / .length() — returns number of characters
- .empty() — true if string is empty.
- .max_size() — theoretical maximum length

```
std::string s = "hello";

std::cout << s.length();  // 5
```

Element Access

- s[i] — no bounds checking
- s.at(i) — throws out_of_range if invalid
- .front(), .back() — first and last character

```
char c = s.at(0);  // 'h'
```

Appending & Concatenation

- s += other or s = s + other — operator+ is overloaded
- .append(...) — add strings or characters
- .push_back(ch) — append single character
- .pop_back() — remove last character

```
s += " world";

s.append("!");

s.push_back('?');
```

Searching & Substrings

- .find(sub[, pos]), .rfind(...) — locate substring
- .find_first_of(...), .find_first_not_of(...), .find_last_of(...), .find_last_not_of(...) — search for character sets
- .substr(pos, count) — returns substring

```
if (s.find("world") != std::string::npos) { ... }

std::string word = s.substr(6, 5);  // "world"
```

Modifiers

- .clear() — empties string
- .erase(pos, len) or .erase(iterator range) — remove part of string
- .insert(pos, str) — insert content at position
- .replace(...) — replace substring region with new content
- .resize(n) — shrink or extend (pad with nulls)
- .swap(other) — efficient content swap
- .copy(buf, count, pos) — copy characters into raw buffer

Capacity Management

- .capacity() & .reserve(n) — manage storage space
- .shrink_to_fit() — release unused memory

```
s.reserve(100);
```

Comparisons

- s1 == s2, !=, <, > — lexicographical comparisons using overloaded operators
- .compare(other) — returns integer (<, =, >) indicating order

```
if (s == "hello") { ... }

int diff = s.compare("world");
```

I/O & Numeric Conversion

- std::getline(cin, s) — read full line into string
- operator<< / >> — stream insertion/extraction
- std::stoi, std::stol, std::stoll, std::stoul, std::stod, etc. — convert string to numeric types
- std::to_string(val) — convert number to string

```
std::string line;

std::getline(std::cin, line);

int x = std::stoi(line);
```

std::string y = std::to_string(3.14);

C++20/23 Enhancements

- .starts_with(prefix), .ends_with(suffix) in C++20
- .contains(sub) in C++23

if (s.starts_with("Hi")) { ... }

if (s.contains("C++")) { ... }

3. C-Style String Functions (for completeness)

Although not typically used in modern C++, C-style char* strings can be manipulated using <cstring> utilities:

- strlen, strcpy, strncpy, strcat, strncat, strcmp, strncmp
- strchr, strrchr, strstr, strspn, strcspn, strpbrk, strtok
- memcpy, memmove, memcmp, memset for buffer-level operations.

These require manual management of buffers and null terminations.

4. Quick Reference Table

Category	**Function(s)**	**Description**
Size & Capacity	size()/length(), capacity(), reserve(), etc.	String length and memory management
Access & Index	operator[], at(), front()/back()	Character access
Concatenation	+, append(), push_back()	Build or extend strings
Search & Substr	find(), rfind(), substr()	Locate and extract substrings
Modify	insert(), erase(), replace(), resize(), etc.	Edit string content
Compare	==, compare()	Lexicographical comparison
I/O & Conversions	getline(), >>, std::stoi(), std::to_string()	Input/output and number-string conversions
C-Style Tools	strcpy, strlen, etc.	Legacy low-level string operations

Explain the key concepts of Object-Oriented Programming (OOP).

1. Classes & Objects

- A **class** acts as a blueprint defining **state (attributes)** and **behavior (methods)** of a type of object.
- An **object** is an **instance of a class**, existing at runtime with its own state and functionality .

🔐 2. Encapsulation & Information Hiding

- **Encapsulation** combines data and methods into a single class and restricts direct access from outside, typically using access modifiers like private, protected, or public .
- **Information hiding** ensures internal details are hidden, exposing only a stable interface, making code maintenance safer and simpler.
- From a developer perspective:

"Encapsulation is hiding implementation such that the contract is not broken with the external consumer when changes are made."

🧩 3. Abstraction

- **Abstraction** focuses on exposing only essential features while hiding implementation complexity.
- Example: providing public methods while hiding internal workings. Users interact via a simple interface without needing to understand details.

🔁 4. Inheritance

- **Inheritance** allows one class (subclass/derived) to inherit properties and behavior from another (superclass/base) class, promoting code reusability and hierarchical design.
- Supports **method overriding**, where a subclass provides its own implementation of an inherited method .

🔄 5. Polymorphism

- **Polymorphism** permits objects of different classes to be treated as instances of a common superclass, enabling methods like draw() to work on multiple derived types.
- Includes static (compile-time) polymorphism like overloading, and runtime polymorphism via method overriding and dynamic dispatch.

📦 6. Modularity, Composition & Design Principles

- **Modularity**: Classes encapsulate functionality into reusable units, making systems clearer and easier to maintain.
- **Composition**: A class can include objects of other classes, modeling "has-a" relationships rather than relying on inheritance.
- **Coupling & Cohesion**: Good design strives for low coupling (minimal interdependencies) and high cohesion (well-scoped single responsibility).
- **Open–Closed Principle** (SOLID): Systems should be open for extension but closed for modification, enabling new behaviors without altering existing code.

What are classes and objects in C++? Provide an example.

What is a Class?

- A **class** is like a blueprint that defines:
  - **Data members** (attributes/state)
  - **Member functions** (behavior/methods)
- Defined using the class keyword, it doesn't allocate memory until objects are created

Syntax:

```
class Person {

public:

   std::string name;

   int age;

   void greet() {

      std::cout << "Hello, I'm " << name << "\n";

   }

};
```

What is an Object?

- An **object** is an **instance** of a class—it holds real data according to the class blueprint.
- Created by declaring a variable of the class type .

Example:

```
Person p1;

p1.name = "Alice";

p1.age = 30;

p1.greet();  // Outputs: Hello, I'm Alice
```

Example: Car Class

```
#include <iostream>

#include <string>
```

```
using namespace std;

class Car {
public:
    string brand;
    string model;
    int year;
    void displayInfo() {
        cout << year << " " << brand << " " << model << "\n";
    }
};

int main() {
    Car car1;
    car1.brand = "Toyota";
    car1.model = "Camry";
    car1.year = 2020;
    car1.displayInfo();  // Outputs: 2020 Toyota Camry

    Car car2;
    car2.brand = "Ford";
    car2.model = "Mustang";
    car2.year = 1969;
    car2.displayInfo();  // Outputs: 1969 Ford Mustang
```

```
    return 0;
}
```

What is inheritance in C++? Explain with an example.

Inheritance in C++ is a fundamental **OOP mechanism** that enables a class (**derived/child**) to inherit attributes and behaviors from another class (**base/parent**), promoting code reuse and logical hierarchies.

Key Concept

- **Base class**: Defines common data and functions.
- **Derived class**: Inherits from the base and can add or override features.

Syntax:

```
class Derived : public Base { /* ... */ };
```

Public inheritance means that public members of Base remain public in Derived

Simple Example

```
#include <iostream>

using namespace std;


// Base class

class Animal {

public:

    void eat() {

        cout << "I can eat!\n";

    }

    void sleep() {

        cout << "I can sleep!\n";

    }
```

```
};


// Derived class
class Dog : public Animal {
public:
    void bark() {
        cout << "I can bark! Woof woof!!\n";
    }
};


int main() {
    Dog dog1;
    dog1.eat();   // inherited
    dog1.sleep(); // inherited
    dog1.bark();  // own method
    return 0;
}
```

Why Inheritance?

1. **Code reuse**: Write common behavior in base class.
2. **Hierarchies**: Model relationships like "Dog is an Animal".
3. **Expandable**: Derived classes can add or override behavior.

🔹 **Inheritance Variants**

C++ supports:

- **Single inheritance** (one base): as shown above.
- **Multiple inheritance** (from more than one base):

Access Control in Inheritance

- public: public → public, protected → protected
- protected: public and protected both become protected
- private: public and protected both become private

By default, classes use **private** inheritance, and struct uses public\

What is encapsulation in C++? How is it achieved in classes?

Encapsulation in C++ is the practice of **bundling data (attributes) and methods (functions) together into a class** while **hiding internal details** and exposing a controlled interface. It helps protect the internal state of an object and prevent unintended misuse.

How is encapsulation achieved?

1. **Defining a class**: All related data and functions are grouped in a single unit.
2. **Access specifiers (private/public/protected)**:
    - private members are hidden from outside the class.
    - public methods serve as a safe interface to interact with hidden data.

3. **Using getter/setter methods**: These allow controlled read/write access, possibly with validation.

Encapsulation enforces **data hiding** and maintains **internal integrity**, preventing external code from directly manipulating an object's state.

Example:-

```
#include <iostream>

#include <string>

using namespace std;



class Employee {

private:

   int empId;

   string empName;

   double salary;  // hidden internal data



public:
```

```
    // Constructor
    Employee(int id, string name, double sal)
        : empId(id), empName(name), salary(sal) {}

    // Getters (accessors)
    int getId() const { return empId; }
    string getName() const { return empName; }
    double getSalary() const { return salary; }

    // Setter (mutator) with validation
    void setSalary(double sal) {
        if (sal >= 0)
            salary = sal;
        else
            cout << "Error: Salary must be non-negative" << endl;
    }
};

int main() {
    Employee emp(101, "John Doe", 5000.0);

    cout << "ID: " << emp.getId()
         << ", Name: " << emp.getName()
         << ", Salary: " << emp.getSalary() << "\n";
```

```
    emp.setSalary(-100);  // invalid, will be rejected
    emp.setSalary(5500);  // updates salary
    cout << "Updated Salary: " << emp.getSalary() << "\n";

    // emp.salary = 6000;

    return 0;
}
```