Module 3

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

**1. Basic Concept**

* **Procedural Programming (PP):**  
  Focuses on **functions (procedures)** that operate on data. Programs are structured as a sequence of steps (procedures).
* **Object-Oriented Programming (OOP):**  
  Focuses on **objects** that combine both **data (attributes)** and **behavior (methods)** into a single unit.

**2. Approach**

* **PP:** Top-down approach → problem is broken into smaller functions.
* **OOP:** Bottom-up approach → problem is solved by creating objects and defining their interactions.

**3. Data Handling**

* **PP:** Data is usually **global** or passed between functions. Functions act on this data.
* **OOP:** Data is **encapsulated** inside objects, accessed only through defined methods (abstraction).

**4. Reusability**

* **PP:** Limited reusability. Functions can be reused, but data and functions are separate.
* **OOP:** High reusability through **inheritance** and **polymorphism**.

**5. Key Features**

* **PP:** Uses functions, loops, conditionals. Emphasis is on logic.
* **OOP:** Supports **Encapsulation, Inheritance, Polymorphism, Abstraction** (the four pillars of OOP).

**6. Example (Conceptual)**

**Procedural (C-like):**

int width, height;

int area(int w, int h) {

return w \* h;

}

int main() {

width = 5;

height = 10;

printf("Area: %d", area(width, height));

}

**OOP (Java-like):**

class Rectangle {

int width, height;

Rectangle(int w, int h) {

width = w;

height = h;

}

int area() {

return width \* height;

}

}

public class Main {

public static void main(String[] args) {

Rectangle rect = new Rectangle(5, 10);

System.out.println("Area: " + rect.area());

}

}

**7. Examples of Languages**

* **PP:** C, Pascal, Fortran.
* **OOP:** Java, C++, Python, C#, Ruby.

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

# **Advantages of OOP over POP**

### 1. **Encapsulation (Data Security)**

* **OOP:** Data and methods are bundled into objects, and access is controlled using access modifiers (private, public, protected).
* **POP:** Data is usually global and can be directly accessed/modified by any function.
* **Advantage:** OOP provides **data hiding**, which improves security and prevents accidental data corruption.

### 2. **Reusability through Inheritance**

* **OOP:** Existing classes can be reused and extended using **inheritance**, reducing redundancy.
* **POP:** Code reuse is limited to function calls; no concept of inheritance.
* **Advantage:** Faster development, less code duplication, and easier maintenance.

### 3. **Polymorphism (Flexibility & Extensibility)**

* **OOP:** A single interface can represent multiple implementations (method overloading/overriding).
* **POP:** Functions cannot easily be reused for different data types or purposes without duplication.
* **Advantage:** Improves flexibility and allows for **extensible designs**.

### 4. **Modularity**

* **OOP:** Programs are divided into objects (self-contained modules).
* **POP:** Programs are divided into functions, but data is often shared across them.
* **Advantage:** OOP modules (objects/classes) are **independent and reusable**, making debugging and testing easier.

### 5. **Abstraction (Focus on What, not How)**

* **OOP:** Abstract classes and interfaces let developers hide complex details and expose only essential features.
* **POP:** Developers must work directly with functions and data without a strong abstraction mechanism.
* **Advantage:** Simplifies complex systems and enhances productivity.

### 6. **Better Maintainability**

* **OOP:** Changes in one class usually don’t affect others if encapsulation and modularity are followed.
* **POP:** Any change in global data may affect many functions, increasing the risk of errors.
* **Advantage:** OOP systems are easier to maintain and scale.

### 7. **Real-World Modeling**

* **OOP:** Maps directly to real-world entities (e.g., Student, Employee, Car objects).
* **POP:** Focuses on step-by-step instructions, which may not reflect real-world systems easily.
* **Advantage:** Makes design more **intuitive and closer to real-world problems**.

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

# Steps to Set Up a C++ Development Environment

### **1. Install a C++ Compiler**

A compiler translates your C++ code into machine code. Common ones are:

* **GCC (GNU Compiler Collection)** – popular on Linux/Windows (via MinGW or WSL).
* **Clang** – used on macOS and some Linux systems.
* **MSVC (Microsoft Visual C++)** – comes with Visual Studio on Windows.

👉 On different OS:

* **Windows:** Install MinGW or Microsoft Visual Studio.
* **Linux (Ubuntu/Debian):**
* sudo apt update
* sudo apt install g++
* **macOS:**
* xcode-select --install

### **2. Choose an IDE or Code Editor**

An IDE/editor helps you write code efficiently with features like syntax highlighting, auto-completion, and debugging.

* **Lightweight Editors:** VS Code, Sublime Text, Atom.
* **Full IDEs:** Visual Studio (Windows), CLion, Code::Blocks, Eclipse CDT, Dev C++.

👉 Most students use **VS Code + GCC/MinGW** because it’s simple and cross-platform.

### **3. Configure the Environment Variables (if needed)**

On Windows, after installing **MinGW**:

* Add the bin folder path (e.g., C:\MinGW\bin) to the **PATH** environment variable.
* Verify installation by typing:
* g++ --version

in **Command Prompt/Terminal**.

### **4. Write a Sample C++ Program**

Example: hello.cpp

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl;

return 0;

}

### **5. Compile the Program**

Open terminal/command prompt in the file’s folder and run:

g++ hello.cpp -o hello

* g++ → Compiler command
* hello.cpp → Source file
* -o hello → Output file (executable named hello)

### **6. Run the Program**

* On **Linux/macOS**:
* ./hello
* On **Windows**:
* hello.exe

### **7. Optional – Debugging Tools**

Install debugging tools like **gdb** (GNU Debugger) or use the built-in debugger in IDEs like Visual Studio or CLion.

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

# **Main Input/Output Operations in C++**

### 1. **Output using** cout

* cout (character output) is used to **display data** on the console.
* It uses the **insertion operator (<<)**.

**Example:**

#include <iostream>

using namespace std;

int main() {

int age = 20;

cout << "Hello, World!" << endl;

cout << "I am " << age << " years old." << endl;

return 0;

}

**Output:**

Hello, World!

I am 20 years old.

### 2. **Input using** cin

* cin (character input) is used to **take input from the user**.
* It uses the **extraction operator (>>)**.

**Example:**

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

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

return 0;

}

**Input:** 15  
**Output:** You entered: 15

### 3. **Character Input/Output (**get() **and** put()**)**

* cin.get() → Reads a single character (including whitespace).
* cout.put() → Outputs a single character.

**Example:**

#include <iostream>

using namespace std;

int main() {

char ch;

cout << "Enter a character: ";

ch = cin.get(); // reads one character

cout << "You entered: ";

cout.put(ch); // prints that character

return 0;

}

**Input:** A  
**Output:** You entered: A

### 4. **getline() for Strings**

* getline(cin, variable) → Reads a full line of text (including spaces).

**Example:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name;

cout << "Enter your full name: ";

getline(cin, name);

cout << "Hello, " << name << "!" << endl;

return 0;

}

**Input:** John Smith  
**Output:** Hello, John Smith!

### 5. **File Input/Output (**ifstream**,** ofstream**)**

C++ also supports file operations.

* ofstream → Write to a file
* ifstream → Read from a file

**Example:**

#include <iostream>

#include <fstream>

using namespace std;

int main() {

// Write to file

ofstream outFile("example.txt");

outFile << "This is a test file." << endl;

outFile.close();

// Read from file

ifstream inFile("example.txt");

string line;

getline(inFile, line);

cout << "File content: " << line << endl;

inFile.close();

return 0;

}

**Output:**  
File content: This is a test file.

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

# **1. Basic (Primitive) Data Types**

These are the fundamental types provided by C++.

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| int | Stores integers (whole numbers) | int age = 21; |
| float | Stores single-precision decimal numbers | float pi = 3.14; |
| double | Stores double-precision decimal numbers | double price = 99.99; |
| char | Stores a single character (1 byte) | char grade = 'A'; |
| bool | Stores true or false | bool isPassed = true; |
| void | Represents "no value" (used in functions) | void greet() { cout << "Hi"; } |

# ✅ **2. Derived Data Types**

Built from basic data types.

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| **Array** | Collection of elements of the same type | int marks[5] = {10, 20, 30, 40, 50}; |
| **Pointer** | Stores memory address of another variable | int x = 10; int\* ptr = &x; |
| **Reference** | An alias for another variable | int y = 5; int &ref = y; |
| **Function** | Group of statements performing a task | int add(int a, int b) { return a+b; } |

# ✅ **3. User-Defined Data Types**

Programmer-defined types to model real-world entities.

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| **struct** | Groups different data types together | struct Student { int id; char name[20]; }; |
| **class** | Encapsulation of data & methods (OOP) | class Car { public: string model; void start() { cout<<"Running"; } }; |
| **enum** | User-defined set of named constants | enum Color { Red, Green, Blue }; |
| **typedef/using** | Create an alias for a data type | typedef unsigned int uint; uint age = 30; |

# ✅ **4. Abstract/Advanced Data Types**

These are built using user-defined + standard types.

* **String (std::string):** Stores a sequence of characters.
* string name = "Alice";
* **Vector, List, Map (STL Containers):** Used for dynamic storage.
* #include <vector>
* vector<int> numbers = {1, 2, 3, 4};

# ✅ **Examples in a Single Program**

#include <iostream>

#include <string>

using namespace std;

// User-defined structure

struct Student {

int rollNo;

string name;

};

int main() {

// Basic data types

int age = 20;

float height = 5.9;

char grade = 'A';

bool isPassed = true;

// Derived data types

int marks[3] = {85, 90, 95};

int x = 10;

int\* ptr = &x; // pointer

// User-defined data types

Student s1 = {101, "John"};

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

cout << "Height: " << height << endl;

cout << "Grade: " << grade << endl;

cout << "Passed: " << isPassed << endl;

cout << "Marks[0]: " << marks[0] << endl;

cout << "Pointer value: " << \*ptr << endl;

cout << "Student Name: " << s1.name << endl;

return 0;

}

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

# **Type Conversion in C++**

Type conversion = changing a variable from one data type to another.  
It can be **implicit (type casting done automatically by the compiler)** or **explicit (done manually by the programmer).**

## ****1. Implicit Type Conversion (Type Casting / Type Promotion)****

* Also called **Type Promotion** or **Type Casting by Compiler**.
* Happens **automatically** when a smaller data type is converted into a larger data type to prevent data loss.
* No loss of data **if the target type can hold the value**.

**Rules:**

* char → int → float → double (promotion chain)
* Lower type automatically promoted to higher type in expressions.

**Example:**

#include <iostream>

using namespace std;

int main() {

int num = 10;

double result = num + 2.5; // int (10) is converted to double (10.0)

cout << "Result = " << result << endl; // Output: 12.5

return 0;

}

Here, int num was **implicitly converted** to double.

## ****2. Explicit Type Conversion (Type Casting by Programmer)****

* Also called **Type Casting**.
* Programmer **forces conversion** from one type to another using **casting operators**.
* Syntax:
  + **C-style cast:** (type)expression
  + **C++ style cast:** static\_cast<type>(expression)

**Example 1 (C-style cast):**

#include <iostream>

using namespace std;

int main() {

double pi = 3.14159;

int x = (int) pi; // Explicit conversion (fractional part lost)

cout << "pi = " << pi << endl;

cout << "After casting, x = " << x << endl;

return 0;

}

**Output:**

pi = 3.14159

After casting, x = 3

**Example 2 (C++ style cast):**

#include <iostream>

using namespace std;

int main() {

double pi = 3.14159;

int x = static\_cast<int>(pi); // Explicit type casting

cout << "x = " << x << endl; // Output: 3

return 0;

}

Q7. **What are the different types of operators in C++? Provide examples of each**

# **Types of Operators in C++**

## ****1. Arithmetic Operators****

Used for mathematical calculations.

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

**Example:**

int a = 10, b = 3;

cout << a + b; // 13

cout << a / b; // 3 (integer division)

cout << a % b; // 1

## ****2. Relational (Comparison) Operators****

Used to compare values. Result is true (1) or false (0).

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

**Example:**

int x = 5, y = 10;

cout << (x < y); // 1 (true)

cout << (x == y); // 0 (false)

## ****3. Logical Operators****

Used for combining conditions.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| && | Logical AND | (x > 0 && y > 0) |
| ` |  | ` |
| ! | Logical NOT | !(x > 0) |

**Example:**

int x = 5, y = -3;

cout << (x > 0 && y > 0); // 0 (false)

cout << (x > 0 || y > 0); // 1 (true)

cout << !(x > 0); // 0 (false)

## ****4. Assignment Operators****

Used to assign values.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| = | Assign | x = 10; |
| += | Add and assign | x += 5; // x = x + 5 |
| -= | Subtract and assign | x -= 5; |
| \*= | Multiply and assign | x \*= 2; |
| /= | Divide and assign | x /= 2; |
| %= | Modulus and assign | x %= 3; |

## ****5. Increment and Decrement Operators****

Used to increase or decrease value by 1.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| ++x | Pre-increment | x=5; ++x; // x=6 |
| x++ | Post-increment | x=5; x++; // x=6 (but returns old value first) |
| --x | Pre-decrement | x=5; --x; // x=4 |
| x-- | Post-decrement | x=5; x--; // x=4 |

## ****6. Bitwise Operators****

Operate on binary (bit-level) data.

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

**Example:**

int a = 5, b = 3; // (5=0101, 3=0011)

cout << (a & b); // 1 (0001)

cout << (a | b); // 7 (0111)

cout << (a ^ b); // 6 (0110)

cout << (a << 1); // 10 (1010)

## ****7. Conditional (Ternary) Operator****

Shortcut for if-else.

int a = 10, b = 20;

int max = (a > b) ? a : b;

cout << max; // 20

## ****8. Miscellaneous Operators****

* **sizeof** → Returns size of a variable/type
* **&** → Address-of operator
* **\*** → Pointer dereference
* **::** → Scope resolution operator

**Example:**

int x = 5;

cout << sizeof(x); // 4 (on most systems)

cout << &x; // prints memory address

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

# **1. Constants in C++**

A **constant** is a variable whose value **cannot be changed** during program execution.  
They are used when you want fixed values that should not be modified by mistake.

### **Ways to define constants**

1. **Using const keyword**

#include <iostream>

using namespace std;

int main() {

const double PI = 3.14159; // constant declaration

// PI = 3.2; ❌ Error: cannot modify a constant

cout << "Value of PI = " << PI << endl;

return 0;

}

1. **Using #define preprocessor**

#include <iostream>

#define PI 3.14159

using namespace std;

int main() {

cout << "Value of PI = " << PI << endl;

return 0;

}

🔹 **Best Practice:** Use const instead of #define because it provides **type safety**.

# ✅ **2. Literals in C++**

A **literal** is a fixed value directly written in the program.  
They represent constant values of different data types.

### **Types of Literals**

1. **Integer Literals**  
   Whole numbers (can be decimal, octal, hexadecimal).
2. int a = 100; // decimal
3. int b = 014; // octal (starts with 0)
4. int c = 0x1A; // hexadecimal (starts with 0x)
5. **Floating-point Literals**  
   Decimal or scientific notation.
6. float x = 3.14f; // float literal
7. double y = 2.71828; // double literal
8. double z = 1.2e3; // scientific notation = 1.2 × 10³ = 1200
9. **Character Literals**  
   A single character inside single quotes.
10. char ch = 'A';
11. char newline = '\n'; // escape sequence
12. **String Literals**  
    Sequence of characters inside double quotes.
13. string name = "Alice";
14. **Boolean Literals**  
    Only two values: true or false.
15. bool isOn = true;
16. bool isOff = false;

# ✅ **Example: Constants & Literals Together**

#include <iostream>

using namespace std;

int main() {

const double PI = 3.14159; // constant

int radius = 5; // integer literal

double area = PI \* radius \* radius; // using constant + literals

cout << "Radius = " << radius << endl;

cout << "Area of circle = " << area << endl;

return 0;

}

**Output:**

Radius = 5

Area of circle = 78.5397

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

# **Conditional Statements in C++**

A conditional statement allows the program to execute **different blocks of code** depending on whether a condition is true or false.

The main conditional statements in C++ are:

1. if
2. if-else
3. if-else if-else
4. switch

## ****1. if-else Statement****

* The **if** block executes if the condition is **true**.
* The **else** block executes if the condition is **false**.

### **Syntax:**

if (condition) {

// code if condition is true

} else {

// code if condition is false

}

### **Example:**

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

if (age >= 18) {

cout << "You are eligible to vote." << endl;

} else {

cout << "You are not eligible to vote." << endl;

}

return 0;

}

**Input:** 20  
**Output:** You are eligible to vote.

## ****2. if-else if-else Ladder****

* Used when there are **multiple conditions**.

### **Example:**

int marks = 75;

if (marks >= 90) {

cout << "Grade: A+" << endl;

} else if (marks >= 75) {

cout << "Grade: A" << endl;

} else if (marks >= 50) {

cout << "Grade: B" << endl;

} else {

cout << "Fail" << endl;

}

## ****3. switch Statement****

* Used when we need to **choose between multiple options** based on a single variable’s value.
* More efficient and readable than multiple if-else statements for equality checks.

### **Syntax:**

switch (expression) {

case constant1:

// code block

break;

case constant2:

// code block

break;

...

default:

// code block if no case matches

}

🔹 **break** is used to exit the switch after a case executes.  
🔹 If break is missing, execution will **fall through** to the next case.

### **Example:**

#include <iostream>

using namespace std;

int main() {

int day;

cout << "Enter day number (1-7): ";

cin >> day;

switch(day) {

case 1: cout << "Monday"; break;

case 2: cout << "Tuesday"; break;

case 3: cout << "Wednesday"; break;

case 4: cout << "Thursday"; break;

case 5: cout << "Friday"; break;

case 6: cout << "Saturday"; break;

case 7: cout << "Sunday"; break;

default: cout << "Invalid day number";

}

return 0;

}

**Input:** 3  
**Output:** Wednesday

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

# **1. for loop**

* Best when the **number of iterations is known** in advance.
* Combines **initialization, condition, and update** in one line.

### **Syntax:**

for(initialization; condition; update) {

// code to execute

}

### **Example:**

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 5; i++) {

cout << "Iteration: " << i << endl;

}

return 0;

}

**Output:**

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

Iteration: 5

# **2. while loop**

* Best when the **number of iterations is not known** in advance.
* The condition is checked **before** the loop body runs.
* If the condition is **false at the start**, the loop body **may not execute at all**.

### **Syntax:**

while(condition) {

// code to execute

}

### **Example:**

#include <iostream>

using namespace std;

int main() {

int i = 1;

while (i <= 5) {

cout << "Iteration: " << i << endl;

i++;

}

return 0;

}

# **3. do-while loop**

* Similar to while, but the loop body is executed **at least once**, even if the condition is false.
* Condition is checked **after** executing the loop body.

### **Syntax:**

do {

// code to execute

} while(condition);

### **Example:**

#include <iostream>

using namespace std;

int main() {

int i = 1;

do {

cout << "Iteration: " << i << endl;

i++;

} while (i <= 5);

return 0;

}

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

# **1. break Statement**

* Immediately **terminates the loop** (for, while, or do-while).
* Control moves to the **first statement after the loop**.

### **Example (break in a for loop):**

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 10; i++) {

if (i == 5) {

break; // exit the loop when i = 5

}

cout << "i = " << i << endl;

}

return 0;

}

**Output:**

1

2

3

4

👉 The loop stopped when i == 5.

# 🔹 **2. continue Statement**

* Skips the **current iteration** of the loop.
* Control moves to the **next iteration** of the loop.
* The loop itself does **not terminate**.

### **Example (continue in a for loop):**

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 10; i++) {

if (i == 5) {

continue; // skip when i = 5

}

cout << "i = " << i << endl;

}

return 0;

}

**Output:**

1

2

3

4

6

7

8

9

10

👉 The loop skipped i = 5, but continued running.

# **Example in a while loop**

#include <iostream>

using namespace std;

int main() {

int i = 0;

while (i < 10) {

i++;

if (i == 3) continue; // skip printing 3

if (i == 7) break; // stop when i = 7

cout << "i = " << i << endl;

}

return 0;

}

**Output:**

1

2

4

5

6

=> 3 was skipped using continue, and the loop terminated at 7 using break.

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

# **Nested Control Structures in C++**

A **nested control structure** means **one control structure (like if, loop, or switch) inside another**.

* This allows us to build more **complex logic**.
* The most common nested structures are:
  + if inside if (nested if-else)
  + Loop inside loop (nested loops)
  + switch inside switch

## ****1. Nested if-else Example****

#include <iostream>

using namespace std;

int main() {

int age, marks;

cout << "Enter age: ";

cin >> age;

cout << "Enter marks: ";

cin >> marks;

if (age >= 18) {

if (marks >= 50) {

cout << "Eligible for admission." << endl;

} else {

cout << "Not eligible (marks too low)." << endl;

}

} else {

cout << "Not eligible (age below 18)." << endl;

}

return 0;

}

👉 Here, the **outer if** checks age, and the **inner if** checks marks.

## ****2. Nested Loops Example****

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 3; i++) { // outer loop

for (int j = 1; j <= 3; j++) { // inner loop

cout << "(" << i << ", " << j << ") ";

}

cout << endl;

}

return 0;

}

**Output:**

(1, 1) (1, 2) (1, 3)

(2, 1) (2, 2) (2, 3)

(3, 1) (3, 2) (3, 3)

👉 Useful in **tables, grids, and matrix operations**.

## ****3. Nested switch Example****

#include <iostream>

using namespace std;

int main() {

int category = 1, option = 2;

switch (category) {

case 1:

cout << "Category: Electronics" << endl;

switch (option) {

case 1: cout << "Laptop" << endl; break;

case 2: cout << "Smartphone" << endl; break;

default: cout << "Other electronics" << endl;

}

break;

case 2:

cout << "Category: Clothing" << endl;

break;

default:

cout << "Invalid category" << endl;

}

return 0;

}.

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

A **function** in C++ is a **block of code** that performs a specific task.

* It helps in **code reusability** (write once, use many times).
* It makes programs more **organized, readable, and maintainable**.

👉 Example: Instead of writing the same code to add two numbers in 5 places, we can **define a function** add() and call it wherever needed.

# 🔹 **Parts of a Function**

## 1. ****Function Declaration (Prototype)****

* Tells the compiler the **name**, **return type**, and **parameters** of the function.
* Written **before main()** (or in a header file).

**Syntax:**

return\_type function\_name(parameter\_list);

**Example:**

int add(int a, int b); // function declaration

## 2. ****Function Definition****

* Contains the **actual code** (body) of the function.
* Written either **before or after main()**.

**Syntax:**

return\_type function\_name(parameter\_list) {

// function body

return value; // if return\_type is not void

}

**Example:**

int add(int a, int b) {

return a + b;

}

## 3. ****Function Call****

* Executes the function.
* Written inside **main()** or another function.

**Syntax:**

function\_name(arguments);

**Example:**

int result = add(5, 10); // function call

# 🔹 **Complete Example**

#include <iostream>

using namespace std;

// Function Declaration

int add(int a, int b);

int main() {

int num1 = 10, num2 = 20;

// Function Call

int sum = add(num1, num2);

cout << "Sum = " << sum << endl;

return 0;

}

// Function Definition

int add(int a, int b) {

return a + b;

}

**Output:**

Sum = 30

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

# **Scope of Variables in C++**

* **Scope** defines the **lifetime** and **visibility** of a variable.
* Determines **where** in the program the variable can be used.

There are mainly **two types of scope** in C++:

1. **Local Scope**
2. **Global Scope**

## ****1. Local Scope****

* A variable declared **inside a function, block ({ }), or loop**.
* Exists **only while the function/block executes**.
* **Not accessible outside** that function/block.

### **Example:**

#include <iostream>

using namespace std;

int main() {

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

if (x > 5) {

int y = 20; // local variable to if-block

cout << "Inside block, y = " << y << endl;

}

// cout << y; ❌ Error: y is not accessible here

cout << "x = " << x << endl;

return 0;

}

**Output:**

Inside block, y = 20

x = 10

👉 y is destroyed when the block ends.

## ****2. Global Scope****

* A variable declared **outside all functions**.
* Accessible **throughout the program** (in all functions).
* Exists for the **entire lifetime of the program**.

### **Example:**

#include <iostream>

using namespace std;

int g = 100; // global variable

void display() {

cout << "Global variable g = " << g << endl;

}

int main() {

cout << "Accessing global variable in main: " << g << endl;

display(); // also accesses g

return 0;

}

**Output:**

Accessing global variable in main: 100

Global variable g = 100

👉 g is accessible both in main() and display().

**Output:**

Local x = 10

Global x = 50

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

1. **Base Case** → Condition where the recursion stops.
2. **Recursive Case** → Function calls itself with smaller/simpler input.

# **General Syntax**

return\_type function\_name(parameters) {

if (base\_condition) {

// stop recursion

return value;

} else {

// recursive call

return function\_name(smaller\_problem);

}

}

# 🔹 **Example 1: Factorial using Recursion**

Factorial of n → n! = n × (n-1) × (n-2) … × 1

### **Code:**

#include <iostream>

using namespace std;

// Recursive function for factorial

int factorial(int n) {

if (n == 0 || n == 1) { // Base case

return 1;

}

return n \* factorial(n - 1); // Recursive call

}

int main() {

int num = 5;

cout << "Factorial of " << num << " = " << factorial(num) << endl;

return 0;

}

**Output:**

Factorial of 5 = 120

# 🔹 **Example 2: Fibonacci Series using Recursion**

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, …

* F(0) = 0, F(1) = 1
* F(n) = F(n-1) + F(n-2)

### **Code:**

#include <iostream>

using namespace std;

int fibonacci(int n) {

if (n == 0) return 0; // Base case

if (n == 1) return 1; // Base case

return fibonacci(n - 1) + fibonacci(n - 2); // Recursive call

}

int main() {

int n = 6;

cout << "Fibonacci sequence up to " << n << " terms: ";

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

cout << fibonacci(i) << " ";

}

return 0;

}

**Output:**

Fibonacci sequence up to 6 terms: 0 1 1 2 3 5

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

A **function prototype** is a **declaration** of a function that tells the compiler:

* **Return type** of the function
* **Function name**
* **Number and type of parameters**

👉 It **does not contain the body (actual code)**.  
👉 Ends with a **semicolon (;)**.

## ****Syntax****

return\_type function\_name(parameter\_list);

### Example:

int add(int a, int b); // function prototype

# 🔹 **Why are Function Prototypes Used?**

1. **Compiler knows function details before its use**
   * In C++, functions must be declared **before they are called**.
   * If the function definition is written **after main()**, we need a prototype.
2. **Type Checking**
   * Ensures correct number and type of arguments are passed during function calls.
   * Prevents errors like passing float when an int is expected.
3. **Improves Code Organization**
   * Allows us to place function definitions anywhere (before or after main).
   * Useful in **header files** for larger projects.

# 🔹 **Example Without Prototype (Error)**

#include <iostream>

using namespace std;

int main() {

int result = add(5, 10); // ❌ Error: compiler doesn't know add()

cout << "Sum = " << result;

return 0;

}

int add(int a, int b) {

return a + b;

}

# 🔹 **Example With Prototype (Correct)**

#include <iostream>

using namespace std;

// Function Prototype

int add(int, int);

int main() {

int result = add(5, 10); // compiler knows add() exists

cout << "Sum = " << result;

return 0;

}

// Function Definition

int add(int a, int b) {

return a + b;

}

**Output:**

Sum = 15

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

An **array** is a collection of **elements of the same data type**, stored in **contiguous memory locations**, and accessed using an **index**.

* Indexing starts from **0**.
* Useful when we want to store **multiple values of the same type** (like marks of 50 students).

## ****1. Single-Dimensional Array****

* Also called a **1D array** or **linear array**.
* Stores data in a **single row**.
* Accessed with **one index**.

### **Syntax**

data\_type array\_name[size];

### **Example**

#include <iostream>

using namespace std;

int main() {

int marks[5] = {90, 85, 88, 76, 95}; // single-dimensional array

cout << "Marks of students: " << endl;

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

cout << "Student " << i+1 << ": " << marks[i] << endl;

}

return 0;

}

**Output:**

Student 1: 90

Student 2: 85

Student 3: 88

Student 4: 76

Student 5: 95

👉 Here, marks[i] is accessed using a **single index**.

## ****2. Multi-Dimensional Array****

* Stores data in **rows and columns** (like a table or matrix).
* Accessed using **two or more indices**.
* Most common is the **2D array** (matrix).

### **Syntax (2D array)**

data\_type array\_name[rows][columns];

### **Example (2D array)**

#include <iostream>

using namespace std;

int main() {

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

cout << "Matrix elements: " << endl;

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

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

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

}

cout << endl;

}

return 0;

}

**Output:**

1 2 3

4 5 6

👉 Here, matrix[i][j] is accessed using **two indices**.

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

# **1. C-Style Strings (Character Arrays)**

* Represented as an **array of characters**.
* Always terminated by a **null character ('\0')**.
* Supported by functions from <cstring> (like strlen, strcpy, strcat, etc.).

### **Example:**

#include <iostream>

#include <cstring> // for string functions

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

cout << "String 1: " << str1 << endl;

cout << "String 2: " << str2 << endl;

// Concatenate

strcat(str1, str2);

cout << "Concatenated: " << str1 << endl;

// Length

cout << "Length of str1: " << strlen(str1) << endl;

return 0;

}

**Output:**

String 1: Hello

String 2: World

Concatenated: HelloWorld

Length of str1: 10

👉 Drawback: You must manage **array size** and **null terminator (\0)** manually.

# 🔹 **2. C++ Strings (**std::string**)**

* Provided by <string> header.
* Easier and safer than C-style strings.
* Supports many **built-in functions** for string manipulation.

### **Example:**

#include <iostream>

#include <string> // for C++ string class

using namespace std;

int main() {

string str1 = "Hello";

string str2 = "World";

cout << "String 1: " << str1 << endl;

cout << "String 2: " << str2 << endl;

// Concatenate

string str3 = str1 + " " + str2;

cout << "Concatenated: " << str3 << endl;

// Length

cout << "Length of str3: " << str3.length() << endl;

// Access characters

cout << "First character: " << str3[0] << endl;

// Substring

cout << "Substring (0-4): " << str3.substr(0, 5) << endl;

// Compare

if (str1 == "Hello")

cout << "Strings are equal" << endl;

return 0;

}

**Output:**

String 1: Hello

String 2: World

Concatenated: Hello World

Length of str3: 11

First character: H

Substring (0-4): Hello

Strings are equal

Q19. **How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.**

# **1. Initializing a 1D Array**

A **1D array** is like a list of elements stored in contiguous memory.

### ✅ Different ways to initialize a 1D array:

### **(a) Full Initialization**

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

### **(b) Partial Initialization (remaining values set to 0)**

int arr[5] = {10, 20}; // becomes {10, 20, 0, 0, 0}

### **(c) Size Inference (compiler calculates size automatically)**

int arr[] = {1, 2, 3, 4, 5}; // size = 5

### **(d) Default Initialization (all elements = 0)**

int arr[5] = {0}; // becomes {0, 0, 0, 0, 0}

# **2. Initializing a 2D Array**

A **2D array** is like a table (rows and columns).

### ✅ Different ways to initialize a 2D array:

### **(a) Row-wise Initialization**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

### **(b) Inline Initialization**

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

👉 Stored row-wise in memory.

### **(c) Partial Initialization**

int matrix[2][3] = {

{1}, // becomes {1, 0, 0}

{4, 5} // becomes {4, 5, 0}

};

### **(d) Default Initialization**

int matrix[2][3] = {0}; // all elements become 0

# **Example Program (1D & 2D Initialization)**

#include <iostream>

using namespace std;

int main() {

// 1D Array

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

cout << "1D Array Elements: ";

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

cout << arr[i] << " ";

}

cout << endl;

// 2D Array

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

cout << "2D Array Elements: " << endl;

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

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

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

}

cout << endl;

}

return 0;

}

**Output:**

1D Array Elements: 10 20 30 40 50

2D Array Elements:

1 2 3

4 5 6

Q20. **Explain string operations and functions in C++.**

# **1. C-Style String Operations (using** <cstring>**)**

A **C-string** is an array of characters ending with a **null character ('\0')**.

### Common Functions:

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| strlen(str) | Returns length of string | strlen("Hello") → 5 |
| strcpy(dest, src) | Copies src into dest | "World" → "World" |
| strcat(dest, src) | Concatenates src to dest | "Hello" + "World" |
| strcmp(str1, str2) | Compares strings (0 if equal) | strcmp("a","a") → 0 |
| strchr(str, ch) | Finds first occurrence of char | strchr("Hello",'e') → "ello" |
| strstr(str, sub) | Finds substring inside string | strstr("HelloWorld","World") → "World" |

### Example:

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

cout << "Length of str1: " << strlen(str1) << endl;

strcpy(str2, str1);

cout << "After strcpy: " << str2 << endl;

strcat(str1, " C++");

cout << "After strcat: " << str1 << endl;

cout << "Comparison: " << strcmp("abc", "abc") << endl; // 0 = equal

return 0;

}

# 🔹 **2. C++ String Operations (**std::string**)**

Modern strings are objects of the **string class** from <string>.  
They are much easier and safer than C-Style strings.

### Common Operations and Functions:

| **Operation / Function** | **Description** | **Example** |
| --- | --- | --- |
| + | Concatenation | "Hello" + " World" |
| += | Append | s += " C++" |
| .length() / .size() | Length of string | "Hello".length() → 5 |
| .at(i) / [i] | Access character at index | s[0] → 'H' |
| .substr(pos, len) | Extract substring | "Hello".substr(1,3) → "ell" |
| .find(str) | Find substring | "Hello World".find("World") → 6 |
| .compare(str) | Compare strings | s1.compare(s2) (0 = equal) |
| .empty() | Checks if string is empty | s.empty() |
| .insert(pos, str) | Insert string at position | s.insert(5,"World") |
| .erase(pos, len) | Erase part of string | s.erase(0,5) |
| .replace(pos, len, str) | Replace part of string | s.replace(0,5,"Hi") |

### Example:

#include <iostream>

#include <string>

using namespace std;

int main() {

string str1 = "Hello";

string str2 = "World";

// Concatenation

string str3 = str1 + " " + str2;

cout << "Concatenated: " << str3 << endl;

// Length

cout << "Length: " << str3.length() << endl;

// Substring

cout << "Substring: " << str3.substr(0, 5) << endl;

// Find

cout << "Find 'World': " << str3.find("World") << endl;

// Replace

str3.replace(0, 5, "Hi");

cout << "After replace: " << str3 << endl;

return 0;

}

**Output:**

Concatenated: Hello World

Length: 11

Substring: Hello

Find 'World': 6

After replace: Hi World

Q21. Explain the key concepts of Object-Oriented Programming (OOP).

# **Key Concepts of OOP**

### 1. **Class**

* A **blueprint** or **template** for creating objects.
* Defines **data members (attributes/variables)** and **member functions (methods)**.

Example:

class Car {

public:

string brand;

int speed;

void display() {

cout << "Brand: " << brand << ", Speed: " << speed << endl;

}

};

### 2. **Object**

* An **instance of a class**.
* Each object has its **own copy of data members**, but can use the class functions.

✅ Example:

Car c1;

c1.brand = "Toyota";

c1.speed = 120;

c1.display();

### 3. **Encapsulation**

* Wrapping of **data** and **functions** together inside a class.
* Protects data using **access specifiers** (public, private, protected).

✅ Example:

class Account {

private:

int balance; // hidden data

public:

void setBalance(int b) { balance = b; }

int getBalance() { return balance; }

};

👉 balance is protected from direct access.

### 4. **Abstraction**

* **Hiding implementation details** and showing only the **essential features**.
* Achieved using **classes, abstract classes, and interfaces**.

✅ Example:

class Shape {

public:

virtual void draw() = 0; // pure virtual function (abstraction)

};

### 5. **Inheritance**

* **Reusing code** by creating new classes (child/derived) from existing ones (parent/base).
* Types: **Single, Multiple, Multilevel, Hierarchical, Hybrid**.

✅ Example:

class Vehicle {

public:

void start() { cout << "Vehicle started" << endl; }

};

class Car : public Vehicle {

public:

void drive() { cout << "Car is driving" << endl; }

};

👉 Car inherits from Vehicle.

### 6. **Polymorphism**

* **One name, many forms** → same function behaves differently in different contexts.
* Types:
  + **Compile-time (Static)** → Function Overloading, Operator Overloading
  + **Runtime (Dynamic)** → Function Overriding (with virtual functions).

✅ Example (Function Overloading):

class Print {

public:

void show(int x) { cout << "Integer: " << x << endl; }

void show(string s) { cout << "String: " << s << endl; }

};

✅ Example (Function Overriding):

class Animal {

public:

virtual void sound() { cout << "Some sound" << endl; }

};

class Dog : public Animal {

public:

void sound() override { cout << "Bark" << endl; }

};

Q22. What are classes and objects in C++? Provide an example

# **1. Class in C++**

* A **class** is a **blueprint or template** for creating objects.
* It defines:
  1. **Data members (variables)** → attributes of the object
  2. **Member functions (methods)** → actions the object can perform
* Declared using the class keyword.

### **Syntax**

class ClassName {

public:

// data members

int data;

// member functions

void display() {

cout << "Data = " << data << endl;

}

};

# 🔹 **2. Object in C++**

* An **object** is an **instance of a class**.
* Each object has its **own copy of data members**, but can use the same member functions.

### **Syntax**

ClassName objectName;

# 🔹 **Example: Class and Object**

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

string brand;

int speed;

void display() {

cout << "Brand: " << brand << ", Speed: " << speed << " km/h" << endl;

}

};

int main() {

// Creating objects

Car car1;

Car car2;

// Assigning values to car1

car1.brand = "Toyota";

car1.speed = 120;

// Assigning values to car2

car2.brand = "Honda";

car2.speed = 150;

// Accessing member function

car1.display();

car2.display();

return 0;

}

**Output:**

Brand: Toyota, Speed: 120 km/h

Brand: Honda, Speed: 150 km/h

Q23. What is inheritance in C++? Explain with an example

* **Inheritance** is an **OOP concept** that allows a class (child/derived class) to **reuse properties and behaviors** of another class (parent/base class).
* Promotes **code reusability** and **modularity**.

**Key Points:**

1. **Base Class (Parent Class)** → provides attributes and methods.
2. **Derived Class (Child Class)** → inherits from base class and can add its own members.
3. **Access Specifiers** in inheritance: public, protected, private

# 🔹 **Types of Inheritance (Basic)**

1. **Single Inheritance** → one base, one derived
2. **Multiple Inheritance** → one derived class inherits from multiple base classes
3. **Multilevel Inheritance** → chain of inheritance (A → B → C)
4. **Hierarchical Inheritance** → multiple derived classes from one base
5. **Hybrid Inheritance** → combination of above

# 🔹 **Syntax**

class DerivedClass : access\_specifier BaseClass {

// additional members

};

# 🔹 **Example: Single Inheritance**

#include <iostream>

using namespace std;

// Base class

class Vehicle {

public:

string brand;

void start() {

cout << brand << " is starting." << endl;

}

};

// Derived class

class Car : public Vehicle { // Car inherits from Vehicle

public:

int speed;

void drive() {

cout << brand << " is driving at " << speed << " km/h." << endl;

}

};

int main() {

Car myCar;

// Accessing base class members

myCar.brand = "Toyota";

myCar.start();

// Accessing derived class members

myCar.speed = 120;

myCar.drive();

return 0;

}

**Output:**

Toyota is starting.

Toyota is driving at 120 km/h.

Q24. What is encapsulation in C++? How is it achieved in classes?

* **Encapsulation** is an **OOP concept** that **bundles data (variables) and methods (functions) together inside a class**.
* It **hides the internal details** of the class from the outside world.
* Helps in **data protection** and **modular code design**.

**Key idea:** Users of the class can interact with it **only through public functions**, not by directly accessing private data.

# 🔹 **How Encapsulation is Achieved in C++**

1. **Access Specifiers** are used inside classes:
   * private → data/methods **cannot be accessed** outside the class.
   * public → data/methods **can be accessed** anywhere.
   * protected → accessible in **derived classes**, but not outside the class.
2. **Getter and Setter functions** are used to **access or modify private data safely**.

# 🔹 **Example: Encapsulation**

#include <iostream>

using namespace std;

class Account {

private:

int balance; // private data, hidden from outside

public:

// Setter function to set balance

void setBalance(int b) {

if (b >= 0) // validation

balance = b;

else

cout << "Invalid balance!" << endl;

}

// Getter function to get balance

int getBalance() {

return balance;

}

};

int main() {

Account myAccount;

// Direct access not allowed → myAccount.balance = 100; ❌

// Access via setter

myAccount.setBalance(500);

// Access via getter

cout << "Account Balance: " << myAccount.getBalance() << endl;

// Trying to set invalid balance

myAccount.setBalance(-200);

return 0;

}

**Output:**

Account Balance: 500

Invalid balance!