***Introduction to OOPS Programming***

* ***Introduction to C++***

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

* The key differences between Procedural Programming and Object-Oriented Programming (OOP) are based on their design philosophy, structure, and data handling. Here's a clear comparison:

| Feature | Procedural Programming | Object-Oriented Programming (OOP) |
| --- | --- | --- |
| Basic Concept | Based on procedures or routines (functions). | Based on real-world entities called objects. |
| Program Structure | Organized as a set of procedures and functions. | Organized as a collection of objects and classes. |
| Data Handling | Data is usually global or passed between functions. | Data is encapsulated within objects. |
| Focus | Focuses on functions and logic. | Focuses on data (objects) and behavior. |
| Encapsulation | Not a primary feature. | Core principle — data and methods are bundled together. |
| Inheritance | Not supported. | Supported — enables code reusability. |
| Polymorphism | Not supported. | Supported — allows for method overloading/overriding. |
| Reusability | Limited code reuse. | High reusability through inheritance and classes. |
| Security (Data Hiding) | Difficult, as data is usually accessible globally. | Easier, with private/protected access modifiers. |
| Examples of Languages | C, Pascal, FORTRAN | Java, C++, Python, C# |

**Summary:**

* Procedural Programming is linear and function-based, best for small tasks or when performance is critical.
* OOP is modular, scalable, and ideal for large, complex software systems that require maintenance and flexibility.

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

* Here are the main advantages of Object-Oriented Programming (OOP) over Procedural-Oriented Programming (POP), along with brief explanations:

**🔑 1. Modularity**

* **OOP**: Programs are divided into classes and objects, making them modular and easier to manage.
* **POP**: Code is written in a sequential manner, which can lead to tightly coupled code that's harder to modify.
* **✅ Advantage**: Easier debugging and code maintenance.

**🔐 2. Data Encapsulation and Hiding**

* **OOP**: Data and functions are encapsulated within objects. Access is controlled using access specifiers (private, public, protected).
* **POP**: Data is typically global and accessible by all functions, increasing the risk of unintended modifications.
* **✅ Advantage**: Improved security and control over data.

**🔁 3. Reusability through Inheritance**

* **OOP**: Allows classes to inherit properties and behaviors from other classes.
* **POP**: No built-in inheritance; code must be rewritten or copied.
* **✅ Advantage**: Reduces redundancy and speeds up development.

**🔄 4. Polymorphism**

* **OOP**: Supports polymorphism (e.g., method overloading and overriding), enabling different implementations using the same interface.
* **POP**: No direct support for polymorphism.
* **✅ Advantage**: Flexibility and ability to use the same function name for different types of behavior.

**🧩 5. Better Problem Modeling**

* **OOP**: Mimics real-world entities through objects, making it intuitive to design and understand complex systems.
* **POP**: Focuses on procedure and logic, which can be harder to relate to real-world concepts.
* **✅ Advantage**: More natural and realistic approach to problem-solving.

**🔧 6. Easier Maintenance and Scalability**

* **OOP**: Modular code and separation of concerns make it easier to update, maintain, and scale.
* **POP**: Changes in code can affect multiple areas due to lack of encapsulation.
* **✅ Advantage**: Ideal for large, long-term, or collaborative software projects.

**🔍 7. Code Readability and Manageability**

* **OOP**: Promotes cleaner, well-organized code through classes and object structures.
* **POP**: May result in long, cluttered functions with repeated logic.
* **✅ Advantage**: Easier to read, understand, and modify code.
* **Summary Table:**

| **Advantage** | **OOP Benefit** |
| --- | --- |
| Modularity | Code is organized into independent classes/objects |
| Data Encapsulation | Hides internal object details |
| Reusability | Inheritance promotes code reuse |
| Polymorphism | Enables flexible code with method overloading/overriding |
| Real-world Modeling | Objects represent real-world entities |
| Maintenance | Easier to update and scale applications |
| Readability | Code is cleaner and easier to manage |

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

* Here’s a clear and complete explanation of the **steps involved in setting up a C++ development environment** on your system:

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

A compiler is needed to convert your C++ code into an executable program.

**🔹 Popular C++ Compilers:**

* **GCC (GNU Compiler Collection)** – Common on Linux and Windows (via MinGW)
* **Clang** – Default on macOS
* **Microsoft Visual C++** – Part of Visual Studio on Windows

**🔸 How to Install:**

* **Windows**: Install **MinGW** or **Visual Studio**
* **Linux (Ubuntu/Debian)**:
* **macOS**:

xcode-select –install

**2. Choose and Install an IDE or Code Editor**

An Integrated Development Environment (IDE) makes coding easier with features like syntax highlighting, debugging, and code suggestions.

* Popular IDEs:
  + **Visual Studio** (Windows only)
  + **Code::Blocks**
  + **Eclipse CDT**
  + **CLion** (paid, with student free tier)
* Lightweight editors:
  + **Visual Studio Code** (VS Code) – needs C++ extension (C/C++ by Microsoft)
  + **Sublime Text** or **Atom**

**3. Configure the IDE/Editor**

* Link the IDE/editor to your installed C++ compiler.
* In **VS Code**:
  + Install the **C++ extension**.
  + Create tasks.json for building and launch.json for debugging (optional but helpful).
  + Ensure the path to the compiler (g++ or clang++) is set correctly.

**4. Write Your First C++ Program**

Create a file hello.cpp:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

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

return 0;

}

**5. Compile and Run the Program**

Command Line (Terminal/Command Prompt):

bash

g++ hello.cpp -o hello

./hello # or hello.exe on Windows

* **From IDE**:  
  Use the build and run buttons (usually F5 or Ctrl+F5 depending on the IDE).

**✅ Summary**

| **Step** | **Task** |
| --- | --- |
| 1 | Install a C++ compiler (e.g., GCC, Clang) |
| 2 | Choose and install an IDE/editor |
| 3 | Configure the compiler in the IDE |
| 4 | Write C++ code |
| 5 | Compile and run the program |
| 6 | (Optional) Configure debugging tools |

1. What are the main in put/outpu to pation sin C++ ? Provide examples.

**🔹 Main Input/Output (I/O) Operations in C++**

In C++, the primary way to perform input and output operations is using streams, provided by the <iostream> header. The two most commonly used streams are:

**🔸 1. Output: cout**

* Used to **display output** on the screen.
* Belongs to the **standard output stream** (std::cout).
* << is the **insertion operator**.

✅ **Example:**

#include <iostream>

using namespace std;

int main() {

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

cout << "The value is: " << 42 << endl;

return 0;

}

**🔸 2. Input: cin**

* Used to **take input** from the user.
* Belongs to the **standard input stream** (std::cin).
* >> is the **extraction operator**.

**✅ Example:**

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

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

return 0;

}

**🔸 3. Other Useful Output Functions**

* endl: Ends the line and flushes the output buffer.
* \n: Also ends the line, but does not flush the buffer.

**🔸 4. Input of Multiple Values**

You can read multiple inputs in a single line:

int a, b;

cin >> a >> b;

🔸 **5. Character and String Input**

char ch;

cin >> ch; // reads one character

string name;

cin >> name; // reads one word (until space)

getline(cin, name); // reads entire line including spaces

**✅ Example:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string fullName;

cout << "Enter your full name: ";

getline(cin, fullName);

cout << "Welcome, " << fullName << "!" << endl;

return 0;

}

**✅ Summary Table**

| **Operation** | **Stream** | **Operator** | **Used For** |
| --- | --- | --- | --- |
| Output | cout | << | Printing to console |
| Input | cin | >> | Reading from user |

* ***Variables, Data Types, and Operators***
  + 1. **What are the different data types available in C++ ? Explain with examples.**
* In C++, **data types** define the type of data a variable can hold. They are categorized mainly into:
* **1. Basic (Primitive) Data Types**

These are the fundamental types provided by the language.

| Data Type | Description | Example |
| --- | --- | --- |
| int | Integer numbers (whole numbers) | int age = 25; |
| float | Floating-point numbers (decimal) | float pi = 3.14; |
| double | Double precision floating-point | double bigPi = 3.1415926535; |
| char | Single character | char grade = 'A'; |
| bool | Boolean (true or false) | bool isPassed = true; |
| void | No value (used for functions returning nothing) | void printMessage(); |

* + - **2. Derived Data Types**

Formed from primitive types.

| Data Type | Description | Example |
| --- | --- | --- |
| array | Collection of elements of the same type | int marks[5] = {90, 85, 88, 92, 80}; |
| pointer | Stores memory address of another variable | int\* ptr = &age; |
| function | Functions that return values | int add(int a, int b); |
| reference | Alias for another variable | int &ref = age; |

* + - **3. User-defined Data Types**

Created by the programmer.

| Type | Description | Example |
| --- | --- | --- |
| struct | Group of variables under one name | struct Student { int id; char name[20]; }; |
| class | Blueprint for objects (OOP) | class Car { public: string model; }; |
| union | Stores different data types in same memory location | union Data { int i; float f; }; |
| enum | Enum set of named integral constants | enum Color { RED, GREEN, BLUE }; |
| typedef | Alias for data type | typedef unsigned int uint; |

* + - **4. Modifiers for Data Types**

Used to change the size or sign of basic types.

| Modifier | Use | Example |
| --- | --- | --- |
| signed | Can hold positive or negative values | signed int num = -10; |
| unsigned | Only positive values | unsigned int num = 10; |
| short | Smaller storage size | short int si = 100; |
| long | Larger storage size | long int li = 100000; |
| long long | Even bigger numbers | long long int bigNum = 1234567890; |

**✅ Example Code**

#include <iostream>

using namespace std;

int main() {

int age = 25;

float weight = 65.5;

char grade = 'A';

bool isPassed = true;

struct Student {

int roll;

char name[20];

};

Student s1 = {1, "Alice"};

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

cout << "Weight: " << weight << endl;

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

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

cout << "Student: " << s1.name << ", Roll: " << s1.roll << endl;

return 0;

}

* + 1. **Explain the difference between implicit and explicit type conversion in C++.**
    - In C++, **type conversion** means converting a variable from one data type to another. There are two main types of type conversion:

**🔹 1. Implicit Type Conversion (Type Promotion / Coercion)**

**✅ Definition:**

It is automatically done by the compiler when two different data types are used in an expression.

**✅ Characteristics:**

* Performed silently.
* Follows the type hierarchy (e.g., char → int → float → double).
* No data loss warning (but it may still cause loss of precision).

**✅ Example:**

int x = 10;

float y = 5.5;

float result = x + y; // int (x) is implicitly converted to float

* + - **Explanation:**  
      x (int) is automatically promoted to float before the addition.

**🔹 2. Explicit Type Conversion (Type Casting)**

**✅ Definition:**

It is **manually done by the programmer** using cast operators.

**✅ Syntax:**

(type)expression

* + - Or in C++ style:

int b = static\_cast<int>(a);

**🔁 Summary Table**

| **Feature** | **Implicit Conversion** | **Explicit Conversion** |
| --- | --- | --- |
| Who performs it? | Compiler | Programmer |
| Syntax | Automatic | Manual (e.g., (int)x) |
| Risk of data loss | Possible, but no warning | Controlled by programmer |
| Use case | Simple operations | Precise conversions |
| Example | int + float → float | float a = 9.75; int b = (int)a; |

**✅ Bonus Tip:**

Prefer **explicit casting** in professional code when there's **any chance of precision loss or ambiguity**, especially with pointers or between signed/unsigned types.

* + 1. **What are the different types of operators in C++ ? Provide examples of each.**
    - In C++, **operators** are special symbols or keywords used to perform operations on variables and values. C++ provides a rich set of built-in operators, which are grouped into several categories:
* **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 |
|  |  |  |

* + - **Example:**

int a = 10, b = 3;

cout << a + b; // Output: 13

cout << a % b; // Output: 1

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

Used to compare two values.

| **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:**

if (a != b) {

cout << "Not Equal";

}

* **3.Logical Operators**

Used to perform logical operations.

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

* + - **Example:**

if (a > 0 && b > 0) {

cout << "Both positive";

}

* **4. Assignment Operators**

Used to assign values to variables.

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

* **5. Increment and Decrement Operators**

Used to increase or decrease the value of a variable by 1.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| ++ | Increment | a++ or ++a |
| -- | Decrement | a-- or --a |

* + - **Example:**

int a = 5;

a++; // a becomes 6

++a; // a becomes 7

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

A shorthand for if-else.

| **Syntax** |  |  | **Description** |
| --- | --- | --- | --- |
| condition ? expr1 : expr2 |  |  | If condition is true, execute expr1, else expr2 |

**Example:**

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

**8. Scope Resolution Operator ::**

Used to define a function outside a class or access a global variable.

* + - **Example:**

int a = 10;

class MyClass {

public:

static int a;

};

int MyClass::a = 20;

* **9. Member Access Operators**

Used to access class or structure members.

| **Operator** | **Description** |
| --- | --- |
| . | Access object member |
| -> | Access pointer to object |

* + - **Example:**

obj.name; // using .

ptr->name; // using ->

* **10. Other Operators**
* **sizeof** – returns the size of data type or variable
* **typeid** – returns type info at runtime (used with RTTI)
* **, (comma)** – separates expressions
  + 1. **Explain nested control structures with an example.**

✅ Nested Control Structures in C++

* + - Nested control structures refer to placing one control structure (like if, while, for, or switch) inside another. This allows more complex decision-making and flow control.

**🔸 Types of Control Structures That Can Be Nested:**

1. if inside if
2. for inside for
3. while inside for, etc.

✅ **Example 1: Nested if statements**

#include <iostream>

using namespace std;

int main() {

int marks;

cout << "Enter marks: ";

cin >> marks;

if (marks >= 50) {

if (marks >= 90) {

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

} else if (marks >= 75) {

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

} else {

cout << "Grade: C" << endl;

}

} else {

cout << "Fail" << endl;

}

return 0;

}

**Explanation:**

* Outer if checks if the student passed (marks >= 50)
* Inner if-else determines the **grade level**

✅ **Example 2: Nested for loops (for pattern printing)**

#include <iostream>

using namespace std;

int main() {

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

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

cout << "\* ";

}

cout << endl;

}

return 0;

}

* + - Output:

\*

\* \*

\* \* \*

**Explanation:**

* The outer for loop runs 3 times (rows).
* The inner for loop prints \* as per the current row.

**✅ Why Use Nested Structures?**

* Handle **complex logic**.
* **Iterate** through multi-dimensional data (like matrices).
* **Create menus** or multi-level decision flows.
* ***Functions and Scope***
  1. **What is a function in C++? Explain the concept of function declaration, definition, and calling.**

**✅ What is a Function in C++?**

* + - A function in C++ is a block of reusable code that performs a specific task. Functions help organize programs into modular, manageable sections and reduce code repetition.

**🔹 Types of Functions:**

1. **Library (predefined) functions** – like sqrt(), cin, cout
2. **User-defined functions** – created by the programmer

**✅ Function Components:**

**1. Function Declaration (Prototype)**

Tells the compiler about the function **name**, **return type**, and **parameters** before its actual definition.

int add(int a, int b); // Declaration

**2. Function Definition**

Contains the actual body/code of the function.

int add(int a, int b) {

return a + b;

}

**3. Function Call**

Executes the function by passing values (arguments) to it.

int result = add(5, 3); // Calling the function

✅ **Complete Example:**

#include <iostream>

using namespace std;

// 1. Function Declaration

int add(int, int);

int main() {

// 3. Function Call

int sum = add(10, 20);

cout << "Sum is: " << sum << endl;

return 0;

}

// 2. Function Definition

int add(int a, int b) {

return a + b;

}

* + - **Output:**

Sum is: 30

**🔸 Why Use Functions?**

* **Reusability** – write once, use many times
* **Modularity** – break complex problems into smaller parts
* **Clarity** – code becomes easier to read and debug

**✅ Optional: Function Without Parameters or Return**

void greet() {

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

}

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

✅ Scope of Variables in C++

* + - In C++, the scope of a variable refers to the region of the program where the variable can be accessed or used.

**🔹 Types of Variable Scope:**

| **Scope Type** | **Where it's declared** | **Accessible in** |
| --- | --- | --- |
| **Local Scope** | Inside a function or block | Only within that function/block |
| **Global Scope** | Outside all functions | Throughout the entire program |
| **Function Parameters** | Inside function parameter list | Only within that function |
| **Block Scope** | Inside {} braces | Only within those braces |
| **Class/Namespace Scope** | Inside class or namespace | Depends on access modifiers |

**✅ 1. Local Variables (Local Scope)**

* Declared **inside** a function or block { }
* **Accessible only within** that function/block
* **Destroyed** once the function/block ends
  + - **Example:**

void show() {

int a = 10; // local variable

cout << a; // Valid here

}

// cout << a; // ❌ Invalid here (out of scope)

**✅ 2. Global Variables (Global Scope)**

* Declared **outside all functions**
* **Accessible from any function** in the file (after declaration)
* **Exists throughout the program**
  + - **Example:**

#include <iostream>

using namespace std;

int x = 100; // global variable

void display() {

cout << x << endl; // Accessing global variable

}

int main() {

cout << x << endl; // Accessing global variable

display();

return 0;

}

* + - **Output:**

100

100

**🔁 Difference Between Local and Global Scope:**

| **Feature** | **Local Variable** | **Global Variable** |
| --- | --- | --- |
| **Declared In** | Inside a function/block | Outside all functions |
| **Accessible In** | Only within its block/function | Entire program (after declaration) |
| **Lifetime** | Exists until function ends | Exists till program ends |
| **Memory Use** | Created and destroyed repeatedly | Created once, uses memory always |
| **Access Priority** | Takes priority over global var | Can be overridden by local var |

**⚠️ Note:**

If a **local and global variable have the same name**, the **local variable hides the global** one inside that scope. To access the global variable, use the **scope resolution operator ::**.

int x = 10;

int main() {

int x = 20;

cout << x << endl; // Prints local x (20)

cout << ::x << endl; // Prints global x (10)

return 0;

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

**✅ Recursion in C++**

* + - **Recursion** is a programming technique where a function **calls itself** directly or indirectly to solve a problem. It is useful for problems that can be **broken down into smaller sub-problems** of the same type.

**🔁 How Recursion Works:**

Each recursive function call creates a **new copy** of the function on the **call stack**.

To prevent infinite recursion, two main components are required:

1. **Base Case** – The condition where the recursion stops.
2. **Recursive Case** – The part where the function calls itself.

**✅ Example: Factorial Using Recursion**

The **factorial** of a number n is:  
n! = n × (n-1) × (n-2) × ... × 1

**🔸 Code:**

#include <iostream>

using namespace std;

int factorial(int n) {

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

return 1;

else

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

}

int main() {

int num = 5;

cout << "Factorial of " << num << " is " << factorial(num);

return 0;

}

* + - **Output:**

Factorial of 5 is 120

**🔍 How It Works (Step-by-Step):**

factorial(5)

= 5 \* factorial(4)

= 5 \* 4 \* factorial(3)

= 5 \* 4 \* 3 \* factorial(2)

= 5 \* 4 \* 3 \* 2 \* factorial(1)

= 5 \* 4 \* 3 \* 2 \* 1

= 120

**✅ Other Common Recursive Examples:**

* Fibonacci series
* Tower of Hanoi
* Binary search
* Tree traversals

**⚠️ Points to Remember:**

* Always define a **base case** to stop recursion.
* Recursion may consume more memory (due to stack usage).
* For large inputs, consider **iteration** to avoid stack overflow.
  1. **What are function prototypes in C++ ? Why are they used?**

**✅ What are Function Prototypes in C++?**

* + - A **function prototype** is a **declaration of a function** that tells the compiler:
* the **function name**
* its **return type**
* and its **parameters** (types and order)

It appears **before the function is actually defined** and is usually written **at the top of the program** or **in a header file**.

**🔹 Syntax of a Function Prototype:**

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

* + - **Example:**

int add(int, int); // Function prototype

This tells the compiler that there's a function named add that takes two int arguments and returns an int.

**✅ Why Are Function Prototypes Used?**

| **Purpose** | **Description** |
| --- | --- |
| ✅ **Forward Declaration** | Tells the compiler about the function before its actual definition. |
| ✅ **Type Checking** | Ensures correct number and type of arguments are passed during a function call. |
| ✅ **Top-Down Programming** | Allows main() or other functions to call a function that is defined later. |
| ✅ **Modularity** | Helps separate **interface** (prototype) from **implementation** (definition), useful in header files. |

**✅ Example With and Without Function Prototype:**

**🔸 With Function Prototype**

#include <iostream>

using namespace std;

int add(int, int); // Function prototype

int main() {

cout << add(3, 4); // Function call

return 0;

}

int add(int a, int b) { // Function definition

return a + b;

}

**🔸 Without Function Prototype (but function defined before main)**

#include <iostream>

using namespace std;

int add(int a, int b) { // Function defined before main

return a + b;

}

int main() {

cout << add(3, 4);

return 0;

}

**⚠️ When is a Function Prototype Required?**

* ✅ **Required** if the function is defined **after main()**
* ❌ **Optional** if the function is **defined before it's called**
* ***Control Flow Statements***
  1. **What are conditional statements in C++ ? Explain the if-else and switch statements.**

**✅ Conditional Statements in C++**

* + - Conditional statements allow a program to make decisions and execute different parts of the code based on conditions (true/false expressions). They are used to control the flow of execution.

**🔹 1. if Statement**

The if statement checks a condition. If the condition is true, the block inside the if executes

int x = 10;

if (x > 5) {

cout << "x is greater than 5";

}

**🔹 2. if-else Statement**

Used when there are **two possible paths**. If the condition is true, the if block executes, otherwise the else block.

int x = 3;

if (x > 5) {

cout << "x is greater than 5";

} else {

cout << "x is 5 or less";

}

**🔹 3. if-else if-else Ladder**

Used to test **multiple conditions** in sequence.

int x = 0;

if (x > 0) {

cout << "Positive";

} else if (x < 0) {

cout << "Negative";

} else {

cout << "Zero";

}

**4. switch Statement**

Used to **select one of many blocks** of code to be executed. It works with **integers, characters, and enumerated types**.

int day = 2;

switch(day) {

case 1:

cout << "Monday";

break;

case 2:

cout << "Tuesday";

break;

case 3:

cout << "Wednesday";

break;

default:

cout << "Invalid day";

}

🔸 break: Prevents the execution from falling through to the next case.

🔸 default: Executes if none of the case values match.

**✅ Summary**

| **Statement** | **Purpose** |
| --- | --- |
| if | Executes a block if the condition is true |
| if-else | Chooses between two blocks |
| if-else if | Tests multiple conditions |
| switch | Selects from many options (cases) based on a value |

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

**🔁 Difference Between for, while, and do-while Loops in C++**

Loops are used to **repeat a block of code** multiple times based on a condition. Here's a comparison of the three main types:

**🔹 1. for Loop**

* Used when the **number of iterations is known** in advance.
* All loop control elements (initialization, condition, and increment/decrement) are in one line.

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

cout << i << " ";

}

**📌 Output:** 1 2 3 4 5

**🔹 2. while Loop**

* Used when the **number of iterations is not known** in advance.
* Condition is checked **before** entering the loop.

int i = 1;

while (i <= 5) {

cout << i << " ";

i++;

}

**📌 Output:** 1 2 3 4 5

**🔹 3. do-while Loop**

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

int i = 1;

do {

cout << i << " ";

i++;

} while (i <= 5);

**📌 Output:** 1 2 3 4 5

✅ **Key Differences**

| **Feature** | **for Loop** | **while Loop** | **do-while Loop** |
| --- | --- | --- | --- |
| Condition Check | At the beginning | At the beginning | At the end |
| Executes at least once | ❌ No (if condition is false) | ❌ No (if condition is false) | ✅ Yes |
| Best Use Case | Known number of iterations | Unknown number of iterations | At least one execution is needed |

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

**🔄 break and continue Statements in C++ Loops**

* + - These are **loop control statements** used to alter the flow of loops (for, while, do-while):

**🔹 1. break Statement**

* Used to **exit the loop immediately**, regardless of the loop condition.
* Commonly used to **stop a loop early** when a certain condition is met.

**✅ Example:**

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

if (i == 5) {

break; // Exit the loop when i is 5

}

cout << i << " ";

}

📌 **Output**: 1 2 3 4

**🔹 2. continue Statement**

* Skips the **current iteration** of the loop and jumps to the **next iteration**.
* Useful when you want to **skip certain values** but continue looping.

**✅ Example:**

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

if (i == 3) {

continue; // Skip when i is 3

}

cout << i << " ";

}

**✅ Summary Table**

| **Statement** |  |  |  | **Description** | **Effect** |
| --- | --- | --- | --- | --- | --- |
| break |  |  |  | Exits the loop completely | Stops loop execution immediately |
| continue |  |  |  | Skips current iteration, continues with next | Skips rest of loop body for that iteration |

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

**🔁 Nested Control Structures in C++**

* + - Nested control structures are control structures placed inside another control structure. This includes nesting loops within loops, or conditional statements within loops or other conditionals.

**✅ Common Nested Structures**

1. **if inside if** (nested if)
2. **loop inside loop** (nested loops)
3. **loop inside if**, or **if inside loop**

**🔹 Example 1: Nested if Statement**

int number = 10;

if (number > 0) {

if (number % 2 == 0) {

cout << "Positive Even Number";

} else {

cout << "Positive Odd Number";

}

}

**📌 Output:** Positive Even Number

**🔹 Example 2: Nested for Loop**

Used often in printing patterns or working with matrices.

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

for (int j = 1; j <= 2; j++) {

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

}

cout << endl;

}

**📌 Output:**

(1,1) (1,2)

(2,1) (2,2)

(3,1) (3,2)

🔹 **Example 3: if inside a for loop**

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

if (i % 2 == 0) {

cout << i << " is even\n";

} else {

cout << i << " is odd\n";

}

}

**✅ Summary**

| **Structure** |  |  | **Meaning** |
| --- | --- | --- | --- |
| Nested if |  |  | One condition inside another condition |
| Nested loops |  |  | Loop runs inside another loop |
| Mixed nesting |  |  | Loops and conditionals used together |

* ***Arrays and Strings***

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

**📚 What are Arrays in C++?**

* + - An array in C++ is a collection of elements of the same data type, stored in contiguous memory locations. Arrays allow storing and accessing multiple values using a single variable name with an index.

✅ **Syntax of Array Declaration**

data\_type array\_name[size];

**🔹 Example:**

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

cout << numbers[0]; // Output: 10

Indexing starts from 0, so numbers[0] is the first element.

**🆚 Single-Dimensional vs Multi-Dimensional Arrays**

| **Feature** | **Single-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| Structure | A single row of elements | Table-like structure (rows and columns) |
| Declaration | int arr[5]; | int arr[2][3]; (2 rows, 3 columns) |
| Access Method | arr[i] | arr[i][j] |
| Use Case | Storing a list of values | Representing matrices, tables, grids, etc. |

**🔹 1. Single-Dimensional Array Example**

int marks[4] = {85, 90, 78, 92};

for (int i = 0; i < 4; i++) {

cout << "marks[" << i << "] = " << marks[i] << endl;

}

**📌 Output:**

marks[0] = 85

marks[1] = 90

marks[2] = 78

marks[3] = 92

🔹 **2. Multi-Dimensional Array Example (2D)**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

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

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

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

}

cout << endl;

}

**📌 Output:**

1 2 3

4 5 6

**✅ Summary**

* **Single-Dimensional Array**: Linear collection of elements.
* **Multi-Dimensional Array**: Collection of arrays arranged in a grid (2D, 3D, etc.).
* Arrays are useful when you need to store **multiple values of the same type** efficiently.

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

**📘 String Handling in C++**

* + - In C++, **strings** are sequences of characters used to store and manipulate text. C++ supports strings in **two main ways**:

**✅ 1. C-style Strings (char arrays)**

* Based on arrays of characters.
* End with a **null character ('\0')** to mark the end of the string.
* Require the use of <cstring> header for string functions.

**🔹 Example:**

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char name[20] = "Alice";

cout << "Name: " << name << endl;

cout << "Length: " << strlen(name) << endl;

return 0;

}

**📌 Output:**

Name: Alice

Length: 5

**✅ 2. C++ Strings (using string class)**

* Part of the **Standard Template Library (STL)**.
* Easier and safer to use than char arrays.
* Requires #include <string>.

**🔹 Example:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name = "Alice";

cout << "Name: " << name << endl;

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

name += " Smith"; // Concatenation

cout << "Full Name: " << name << endl;

return 0;

}

**📌 Output:**

Name: Alice

Length: 5

Full Name: Alice Smith

✳️ **Common String Operations in C++ (string class)**

| **Operation** | **Syntax** | **Example** |
| --- | --- | --- |
| Concatenation | str1 + str2 | s = "Hi " + name; |
| Length of string | str.length() | int len = str.length(); |
| Substring | str.substr(pos, len) | str.substr(0, 4) |
| Access character | str[index] | str[2] = 'x'; |
| Comparison | ==, !=, <, > | if (str1 == str2) |
| Input string with spaces | getline(cin, str) | getline(cin, fullName); |

🔍 **Example: Reading Full Name and Manipulating**

#include <iostream>

#include <string>

using namespace std;

int main() {

string fullName;

cout << "Enter your full name: ";

getline(cin, fullName);

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

cout << "Your name has " << fullName.length() << " characters." << endl;

return 0;

}

**✅ Summary**

| **Feature** | **C-style Strings** | **string Class (STL)** |
| --- | --- | --- |
| Data Type | char array | string |
| Header Required | <cstring> | <string> |
| Safer to Use | ❌ Manual management | ✅ Built-in functions |
| Recommended | For legacy/low-level | ✅ Preferred in modern C++ |

1. **How are arrays initialized in C++? Provide examples of both1D and 2Darrays.**

**🔢 Array Initialization in C++**

* + - Arrays in C++ can be initialized at the time of declaration or later in the program. Let’s look at how to initialize both 1D and 2D arrays with examples.

**✅ 1. One-Dimensional (1D) Array Initialization**

🔹 1.1 Static Initialization (at declaration)

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

📌 This sets:

* numbers[0] = 10
* numbers[1] = 20, etc.

You can also omit the size:

int numbers[] = {10, 20, 30};

**🔹 1.2 Partial Initialization**

int numbers[5] = {1, 2}; // Remaining elements are set to 0

**📌 Result:** 1 2 0 0 0

**🔹 1.3 Runtime Initialization**

int numbers[3];

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

cin >> numbers[i];

}

**✅ 2. Two-Dimensional (2D) Array Initialization**

**🔹 2.1 Static Initialization**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

**📌 This sets:**

* matrix[0][0] = 1, matrix[1][2] = 6, etc.

**🔹 2.2 Flattened Initialization**

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

Fills the array row by row.

**🔹 2.3 Partial Initialization**

int matrix[2][3] = {

{1}, // Fills row 0 with 1, 0, 0

{4, 5} // Fills row 1 with 4, 5, 0

};

**📌 Result:**

1 0 0

4 5 0

**✅ Summary Table**

| **Type** |  | **Initialization Syntax** |  |  | **Example Values** |
| --- | --- | --- | --- | --- | --- |
| 1D array |  | int arr[3] = {1, 2, 3}; |  |  | 1 2 3 |
| 2D array |  | int arr[2][2] = {{1, 2}, {3, 4}}; |  |  | 1 2  3 4 |
| Partial |  | int arr[4] = {1, 2}; |  |  | 1 2 0 0 |

1. **Explain string operations and functions in C++.**
   * + In C++, **strings** are sequences of characters. The **Standard Template Library (STL)** provides a std::string class that makes string manipulation easier and more powerful compared to using C-style character arrays.

**✅ Basic String Operations in C++**

* 1. **Declaration and Initialization:**

std::string s1 = "Hello";

std::string s2("World");

* 1. **Concatenation (+):**

std::string result = s1 + " " + s2; // "Hello World"

* 1. **Appending (+=):**

s1 += " Everyone"; // s1 becomes "Hello Everyone"

* 1. **Accessing Characters ([] or .at()):**

char c = s1[0]; // 'H'

char d = s1.at(1); // 'e'

* 1. **Length of String (.length() or .size()):**

int len = s1.length(); // or s1.size()

* 1. **Comparison Operators (==, !=, <, etc.):**

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

* 1. **Substring (.substr()):**

std::string sub = s1.substr(0, 5); // "Hello"

* 1. **Clear String (.clear()):**

s1.clear(); // s1 becomes ""

* 1. **Empty Check (.empty()):**

if (s1.empty()) { ... }

**✅ Useful String Functions**

| **Function** | **Description** |
| --- | --- |
| s.find("text") | Returns the index of first occurrence or string::npos if not found |
| s.rfind("text") | Finds last occurrence of substring |
| s.erase(pos, len) | Removes len characters from pos |
| s.insert(pos, "str") | Inserts substring at specified position |
| s.replace(pos, len, "str") | Replaces part of the string |
| getline(cin, s) | Reads full line of input (with spaces) into string |
| stoi(s) | Converts string to integer |
| to\_string(num) | Converts number to string |
| transform(s.begin(), s.end(), s.begin(), ::toupper) | Converts string to uppercase |
| transform(s.begin(), s.end(), s.begin(), ::tolower) | Converts string to lowercase |

**✅ Example Program**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name = "Alice";

string greeting = "Hello, " + name;

cout << greeting << endl; // Hello, Alice

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

greeting.replace(7, 5, "Bob"); // Replace "Alice" with "Bob"

cout << greeting << endl; // Hello, Bob

if (greeting.find("Bob") != string::npos) {

cout << "Name found!" << endl;

}

return 0;

}

**📝 Summary**

* std::string offers a wide range of powerful and easy-to-use functions.
* Prefer std::string over C-style strings (char[]) in modern C++ for safety and simplicity.
* Functions like substr, find, replace, and insert make string manipulation easier.
* ***Introduction to Object-Oriented Programming***

1. **Explain the key concepts of Object-Oriented Programming(OOP).**
   * + Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of **"objects"**, which contain **data** (attributes) and **code** (methods) that work on the data. OOP helps organize and structure software programs to make them more **modular**, **reusable**, and **easier to maintain**.

Here are the **key concepts of OOP**:

**1. Class**

* A **class** is a blueprint or template for creating objects.
* It defines a set of attributes (variables) and methods (functions) that the created objects will have.
  + - **Example in C++:**

class Car {

public:

string brand;

void drive() {

cout << "Driving..." << endl;

}

};

**2. Object**

* An **object** is an instance of a class.
* It represents a specific implementation of the class with actual values.
  + - **Example:**

Car myCar;

myCar.brand = "Toyota";

myCar.drive();

**3. Encapsulation**

* Encapsulation is the concept of **hiding internal details** and exposing only what is necessary.
* This is done using **access modifiers** like private, public, and protected.

**Benefits:** Data protection, reduced complexity.

* + - **Example:**

class Person {

private:

int age;

public:

void setAge(int a) {

age = a;

}

int getAge() {

return age;

}

};

**4. Abstraction**

* Abstraction means showing only **essential features** and hiding unnecessary details.
* It helps reduce complexity and improve focus.
  + - **Example:**
* A car's user interacts with the steering and pedals but doesn't need to know how the engine works.

**5. Inheritance**

* Inheritance allows a class (child or derived class) to **inherit** attributes and methods from another class (parent or base class).
* Promotes **code reusability**.
  + - **Example:**

class Animal {

public:

void eat() {

cout << "Eating..." << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Barking..." << endl;

}

};

**6. Polymorphism**

* Polymorphism means **many forms** – the ability of different objects to respond in different ways to the same method call.
* Achieved through **function overloading** and **method overriding**.
  + - **Example (Method Overriding):**

class Animal {

public:

virtual void sound() {

cout << "Animal sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Bark" << endl;

}

};

**Summary Table:**

| **Concept** |  |  |  | **Description** |
| --- | --- | --- | --- | --- |
| **Class** |  |  |  | Blueprint for creating objects |
| **Object** |  |  |  | Instance of a class |
| **Encapsulation** |  |  |  | Hiding data and providing access via methods |
| **Abstraction** |  |  |  | Showing essential features, hiding details |
| **Inheritance** |  |  |  | Reusing code by inheriting from base classes |
| **Polymorphism** |  |  |  | Same interface, different behavior |

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

**✅ Classes and Objects in C++**

In C++, classes and objects are the fundamental building blocks of Object-Oriented Programming (OOP).

**🔷 Class**

A **class** is a user-defined data type that acts as a **blueprint** for creating **objects**. It groups **variables** (called **data members**) and **functions** (called **member functions**) into a single unit.

**🔷 Object**

An **object** is a **real-world instance** of a class. When you create an object of a class, it gets its own **copy of the data members**, and you can use it to call member functions.

**📌 Syntax of Class and Object in C++**

#include <iostream>

using namespace std;

// Define a class

class Car {

public:

string brand;

int year;

void display() {

cout << "Brand: " << brand << ", Year: " << year << endl;

}

};

int main() {

// Create an object of the class

Car myCar;

// Set values to object attributes

myCar.brand = "Toyota";

myCar.year = 2022;

// Call a method using the object

myCar.display();

return 0;

}

**🔍 Explanation:**

| **Term** | **Description** |
| --- | --- |
| class Car | Defines a class named Car. |
| brand and year | Data members (attributes) of the class. |
| display() | Member function that prints the car details. |
| Car myCar | An object of the class Car. |
| myCar.brand = "Toyota"; | Setting value to the object's data member. |
| myCar.display(); | Calling the function using the object. |

1. **What is in heritance in C++? Explain with an example.**

**✅ What is Inheritance in C++?**

* + - **Inheritance** is one of the core concepts of **Object-Oriented Programming (OOP)**. It allows a **new class** (called the **derived class**) to **inherit** properties and behaviors (data members and member functions) from an **existing class** (called the **base class**).

**🔹 Why Use Inheritance?**

* Promotes **code reusability**
* Supports **hierarchical classification**
* Enables **polymorphism**

**🔹 Syntax of Inheritance:**

class BaseClass {

// members

};

class DerivedClass : access\_modifier BaseClass {

// additional members

};

The access\_modifier can be:

* public: public and protected members of the base class remain public/protected in the derived class.
* protected: public members become protected.
* private: all members become private (least common in inheritance).

**📌 Example: Public Inheritance**

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

cout << "This animal eats food." << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "The dog barks." << endl;

}

};

int main() {

Dog myDog;

// Calling base class function

myDog.eat();

// Calling derived class function

myDog.bark();

return 0;

}

**🔍 Explanation:**

| **Concept** | **Description** |
| --- | --- |
| Animal | Base class with eat() method |
| Dog | Derived class that inherits from Animal using public inheritance |
| myDog.eat() | Inherited method from Animal |
| myDog.bark() | Method defined in Dog class |

**🔄 Types of Inheritance in C++:**

1. **Single Inheritance** – One base and one derived class.
2. **Multiple Inheritance** – A class inherits from **more than one** base class.
3. **Multilevel Inheritance** – Derived class becomes the base for another class.
4. **Hierarchical Inheritance** – Multiple classes inherit from the same base class.
5. **Hybrid Inheritance** – Combination of more than one type of inheritance.
6. **What is encapsulation in C++ ? How is it achieved in classes?**

**✅ What is Encapsulation in C++?**

* + - **Encapsulation** is one of the fundamental principles of **Object-Oriented Programming (OOP)**. It refers to **binding data (variables)** and the **code (functions)** that operates on the data into a single unit — typically a **class** — and **restricting direct access** to some of the object's components.

**🔒 Key Idea of Encapsulation:**

* Protect data from **unauthorized access or modification**.
* Provide **controlled access** using **public methods** (getters and setters).
* Improve **modularity**, **maintainability**, and **security**.

**🛠️ How is Encapsulation Achieved in C++?**

Encapsulation is achieved using:

1. **Access Specifiers:**
   * private: Accessible only within the class.
   * public: Accessible from outside the class.
   * protected: Accessible within the class and its derived classes.
2. **Setters and Getters**:
   * Used to **read and update private data members** safely.

**📌 Example: Encapsulation in C++**

#include <iostream>

using namespace std;

class Employee {

private:

int salary; // Private data member

public:

// Setter function (write access)

void setSalary(int s) {

if (s > 0)

salary = s;

else

cout << "Invalid salary!" << endl;

}

// Getter function (read access)

int getSalary() {

return salary;

}

};

int main() {

Employee emp;

emp.setSalary(50000); // Setting value through setter

cout << "Salary: " << emp.getSalary() << endl; // Getting value through getter

return 0;

}

**🔍 Explanation:**

| **Part** |  | **Purpose** |
| --- | --- | --- |
| private int salary |  | Salary is hidden from outside access |
| setSalary() |  | Validates and sets the salary (controlled write access) |
| getSalary() |  | Returns the salary (controlled read access) |
| emp.salary = 50000 |  | ❌ Invalid! Direct access is blocked |
| emp.setSalary(50000) |  | ✅ Valid and safe method of setting data |

**✅ Benefits of Encapsulation:**

* Protects data integrity
* Simplifies maintenance
* Allows internal changes without affecting external code
* Supports modular and clean code design