***Module 3. Introduction to OOPS Programming***

1. Introduction to C++.

**C++** is a **general-purpose, middle-level, object-oriented programming language** developed by **Bjarne Stroustrup** at Bell Labs in **1979** as an extension of the C language. It supports both **procedural** and **object-oriented** programming, making it a **multi-paradigm** language.

**Key Features of C++**

1. **Multi-Paradigm Support**
   * Procedural programming (like C)
   * Object-oriented programming (OOP)
   * Generic programming (templates)
2. **Compiled Language**
   * Source code is compiled into machine code for fast execution.
3. **Object-Oriented Features**
   * Encapsulation
   * Inheritance
   * Polymorphism
   * Abstraction
4. **Low-Level Manipulation**
   * Can work closely with hardware (pointers, memory management).
5. **Portability**
   * Code can run on different platforms with minimal changes.

**What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?**

| **Aspect** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Basic Concept** | Program is organized as a set of **functions** (procedures) operating on data. | Program is organized around **objects**, which combine **data** and **methods**. |
| **Focus** | Focuses on **functions** (logic) first, then on data. | Focuses on **data** (objects) first, then on methods. |
| **Data Handling** | Data is **separate** from functions; functions take data as input and return results. | Data and functions are **bundled together** in objects. |
| **Data Security** | Data is generally **global** or shared between functions, making it less secure. | Data is **encapsulated** inside objects; access is controlled via methods. |
| **Code Reusability** | Code reuse is mainly through **function calls** and libraries. | Code reuse is through **classes** and **inheritance**. |
| **Modularity** | Achieved by dividing program into functions. | Achieved by dividing program into objects and classes. |
| **Examples** | C, Pascal, Fortran | Java, C++, Python (when using OOP features) |
| **Real-World Mapping** | Less direct mapping to real-world entities. | Models real-world entities as objects with properties and behaviors. |
| **Maintenance** | More difficult to maintain for large projects (risk of side effects from shared data). | Easier to maintain and extend due to encapsulation and modularity. |

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

| **Advantage** | **Explanation** | **Why POP is Weaker Here** |
| --- | --- | --- |
| **1. Encapsulation** | In OOP, data (attributes) and methods (functions) are bundled into objects, and access to data is controlled using access modifiers (private, protected, public). | In POP, data is often global or passed between functions, so it’s more exposed and prone to accidental modification. |
| **2. Data Security** | Objects can hide their internal data from outside interference using encapsulation. | POP has less control over who can access and modify data. |
| **3. Reusability** | Classes can be reused across programs, and **inheritance** allows creating new classes based on existing ones. | POP reuses code mainly via function libraries, but it’s harder to extend without modifying existing code. |
| **4. Modularity** | Programs are divided into self-contained objects, making code easier to organize, debug, and maintain. | POP organizes code into functions, but related data and logic are often scattered. |
| **5. Maintainability** | Encapsulation and modularity make it easier to update or fix parts of the program without affecting other parts. | POP’s reliance on shared data increases the risk of unintended side effects when changes are made. |
| **6. Abstraction** | Complex systems can be modeled using classes that represent real-world entities, hiding unnecessary details. | POP has no direct abstraction mechanism—requires manual structuring of functions and data. |
| **7. Polymorphism** | OOP supports **method overloading** and **overriding**, allowing objects to be treated as instances of their parent class. | POP doesn’t have built-in polymorphism—functions must be explicitly named differently for each use. |
| **8. Real-World Modeling** | OOP naturally maps to real-world concepts (objects, attributes, behaviors), making program design more intuitive. | POP requires translating real-world scenarios into step-by-step functions, which can be less intuitive. |

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

## **Steps to Set Up a C++ Development Environment**

### ****1. Choose a Compiler****

A **compiler** converts your C++ code into machine code.

* **Popular compilers**:
  + **GCC** (GNU Compiler Collection) – Linux, Windows (via MinGW)
  + **MSVC** (Microsoft Visual C++) – Windows
  + **Clang** – macOS, Linux, Windows

✅ For beginners, **GCC** (MinGW for Windows) or **MSVC** is recommended.

### ****2. Install a Text Editor or IDE****

An **IDE** (Integrated Development Environment) includes a code editor, compiler integration, debugger, and more.

* **Popular IDEs**:
  + **Code::Blocks** (Beginner-friendly, free)
  + **Dev-C++** (Lightweight)
  + **Visual Studio** (Full-featured, Windows)
  + **CLion** (JetBrains, paid but powerful)
  + **VS Code** (Lightweight, needs compiler installed separately)

### ****3. Install the Compiler****

* **Windows (GCC via MinGW):**
  1. Download MinGW from https://sourceforge.net/projects/mingw/.
  2. Install it and add the **bin** folder to your system’s **PATH** environment variable.
  3. Check installation in Command Prompt:

css

g++ --version

* **macOS:**
  1. Install Xcode Command Line Tools:

lua

xcode-select --install

* **Linux:**
  1. Install GCC:

nginx

sudo apt install g++

### ****4. Write Your First Program****

Example:

cpp

#include <iostream>

using namespace std;

int main() {

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

return 0;

}

### ****5. Compile and Run****

* **From terminal/command prompt:**

bash

g++ hello.cpp -o hello

./hello

* **From an IDE:** Press **Run** or **Build and Run** button.

### ****6. (Optional) Set Up Debugging****

* IDEs like Visual Studio, CLion, or Code::Blocks have built-in debuggers.
* You can also use **gdb** with GCC:

bash

g++ -g hello.cpp -o hello

gdb ./hello

### ****7. Keep Everything Updated****

* Update your compiler to support the latest C++ standards (C++11, C++14, C++17, C++20).
* Keep your IDE updated for better performance and features.

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

## **1. Output in C++ (**cout**)**

* **Purpose:** Display information on the screen.
* **Syntax:**

cpp

cout << data;

* **Example:**

cpp

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl; // Output text

cout << "The number is: " << 42 << endl; // Output variable value

return 0;

}

* **Notes:**
  + << is the **insertion operator** (inserts data into the output stream).
  + endl moves to the next line.

## **2. Input in C++ (**cin**)**

* **Purpose:** Read input from the user (keyboard).
* **Syntax:**

cpp

cin >> variable;

* **Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age; // Input from user

cout << "You are " << age << " years old." << endl;

return 0;

}

* **Notes:**
  + >> is the **extraction operator** (extracts data from the input stream).
  + By default, cin stops reading at whitespace for strings.

## **3. Other Useful I/O Functions**

* **getline()** – Reads a full line (including spaces).

cpp

string name;

cout << "Enter your full name: ";

getline(cin, name);

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

* **File I/O** – Using ifstream (input) and ofstream (output).

cpp

#include <iostream>

#include <fstream>

using namespace std;

int main() {

ofstream outFile("test.txt");

outFile << "Writing to a file." << endl;

outFile.close();

ifstream inFile("test.txt");

string line;

getline(inFile, line);

cout << "Read from file: " << line << endl;

inFile.close();

return 0;

}

✅ **Summary Table:**

| **Operation** | **Object** | **Operator** | **Example** |
| --- | --- | --- | --- |
| Output | cout | << | cout << "Hi"; |
| Input | cin | >> | cin >> x; |
| Line Input | getline() | – | getline(cin, str); |
| File Output | ofstream | << | outFile << "Data"; |
| File Input | ifstream | >> / getline() | inFile >> data; |

### Variables, Data Types, and Operators

## **1. Variables in C++**

A **variable** is a named storage location in memory that holds a value which can change during program execution.

**Rules for naming variables:**

* Must start with a letter or underscore.
* Cannot use reserved keywords.
* Case-sensitive.
* Can contain letters, digits, and underscores.

**Syntax:**

cpp

datatype variableName = value;

**Example:**

cpp

int age = 20; // Integer variable

float price = 99.5; // Floating-point variable

char grade = 'A'; // Character variable

## **2. Data Types in C++**

C++ supports different data types to store various kinds of data.

| **Category** | **Data Type** | **Example** |
| --- | --- | --- |
| **Basic** | int, float, double, char, bool | int count = 5; |
| **Derived** | Arrays, Functions, Pointers, References | int arr[5]; |
| **User-defined** | struct, class, enum | class Student { ... }; |

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int num = 10; // Integer

double pi = 3.1416; // Double

char letter = 'C'; // Character

bool isOn = true; // Boolean

cout << num << ", " << pi << ", " << letter << ", " << isOn << endl;

return 0;

}

## **3. Operators in C++**

Operators are symbols that perform operations on variables and values.

### ****a) Arithmetic Operators****

+ (Addition), - (Subtraction), \* (Multiplication), / (Division), % (Modulus)

cpp

int a = 10, b = 3;

cout << a + b; // 13

cout << a % b; // 1

### ****b) Relational Operators****

==, !=, <, >, <=, >=

cpp

cout << (a > b); // 1 (true)

### ****c) Logical Operators****

&& (AND), || (OR), ! (NOT)

cpp

cout << (a > 5 && b < 5); // 1 (true)

### ****d) Assignment Operators****

=, +=, -=, \*=, /=, %=

cpp

a += 5; // same as a = a + 5

### ****e) Increment/Decrement Operators****

++ (Increment), -- (Decrement)

cpp

a++; // Post-increment

++b; // Pre-increment

### ****f) Conditional (Ternary) Operator****

condition ? expr1 : expr2

cpp

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

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

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

These are the fundamental types built into C++.

| **Data Type** | **Size (approx.)** | **Example** | **Description** |
| --- | --- | --- | --- |
| int | 4 bytes | int age = 20; | Stores integers (whole numbers). |
| float | 4 bytes | float price = 99.5; | Stores single-precision floating-point numbers. |
| double | 8 bytes | double pi = 3.141592; | Stores double-precision floating-point numbers. |
| char | 1 byte | char grade = 'A'; | Stores a single character. |
| bool | 1 byte | bool isOn = true; | Stores true or false. |
| wchar\_t | 2 or 4 bytes | wchar\_t letter = L'Ω'; | Stores wide characters (Unicode). |

## **2. Derived Data Types**

These are built from basic data types.

| **Type** | **Example** | **Description** |
| --- | --- | --- |
| **Array** | int marks[5] = {90, 85, 88, 92, 76}; | Collection of elements of the same type. |
| **Pointer** | int \*p; p = &age; | Stores the memory address of a variable. |
| **Function** | int sum(int a, int b) { return a + b; } | Block of code that performs a task and returns a value. |
| **Reference** | int &ref = age; | Alias for another variable. |

## **3. User-Defined Data Types**

These are created by programmers to represent complex entities.

| **Type** | **Example** | **Description** |
| --- | --- | --- |
| **Structure (struct)** | struct Student { int roll; char name[20]; }; | Groups different data types together. |
| **Class (class)** | class Car { public: string model; int year; }; | Encapsulates data and functions. |
| **Enumeration (enum)** | enum Color { Red, Green, Blue }; | Defines named integer constants. |
| **Typedef / using** | typedef unsigned int uint; | Creates a new name for an existing type. |

## **4. Void Type**

* **void** means “no value” or “empty”.
* Used for:
  + Functions that don’t return a value:

cpp

void greet() {

cout << "Hello!";

}

* + Generic pointers (void \*) that can store the address of any data type.

## **Example Program Using Different Data Types**

cpp

#include <iostream>

using namespace std;

struct Student {

int roll;

char name[20];

};

int main() {

// Basic

int age = 20;

double pi = 3.1416;

char grade = 'A';

bool isPass = true;

// Derived

int marks[3] = {85, 90, 78};

int \*p = &age;

// User-defined

Student s1 = {101, "John"};

enum Color { Red, Green, Blue };

Color favorite = Blue;

// Output

cout << "Age: " << age << ", Pi: " << pi << ", Grade: " << grade << endl;

cout << "Is Pass: " << isPass << endl;

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

cout << "Pointer to age: " << \*p << endl;

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

cout << "Favorite color: " << favorite << endl;

return 0;

}

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

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

Also called **type promotion** or **type coercion**.

* Done **automatically by the compiler**.
* Converts smaller data types to larger ones to avoid data loss (**type promotion**).
* No special syntax needed.

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int num = 10;

double result = num + 3.5; // int is automatically converted to double

cout << result; // Output: 13.5

return 0;

}

Here, num (int) is **implicitly converted** to double before addition.

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

* Done **manually by the programmer**.
* Uses casting operators or constructor syntax.
* You decide exactly how the conversion happens.

**Syntax Options:**

cpp

(type) value // C-style cast

type(value) // Function-style cast

static\_cast<type>(value) // C++ preferred way

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

double pi = 3.1416;

int intPi = (int) pi; // C-style

// int intPi = static\_cast<int>(pi); // C++ style

cout << "Original: " << pi << endl; // 3.1416

cout << "Converted: " << intPi << endl; // 3

return 0;

}

Here, pi is **explicitly converted** to an integer (fractional part lost).

## **Key Differences Table**

| **Feature** | **Implicit Conversion** | **Explicit Conversion** |
| --- | --- | --- |
| **Who does it?** | Compiler | Programmer |
| **Syntax** | No special syntax | (type)value or static\_cast<type>(value) |
| **Control** | No control over conversion | Full control over conversion |
| **Risk of Data Loss** | Less obvious, can happen silently | Intentional, programmer is aware |
| **Example** | int x = 5; double y = x; | double y = 5.6; int x = (int) y; |

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

## **1. Arithmetic Operators**

Used for mathematical calculations.

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

**Example:**

cpp

int a = 10, b = 3;

cout << a + b; // 13

cout << a % b; // 1

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

Used to compare values; returns true (1) or false (0).

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

## **3. Logical Operators**

Used for logical operations.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| && | Logical AND | (a > 5 && b < 5) |
| ` |  | ` |
| ! | Logical NOT | !(a > b) |

## **4. Assignment Operators**

Used to assign values to variables.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| = | Assign | a = 5 |
| += | Add and assign | a += 3 (same as a = a + 3) |
| -= | Subtract and assign | a -= 2 |
| \*= | Multiply and assign | a \*= 2 |
| /= | Divide and assign | a /= 2 |
| %= | Modulus and assign | a %= 2 |

## **5. Increment/Decrement Operators**

Increase or decrease a variable by 1.

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

**Note:**

* **Pre-increment (++a)**: Increments first, then uses value.
* **Post-increment (a++)**: Uses value first, then increments.

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

Shorthand for if-else.

**Syntax:**

cpp

(condition) ? expression1 : expression2;

**Example:**

cpp

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

## **7. Bitwise Operators**

Operate on data at the **bit level**.

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

## **8. Special Operators**

* **sizeof** – Returns size of data type/variable.

cpp

cout << sizeof(int);

* **typeid** – Returns type information (requires <typeinfo>).

cpp

cout << typeid(a).name();

* **Scope Resolution (::)** – Access global variables or class members.
* **Member Access (. and ->)** – Access object members.

## **Example Program Using All Types**

cpp

#include <iostream>

using namespace std;

int main() {

int a = 10, b = 3;

// Arithmetic

cout << "Add: " << a + b << endl;

// Relational

cout << (a > b) << endl;

// Logical

cout << (a > 5 && b < 5) << endl;

// Assignment

a += 2;

cout << "a after += 2: " << a << endl;

// Increment

cout << "Pre-increment: " << ++b << endl;

// Ternary

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

cout << "Max: " << max << endl;

// Bitwise

cout << "a & b: " << (a & b) << endl;

// Special

cout << "Size of int: " << sizeof(int) << endl;

return 0;

}

**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** once it is defined.

* Used when a fixed value is required throughout the program.
* Makes code **more readable**, **less error-prone**, and easier to maintain.

### ****Ways to Define Constants:****

#### (a) Using const keyword

cpp

const double PI = 3.1416;

* Must be initialized when declared.
* Value cannot be changed later.

#### (b) Using #define preprocessor macro

cpp

#define MAX 100

* No data type; simple text replacement before compilation.
* Cannot be scoped inside functions (applies globally).

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

const int AGE = 18;

#define LIMIT 100

cout << "Age: " << AGE << endl;

cout << "Limit: " << LIMIT << endl;

// AGE = 20; // ❌ Error: Cannot modify a constant

return 0;

}

## **2. Literals in C++**

A **literal** is a fixed value written directly in the code (hard-coded constant).

### ****Types of Literals:****

1. **Integer Literals**
   * Example: 10, -25, 0
   * Can be decimal, octal (0 prefix), or hexadecimal (0x prefix).

Cpp

int a = 10; // decimal

int b = 012; // octal (value 10 in decimal)

int c = 0xA; // hexadecimal (value 10 in decimal)

1. **Floating-point Literals**
   * Example: 3.14, -0.5, 2.5e3 (scientific notation)

cpp

double pi = 3.1416;

float expNum = 2.5e3; // 2500

1. **Character Literals**
   * Example: 'A', '1', '\n'
   * Enclosed in **single quotes**.

cpp

char grade = 'A';

1. **String Literals**
   * Example: "Hello", "C++ is fun"
   * Enclosed in **double quotes**.

cpp

string msg = "Hello, World!";

1. **Boolean Literals**
   * true (1) and false (0)

cpp

bool isOn = true;

## **Difference Between Constants and Literals**

| **Aspect** | **Constants** | **Literals** |
| --- | --- | --- |
| **Definition** | Named storage location with a fixed value | Actual fixed value written in the code |
| **Example** | const double PI = 3.14; | 3.14 |
| **Changeability** | Cannot change after initialization | Always fixed |
| **Purpose** | Improve readability and maintainability | Provide actual values directly |

✅ **Quick Tip for Exams:**

* **Constant** = A fixed value with a name.
* **Literal** = The actual value itself.

**3.Control Flow Statements**

## **Control Flow Statements in C++**

Control flow statements decide **the order in which instructions are executed** in a program.

They are divided into **three main categories**:

## **1. Decision-Making Statements (Selection)**

Used to **choose** different actions based on conditions.

### (a) if ****statement****

Executes a block only if a condition is true.

cpp

if (age >= 18) {

cout << "You are an adult.";

}

### (b) if-else ****statement****

Chooses between two options.

cpp

if (marks >= 40)

cout << "Pass";

else

cout << "Fail";

### (c) if-else if ****ladder****

Checks multiple conditions in sequence.

cpp

if (marks >= 75)

cout << "Distinction";

else if (marks >= 60)

cout << "First Class";

else

cout << "Fail";

### (d) switch ****statement****

Chooses one case from many possible options.

cpp

switch (day) {

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

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

default: cout << "Invalid";

}

## **2. Looping Statements (Iteration)**

Used to **repeat** a block of code multiple times.

### (a) for ****loop****

Runs a block a fixed number of times.

cpp

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

cout << i << " ";

}

### (b) while ****loop****

Runs while a condition remains true.

cpp

int i = 1;

while (i <= 5) {

cout << i << " ";

i++;

}

### (c) do-while ****loop****

Runs **at least once**, then repeats while condition is true.

cpp

int i = 1;

do {

cout << i << " ";

i++;

} while (i <= 5);

## **3. Jump Statements**

Used to **alter the normal flow** of execution.

| **Statement** | **Purpose** |
| --- | --- |
| break | Exit a loop or switch immediately |
| continue | Skip the rest of the loop and go to the next iteration |
| goto | Jump to a labeled statement (generally discouraged) |
| return | Exit from a function and optionally return a value |

**Example:**

cpp

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

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

if (i == 5) break; // stop when i=5

cout << i << " ";

}

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

## **1. What Are Conditional Statements?**

Conditional statements in C++ are used to **make decisions** in a program.  
They allow the program to execute **different code blocks** based on whether a **condition** evaluates to **true** or **false**.

**Main conditional statements in C++:**

* if
* if-else
* if-else if
* switch

## **2.** if-else **Statement**

The if-else statement executes **one block of code if a condition is true**, and **another block if the condition is false**.

**Syntax:**

cpp

if (condition) {

// Code if condition is true

} else {

// Code if condition is false

}

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int marks;

cout << "Enter marks: ";

cin >> marks;

if (marks >= 40) {

cout << "Pass";

} else {

cout << "Fail";

}

return 0;

}

**How it works:**

* If marks >= 40 is true → prints "Pass".
* Else → prints "Fail".

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

The switch statement allows you to choose **one case to execute** from many possible options based on the value of an expression.

**Syntax:**

cpp

switch (expression) {

case value1:

// Code for case 1

break;

case value2:

// Code for case 2

break;

...

default:

// Code if no case matches

}

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int day;

cout << "Enter day number (1-3): ";

cin >> day;

switch (day) {

case 1:

cout << "Monday";

break;

case 2:

cout << "Tuesday";

break;

case 3:

cout << "Wednesday";

break;

default:

cout << "Invalid day";

}

return 0;

| **Feature** | **if-else** | **switch** |
| --- | --- | --- |
| **Condition Type** | Can test any expression (numbers, strings, ranges) | Tests only discrete values (usually integers, enums, characters) |
| **Multiple Conditions** | Can use logical operators (&&, ` |  |
| **Readability** | Better for complex conditions | Better for menu-style choices |

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

## **1.** for **Loop**

* Used when you **know** in advance **how many times** you want to repeat a task.
* Initialization, condition check, and update are in **one line**.

**Syntax:**

cpp

for (initialization; condition; update) {

// Code to repeat

}

**Example:**

cpp

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

cout << i << " ";

}

## **2.** while **Loop**

* Used when you **do not know in advance** how many times the loop will run.
* Condition is **checked before** the loop body runs.

**Syntax:**

cpp

while (condition) {

// Code to repeat

}

**Example:**

cpp

int i = 1;

while (i <= 5) {

cout << i << " ";

i++;

}

## **3.** do-while **Loop**

* Similar to while loop, but **always runs at least once** because the condition is **checked after** the loop body.

**Syntax:**

cpp

do {

// Code to repeat

} while (condition);

**Example:**

cpp

int i = 1;

do {

cout << i << " ";

i++;

} while (i <= 5);

## **4. Key Differences Table**

| **Feature** | **for loop** | **while loop** | **do-while loop** |
| --- | --- | --- | --- |
| **When to use** | When the number of iterations is known | When the number of iterations is unknown | When you need to run at least once |
| **Condition Check** | Before each iteration | Before each iteration | After each iteration |
| **Execution Guarantee** | May not execute if condition is false initially | May not execute if condition is false initially | Executes at least once |
| **Syntax** | Compact (init, condition, update in one line) | Init and update outside loop | Init outside, update inside, condition after loop |
| **Common Use** | Counting loops | Reading data until a condition | Menu-driven programs |

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

## **1.** break **Statement**

* **Purpose:** Immediately **exits** the loop, regardless of the loop condition.
* Commonly used to **stop a loop early** when a certain condition is met.

**Syntax:**

cpp

break;

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

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

if (i == 5) {

break; // exits loop when i is 5

}

cout << i << " ";

}

return 0;

}

**Output:**

1 2 3 4

Here, the loop stops completely when i == 5.

## **2.** continue **Statement**

* **Purpose:** Skips the **current iteration** and moves to the **next iteration** of the loop.
* Does **not** terminate the loop entirely.

**Syntax:**

cpp

continue;

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

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

if (i == 3) {

continue; // skips printing when i is 3

}

cout << i << " ";

}

return 0;

}

**Output:**

1 2 4 5

Here, the loop skips printing 3 but continues with the rest.

## **3. Key Differences**

| **Feature** | **break** | **continue** |
| --- | --- | --- |
| **Effect** | Exits loop completely | Skips current iteration |
| **Loop Behavior** | No further iterations after break | Moves to the next iteration |
| **Common Use** | Stop searching once found | Skip unwanted cases but keep looping |
|  |  |  |

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

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

**Definition:**  
A **nested control structure** means placing one control structure **inside another**.  
This can be:

* An if inside another if (**nested if**)
* A loop inside another loop (**nested loops**)
* A loop inside an if, or vice versa

They are used when a decision or repetition depends on **multiple levels of conditions**.

### ****Example 1: Nested**** if ****Statement****

cpp

#include <iostream>

using namespace std;

int main() {

int age = 20;

char citizen = 'Y';

if (age >= 18) { // Outer if

if (citizen == 'Y') { // Inner if

cout << "Eligible to vote";

} else {

cout << "Not a citizen, cannot vote";

}

} else {

cout << "Too young to vote";

}

return 0;

}

**Explanation:**

* First, we check if the person is **18 or older**.
* If **yes**, we then check **citizenship**.
* This requires one decision **inside** another.

### ****Example 2: Nested Loop****

cpp

#include <iostream>

using namespace std;

int main() {

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

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

cout << "i = " << i << ", j = " << j << endl;

}

}

return 0;

}

**Output:**

ini

i = 1, j = 1

i = 1, j = 2

i = 2, j = 1

i = 2, j = 2

i = 3, j = 1

i = 3, j = 2

**Explanation:**

* For each value of i, the **inner loop** runs completely.
* This is common in **tables, patterns, and matrix operations**.

✅ **Key Points:**

* Helps handle **complex decision-making** or **repeated tasks in multiple dimensions**.
* Care must be taken to avoid **too many levels** (can make code hard to read).

**4.**  **Functions and Scope**

## **Functions in C++**

A **function** is a block of code designed to perform a specific task.  
It helps in **code reusability**, **modularity**, and **better readability**.

### ****Types of Functions****

1. **Library (Predefined) Functions**
   * Provided by C++ libraries.
   * Example: sqrt(), pow(), strlen().
2. **User-Defined Functions**
   * Created by the programmer.
   * Example:

cpp

#include <iostream>

using namespace std;

// Function definition

int add(int a, int b) {

return a + b;

}

int main() {

cout << "Sum = " << add(5, 3);

return 0;

}

### ****Structure of a Function****

cpp

returnType functionName(parameters) {

// body of the function

return value; // if needed

}

### ****Function Categories****

* **With arguments and return value**
* **With arguments and no return value**
* **No arguments and return value**
* **No arguments and no return value**

## **Scope in C++**

**Scope** determines **where a variable can be accessed** in the program.

### ****Types of Scope****

1. **Local Scope**
   * Variable declared **inside a function/block**.
   * Accessible only within that block.

cpp

void test() {

int x = 10; // local

cout << x;

}

1. **Global Scope**
   * Variable declared **outside all functions**.
   * Accessible from any function in the file.

cpp

int g = 100; // global

void show() {

cout << g;

}

1. **Function Scope**
   * Labels inside functions (used with goto) have scope within that function.
2. **Block Scope**
   * Variables inside { } are accessible only within that block.

### ****Example Demonstrating Scope****

cpp

#include <iostream>

using namespace std;

int globalVar = 50; // Global scope

void show() {

int localVar = 10; // Local scope

cout << "Local: " << localVar << endl;

cout << "Global: " << globalVar << endl;

}

int main() {

show();

cout << "Global again: " << globalVar << endl;

return 0;

}

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

## **What is a Function in C++?**

A **function** is a block of code that performs a specific task.  
It helps to:

* Avoid code repetition (**reusability**)
* Make programs **modular** and easier to debug
* Improve readability

## **Three Main Parts of a Function**

A function in C++ has three main stages:

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

* Tells the compiler **the name, return type, and parameters** of the function before it is used.
* Ends with a semicolon.
* **Syntax:**

cpp

returnType functionName(parameterList);

* **Example:**

cpp

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

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

* Contains the **actual code** (body) of the function.
* **Syntax:**

cpp

returnType functionName(parameterList) {

// statements

return value;

}

* **Example:**

cpp

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

return a + b;

}

### ****3. Function Calling****

* Executes the function by using its name followed by parentheses.
* Must match the **number and type of parameters** in the declaration.
* **Example:**

cpp

int sum = add(5, 3); // function call

## **Complete Example**

cpp

#include <iostream>

using namespace std;

// 1. Function Declaration

int add(int, int);

int main() {

// 3. Function Calling

int result = add(10, 20);

cout << "Sum = " << result;

return 0;

}

// 2. Function Definition

int add(int a, int b) {

return a + b;

}

### ****Output:****

ini

Sum = 30

✅ **Key Points to Remember**

* **Declaration**: Informs compiler about function (optional if defined before main()).
* **Definition**: Contains the actual code.
* **Calling**: Executes the function.
* Function can have **parameters** (input) and **return values** (output).

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

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

The **scope** of a variable refers to the **region of the program** where that variable can be accessed or used.

### ****Types of Scope****

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

* Variable declared **inside a function or block { }**.
* Accessible **only within that block**.
* Created when the block starts execution and destroyed when it ends.

**Example:**

cpp

#include <iostream>

using namespace std;

void show() {

int x = 10; // Local variable

cout << "Local variable x = " << x << endl;

}

int main() {

show();

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

return 0;

}

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

* Variable declared **outside all functions**.
* Accessible from **any function** in the program.
* Created when the program starts and destroyed when it ends.

**Example:**

cpp

#include <iostream>

using namespace std;

int g = 100; // Global variable

void show() {

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

}

int main() {

show();

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

return 0;

}

## **Difference between Local and Global Scope**

| **Aspect** | **Local Scope** | **Global Scope** |
| --- | --- | --- |
| **Declaration** | Inside a function or block { } | Outside all functions |
| **Accessibility** | Only within the block where declared | Accessible from any function |
| **Lifetime** | Created when block starts, destroyed when it ends | Exists for the entire program execution |
| **Memory** | Stored in **stack** | Stored in **global/static memory** |
| **Priority** | Local variable with same name overrides global variable inside the block | Used if no local variable with same name exists |

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

## **What is Recursion in C++?**

**Recursion** is the process in which a **function calls itself** directly or indirectly to solve a problem.

A recursive function has **two main parts**:

1. **Base Case** → The condition that stops the recursion (prevents infinite loop).
2. **Recursive Case** → The function calls itself with a smaller/simpler input.

### ****General Syntax****

cpp

returnType functionName(parameters) {

if (base\_condition) {

// stop recursion

return value;

}

else {

// recursive call

return functionName(modified\_parameters);

}

}

## **Example: Factorial using Recursion**

cpp

#include <iostream>

using namespace std;

// Recursive function to calculate factorial

int factorial(int n) {

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

return 1;

else // Recursive case

return n \* factorial(n - 1);

}

int main() {

int num = 5;

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

return 0;

}

### ****Output****

mathematica

Factorial of 5 = 120

### ****How it Works (for factorial(5))****

1. factorial(5) → 5 × factorial(4)
2. factorial(4) → 4 × factorial(3)
3. factorial(3) → 3 × factorial(2)
4. factorial(2) → 2 × factorial(1)
5. factorial(1) → 1 (base case reached)

Then the results are multiplied **backwards**

5 × 4 × 3 × 2 × 1 = 120

✅ **Advantages**

* Simplifies code for problems like factorial, Fibonacci, tree traversal, etc.
* Reduces the need for complex loops.

⚠ **Disadvantages**

* Uses more memory (function call stack).
* Can be slower than iterative solutions.

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

## **Function Prototypes in C++**

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

* **Function name**
* **Return type**
* **Parameters** (number, type, and order)

It **does not** contain the actual function body.  
It **ends with a semicolon (;)**.

### ****Syntax****

cpp

returnType functionName(parameterType1, parameterType2, ...);

### ****Example****

cpp

#include <iostream>

using namespace std;

// Function prototype

int add(int, int);

int main() {

int result = add(5, 3); // Function call

cout << "Sum = " << result;

return 0;

}

// Function definition

int add(int a, int b) {

return a + b;

}

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

1. **Allows calling a function before its definition**
   * The compiler knows about the function’s existence before encountering it in code.
2. **Helps the compiler check for correct arguments**
   * Number and type of arguments are checked during compilation.
3. **Improves code organization**
   * Function definitions can be placed after main() without errors.

### ****Without Prototype (Possible Error)****

cpp

#include <iostream>

using namespace std;

int main() {

cout << add(5, 3); // ❌ Compiler may not know about add()

return 0;

}

int add(int a, int b) {

return a + b;

}

**Solution:** Add a **prototype** before main().

💡 **Key Point:** Function prototypes are especially useful when:

* Functions are defined **after** main().
* Functions are declared in **header files** for use in multiple source files.

**5.Arrays and Strings**

## **1. Arrays in C++**

An **array** is a collection of elements **of the same data type**, stored in **contiguous memory locations**, accessed using **index numbers** (starting from 0).

### ****Syntax****

cpp

dataType arrayName[size];

### ****Example****

cpp

#include <iostream>

using namespace std;

int main() {

int marks[5] = {90, 85, 88, 92, 80}; // Array initialization

// Accessing array elements

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

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

}

return 0;

}

### ****Types of Arrays****

1. **One-dimensional array** → Linear list of elements.
2. **Two-dimensional array** → Table/matrix form.
3. **Multi-dimensional array** → Array of arrays.

### ****Example (2D Array)****

cpp

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

## **2. Strings in C++**

A **string** is a sequence of characters **ending with a null character ('\0')**.

### ****Types of Strings****

1. **C-Style Strings** (Character arrays)

cpp

char name[10] = "Hello";

* + Must be terminated with '\0'.
  + Functions: strcpy(), strlen(), strcmp() from <cstring>.

1. **C++ String Class** (std::string)

cpp

#include <string>

string name = "Hello";

* + Easier to use, supports operators like +, ==.
  + Functions: .length(), .substr(), .append().

### ****Example (String Class)****

cpp

#include <iostream>

#include <string>

using namespace std;

int main() {

string first = "Hello";

string second = "World";

string full = first + " " + second; // Concatenation

cout << full;

return 0;

}

### ****Array vs. String****

| **Feature** | **Array** | **String** |
| --- | --- | --- |
| Data type | Can hold any data type | Holds only characters |
| Null terminator | Not required | Required in C-style strings |
| Size change | Fixed after declaration | Dynamic in std::string |

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

## **Arrays in C++**

An **array** is a collection of **elements of the same data type** stored in **contiguous memory locations** and accessed using **index numbers** starting from **0**.

### ****Declaration Syntax****

cpp

dataType arrayName[size];

### ****Example****

cpp

int marks[5] = {90, 85, 88, 92, 80};

## **Types of Arrays**

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

* Stores data in a **linear form**.
* Accessed using **one index**: arrayName[index].
* Index starts from 0 to size - 1.

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

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

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

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

}

return 0;

}

**Memory layout:**  
[1][2][3][4][5]

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

* Stores data in **table-like (2D)** or **higher dimensional** form.
* Accessed using **multiple indices**.
* Common form: **2D array** → rows × columns.

**Syntax:**

cpp

dataType arrayName[rows][columns];

**Example (2D Array):**

cpp

#include <iostream>

using namespace std;

int main() {

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;

}

return 0;

}

**Memory layout:**  
Row 0 → [1][2][3]  
Row 1 → [4][5][6]

### ****Key Differences****

| **Feature** | **Single-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| Structure | Linear | Table-like (rows & columns) |
| Index count | One | Two or more |
| Example syntax | int a[5]; | int a[3][4]; |
| Storage | Sequential elements in 1 line | Stored row by row in memory |
| Usage | Simple lists | Matrices, grids, tables |

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

## **1. What is a String in C++?**

A **string** is a sequence of characters stored in memory.  
In C++, there are **two main ways** to handle strings:

1. **C-Style Strings** → Character arrays ending with a null character '\0'.
2. **C++ String Class (std::string)** → Provided by the **Standard Template Library (STL)**.

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

### ****Declaration & Initialization****

cpp

char name[10] = "Hello"; // '\0' is automatically added

### ****Common Functions**** (in <cstring>):

* strlen(str) → Returns length of the string (without \0).
* strcpy(dest, src) → Copies string.
* strcat(str1, str2) → Concatenates strings.
* strcmp(str1, str2) → Compares strings.

**Example:**

cpp

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

strcat(str1, str2); // Concatenate

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

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

return 0;

}

## **3. C++** std::string **Class**

### ****Advantages****

* Easier to use.
* Automatically manages memory.
* Supports operators (+, ==, etc.).

### ****Declaration & Initialization****

cpp

#include <string>

string name = "Hello";

### ****Common Member Functions****

* .length() → Returns length.
* .append(str) → Adds at the end.
* .substr(pos, len) → Extracts substring.
* .find(str) → Finds position of substring.
* .erase(pos, len) → Removes part of the string.

**Example:**

cpp

#include <iostream>

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = "World";

string s3 = s1 + " " + s2; // Concatenation

cout << "Concatenated: " << s3 << endl;

cout << "Length: " << s3.length() << endl;

s3.append("!!!");

cout << "After append: " << s3 << endl;

cout << "Substring (0,5): " << s3.substr(0, 5) << endl;

return 0;

}

## **4. Difference Between C-Style Strings and** std::string

| **Feature** | **C-Style String** | **std::string Class** |
| --- | --- | --- |
| Storage | Fixed-size character array | Dynamic size |
| Null terminator | Required ('\0') | Not needed |
| Operations | Using <cstring> functions | Built-in member functions |
| Memory handling | Manual | Automatic |

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

## **1. Initializing One-Dimensional (1D) Arrays**

A **1D array** is a list of elements of the same type stored in contiguous memory.

### ****Syntax****

cpp

datatype arrayName[size] = {value1, value2, ...};

### ****Examples****

cpp

#include <iostream>

using namespace std;

int main() {

// Method 1: Explicit size & values

int arr1[5] = {10, 20, 30, 40, 50};

// Method 2: Size inferred from values

int arr2[] = {1, 2, 3, 4};

// Method 3: Partial initialization (rest set to 0)

int arr3[5] = {7, 8};

// Display

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

cout << arr1[i] << " ";

}

cout << endl;

return 0;

}

## **2. Initializing Two-Dimensional (2D) Arrays**

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

### ****Syntax****

cpp

datatype arrayName[rows][columns] = {

{value1, value2, ...},

{value3, value4, ...}

};

### ****Examples****

cpp

#include <iostream>

using namespace std;

int main() {

// Method 1: Fully specified initialization

int matrix1[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

// Method 2: Without specifying rows

int matrix2[][3] = {

{7, 8, 9},

{10, 11, 12}

};

// Method 3: Partial initialization (missing values set to 0)

int matrix3[2][3] = {

{1},

{4, 5}

};

// Display

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

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

cout << matrix1[i][j] << " ";

}

cout << endl;

}

return 0;

}

✅ **Key Points to Remember**

* If fewer values are provided, the rest are **automatically initialized to 0**.
* For 2D arrays, **nested braces {}** help match rows and columns.
* You can **omit the size** of the first dimension in initialization if you provide all values.

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

## **1. Strings in C++**

C++ supports two main ways to handle strings:

1. **C-style strings** → arrays of char ending with a null character ('\0')
2. **std::string class** → part of the **Standard Template Library (STL)**, easier and safer to use

## **2. Common String Operations**

### ****A) Using C-Style Strings (****char[]****)****

To use string functions, you need:

cpp

#include <cstring> // for string handling functions

#### Common Functions:

| **Function** | **Purpose** | **Example** |
| --- | --- | --- |
| strlen(s) | Returns length of string | strlen("Hello") → 5 |
| strcpy(dest, src) | Copies string | strcpy(name, "John"); |
| strcat(s1, s2) | Concatenates two strings | strcat(first, last); |
| strcmp(s1, s2) | Compares strings (0 if equal) | strcmp("A", "B"); |
| strrev(s)\* | Reverses string (non-standard) | — |

**Example:**

cpp

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

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

strcat(str1, str2);

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

strcpy(str1, "Hi");

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

return 0;

}

### ****B) Using**** std::string ****(Recommended)****

To use:

cpp

#include <string>

#### Common Member Functions:

| **Function** | **Purpose** | **Example** |
| --- | --- | --- |
| length() / size() | Returns length | s.length() |
| append(str) | Adds to end | s.append(" World") |
| + operator | Concatenate | s1 + s2 |
| substr(pos, len) | Extract substring | s.substr(0, 5) |
| find(str) | Find position of substring | s.find("Hello") |
| compare(str) | Compare strings | s1.compare(s2) |

**Example:**

cpp

#include <iostream>

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = "World";

cout << "Length of s1: " << s1.length() << endl;

string s3 = s1 + " " + s2;

cout << "Concatenated: " << s3 << endl;

cout << "Substring (0-5): " << s3.substr(0, 5) << endl;

cout << "Position of 'World': " << s3.find("World") << endl;

return 0;

}

## **3. Key Differences**

| **Feature** | **C-Style Strings** | **std::string** |
| --- | --- | --- |
| Header | <cstring> | <string> |
| Safety | Risk of overflow | Auto-managed memory |
| Ease | More complex | Easier to use |
| Operators | Not overloaded | Supports +, = etc. |

1. **Introduction to Object-Oriented Programming**

## **Introduction to Object-Oriented Programming (OOP)**

### ****Definition****

Object-Oriented Programming (OOP) is a programming paradigm that organizes a program into **objects**, which are instances of **classes**.  
Each object combines **data (attributes)** and **functions (methods)** into a single unit.

## **Main Features of OOP**

1. **Encapsulation** – Bundling data and methods together and restricting direct access.
2. **Abstraction** – Hiding unnecessary implementation details and showing only the essential features.
3. **Inheritance** – Creating new classes from existing ones to reuse code.
4. **Polymorphism** – Ability to use the same function or operator in different ways.

## **Advantages of OOP over Procedural Programming**

* **Modularity** – Code is divided into classes and objects for better organization.
* **Reusability** – Classes can be reused in multiple programs.
* **Scalability & Maintainability** – Easier to modify and extend programs.
* **Security** – Encapsulation hides sensitive data from outside interference.

## **Basic OOP Terms**

| **Term** | **Meaning** |
| --- | --- |
| **Class** | Blueprint or template for creating objects. |
| **Object** | Instance of a class containing actual data. |
| **Method** | Function defined inside a class that operates on its data. |
| **Attribute** | Data members (variables) inside a class. |

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

## **Key Concepts of OOP**

### ****1. Class****

* **Definition**: A blueprint or template for creating objects.
* **Contains**: Data members (**attributes**) and member functions (**methods**).
* **Example**:

cpp

CopyEdit

class Car {

public:

string brand;

int year;

void display() {

cout << brand << " - " << year << endl;

}

};

### ****2. Object****

* **Definition**: An instance of a class that has its own **copy** of data members.
* **Example**:

cpp

CopyEdit

Car car1; // object of class Car

car1.brand = "Toyota";

car1.year = 2022;

### ****3. Encapsulation****

* **Meaning**: Wrapping **data** and **methods** into one unit (class) and restricting access to the data.
* **Achieved By**: Using access specifiers (private, protected, public).
* **Example**:

cpp

CopyEdit

class BankAccount {

private:

double balance;

public:

void deposit(double amount) { balance += amount; }

double getBalance() { return balance; }

};

### ****4. Abstraction****

* **Meaning**: Showing **only necessary details** to the user while hiding the background implementation.
* **Example**: You call car.startEngine() without knowing the complex mechanism inside.
* **Achieved By**: Using **abstract classes** and **interfaces** (pure virtual functions in C++).

### ****5. Inheritance****

* **Meaning**: The process of creating a new class (**derived class**) from an existing class (**base class**).
* **Types**:
  + Single
  + Multiple
  + Multilevel
  + Hierarchical
  + Hybrid
* **Example**:

cpp

class Vehicle {

public:

string brand;

};

class Car : public Vehicle {

public:

int year;

};

### ****6. Polymorphism****

* **Meaning**: The ability of the same function name to perform different tasks.
* **Types**:
  1. **Compile-time (Overloading)** – Function overloading, Operator overloading.
  2. **Run-time (Overriding)** – Achieved using virtual functions.
* **Example**:

cpp

class Shape {

public:

virtual void draw() { cout << "Drawing Shape\n"; }

};

class Circle : public Shape {

public:

void draw() override { cout << "Drawing Circle\n"; }

};

✅ **Summary Table**

| **Concept** | **Purpose** |
| --- | --- |
| **Class** | Blueprint for objects |
| **Object** | Instance of a class |
| **Encapsulation** | Data hiding and bundling |
| **Abstraction** | Hiding complexity |
| **Inheritance** | Code reusability |
| **Polymorphism** | Same interface, different behavior |

**What are classes and objectsin C++? Provide an example**

### ****1. Class****

* **Definition**: A class is a **blueprint** or **template** that defines the data members (variables) and member functions (methods) of an object.
* **Purpose**: Organizes data and functions into one unit.
* **Syntax**:

cpp

class ClassName {

// Access specifiers: public, private, protected

data\_members;

member\_functions;

};

### ****2. Object****

* **Definition**: An **instance** of a class.  
  When you create an object, it gets its own **copy of data members** and can use the member functions of the class.
* **Purpose**: Represents a real-world entity in the program.

### ****Example****

cpp

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

string brand;

int year;

void displayInfo() {

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

}

};

int main() {

// Creating objects of the Car class

Car car1;

car1.brand = "Toyota";

car1.year = 2022;

car1.displayInfo();

Car car2;

car2.brand = "Honda";

car2.year = 2023;

car2.displayInfo();

return 0;

}

**Output:**

yaml

Brand: Toyota, Year: 2022

Brand: Honda, Year: 2023

✅ **Key Points**

* **Class** = Design / Blueprint
* **Object** = Actual implementation of that design
* You can create multiple **objects** from one **class**.

**What isinheritance in C++? Explain with an example.**

## **Inheritance in C++**

### ****Definition****

Inheritance is an **OOP concept** in which a new class (**derived class**) is created from an existing class (**base class**).  
The **derived class** inherits the data members and member functions of the **base class**, which promotes **code reusability** and reduces redundancy.

### ****Key Benefits****

* **Code Reusability** – Use existing class functionality without rewriting code.
* **Extensibility** – Add new features to existing classes.
* **Maintainability** – Changes in base class automatically apply to derived classes.

### ****Types of Inheritance in C++****

1. **Single Inheritance** – One base → One derived class
2. **Multiple Inheritance** – Multiple base → One derived class
3. **Multilevel Inheritance** – Derived class becomes base for another
4. **Hierarchical Inheritance** – One base → Multiple derived classes
5. **Hybrid Inheritance** – Combination of above types

### ****Example – Single Inheritance****

cpp

#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 {

public:

int year;

void displayInfo() {

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

}

};

int main() {

Car myCar;

myCar.brand = "Toyota";

myCar.year = 2022;

myCar.start(); // Function from base class

myCar.displayInfo(); // Function from derived class

return 0;

}

**Output:**

yam

Toyota is starting...

Brand: Toyota, Year: 2022

✅ **Key Points to Remember**

* Use public inheritance to keep base class public members **public** in derived class.
* Private members of the base class are **not directly accessible** in the derived class.
* A derived class can add new members and also override existing methods.

**What is encapsulation in C++? How isit achieved in classes?**

## **Encapsulation in C++**

### ****Definition****

Encapsulation is the **OOP concept** of **binding data and functions** that operate on that data **into a single unit** (a class) and **restricting direct access** to the data from outside the class.  
It ensures **data hiding** and **controlled access**.

### ****Why Use Encapsulation?****

* **Security** – Prevents accidental modification of data.
* **Control** – You can control how data is accessed or modified.
* **Maintainability** – Easier to change code without affecting other parts.

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

1. **Use Access Specifiers**:
   * **private** – Data hidden from outside (default for class members).
   * **public** – Accessible from outside.
   * **protected** – Accessible in derived classes.
2. **Provide public functions (getters/setters)** to **read/write private data**.

### ****Example****

cpp

#include <iostream>

using namespace std;

class Student {

private:

string name; // private data member

int age; // private data member

public:

// Setter for name

void setName(string n) {

name = n;

}

// Getter for name

string getName() {

return name;

}

// Setter for age

void setAge(int a) {

if (a > 0) // validation before setting

age = a;

}

// Getter for age

int getAge() {

return age;

}

};

int main() {

Student s1;

// Set data using setters

s1.setName("John");

s1.setAge(20);

// Get data using getters

cout << "Name: " << s1.getName() << endl;

cout << "Age: " << s1.getAge() << endl;

return 0;

}

**Output:**

makefile

Name: John

Age: 20

✅ **Key Points to Remember**

* Encapsulation = **Data Hiding + Controlled Access**.
* Achieved using **private/protected members** + **public getters/setters**.
* Prevents unauthorized access and ensures data integrity.