**Module #3 Introduction to OOPS Programming**

1. Introduction to C++

LAB EXERCISES:

1. First C++ Program: Hello World o Write a simple C++ program to display "Hello, World!". o Objective: Understand the basic structure of a C++ program, including #include, main(), and cout

Ans. #include<iostream>

Using namespace std;

Int main(){

Cout<<”Hello World: “<<end1;

}

Basic Input/Output

o Write a C++ program that accepts user input for their name and age and then displays a personalized greeting. o Objective: Practice input/output operations using cin and cout

Ans. #include<iostream>

#include<string>

using namespace std;

int main(){

string name;

int age;

cout<< "enter the name: "<<end1;

cin>>name;

cout<< "enter the age: "<<end1;

cin>>age;

cout<<name<<end1<<age;

return 0;

}

POP vs. OOP Comparison Program

Write two small programs: one using Procedural Programming (POP) to calculate the area of a rectangle, and another using Object-Oriented Programming (OOP) with a class and object for the same task. o Objective: Highlight the difference between POP and OOP approaches.

Ans. Here are two small C++ programs:

1. **Procedural Programming (POP) Approach** – Uses functions to calculate the area of a rectangle.
2. **Object-Oriented Programming (OOP) Approach** – Uses a class and object to achieve the same task.
3. **1. Procedural Programming (POP) Approach**

#include <iostream>

using namespace std;

// Function to calculate the area of a rectangle

double calculateArea(double length, double width) {

return length \* width;

}

int main() {

double length, width;

cout << "Enter length: ";

cin >> length;

cout << "Enter width: ";

cin >> width;

double area = calculateArea(length, width);

cout << "Area of the rectangle: " << area << endl;

}

**2. Object-Oriented Programming (OOP) Approach**

#include <iostream>

using namespace std;

// Class representing a Rectangle

class Rectangle {

private:

double length, width;

public:

// Constructor to initialize the rectangle

Rectangle(double l, double w) {

length = l;

width = w;

}

double getArea() {

return length \* width;

}

};

int main() {

double length, width;

cout << "Enter length: ";

cin >> length;

cout << "Enter width: ";

cin >> width;

Rectangle rect(length, width);

cout << "Area of the rectangle: " << rect.getArea() << endl;

}

1. Setting Up Development Environment o Write a program that asks for two numbers and displays their sum. Ensure this is done after setting up the IDE (like Dev C++ or CodeBlocks). o Objective: Help students understand how to install, configure, and run programs inan IDE.

Ans. Before writing the program, let’s go through the steps to **set up an IDE** (like **Dev-C++** or **CodeBlocks**) for C++ development.

#include <iostream>

using namespace std;

int main() {

int num1, num2, sum;

cout << "Enter first number: ";

cin >> num1;

cout << "Enter second number: ";

cin >> num2;

sum = num1 + num2;

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

return 0;

}

THEORY EXERCISE:

1. What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP

Ans. The key differences between Procedural programming and object programming (OOP) are as follows :

|  |  |  |  |
| --- | --- | --- | --- |
| | **Feature** | **Procedural Programming (POP)** | **Object-Oriented Programming (OOP)** | | --- | --- | --- | |
| |  |  |  | | --- | --- | --- | | **Approach** | Follows a step-by-step procedure using functions. | Organizes code using objects and classes. | |
| |  |  |  | | --- | --- | --- | | **Structure** | Uses functions to operate on data. | Encapsulates data and behavior within objects. | |
| |  |  |  | | --- | --- | --- | | **Data Handling** | Data is passed explicitly between functions. | Data is encapsulated inside objects. | |
| |  |  |  | | --- | --- | --- | | **Encapsulation** | No data hiding; all functions can access global data. | Data is hidden (private/protected) and only accessible via methods. | |
| |  |  |  | | --- | --- | --- | | **Code Reusability** | Requires repeated function calls. | Promotes reuse through inheritance and polymorphism. | |
| |  |  |  | | --- | --- | --- | | **Security** | Less secure since data is accessible globally. | More secure due to encapsulation and access control. | |
| |  |  |  | | --- | --- | --- | | **Scalability** | Difficult to manage for large projects. | More manageable and scalable due to modular design. | |
| |  |  |  | | --- | --- | --- | | **Examples** | C, early C++, Pascal | C++, Java, Python, C# | |

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

Ans. Object-Oriented Programming (OOP) has several advantages over Procedure-Oriented Programming (POP). Here are the main ones:

1. **Encapsulation**
   * OOP allows data to be wrapped inside objects using classes, restricting direct access and reducing data modification risks.
   * In POP, data is usually global and can be accessed by any function, making it prone to accidental changes.
2. **Reusability**
   * OOP supports code reuse through **inheritance**, where a new class can derive properties from an existing class.
   * POP lacks inheritance, leading to repetitive code and reduced maintainability.
3. **Scalability and Maintainability**
   * OOP programs are easier to modify and expand due to the modular structure of classes and objects.
   * In POP, changes in one function may affect many other functions, making debugging and maintenance difficult.
4. **Data Security**
   * OOP enforces **data hiding** using access specifiers (private, protected, public), preventing unauthorized access.
   * POP does not provide such data protection mechanisms.
5. **Code Organization and Modularity**
   * OOP promotes modularity by organizing related data and functions together in classes.
   * POP uses a top-down approach where functions operate on global data, leading to a less structured codebase.
6. **Abstraction**
   * OOP allows hiding complex implementation details and exposing only necessary functionalities via **abstract classes and interfaces**.
   * POP does not inherently support abstraction, making programs more complex as they grow.
7. **Polymorphism**
   * OOP supports polymorphism, allowing different classes to use the same method name but behave differently.
   * POP lacks this feature, requiring function names to be explicitly different for different functionalities.
8. **Better Problem-Solving Approach**
   * OOP models real-world entities more effectively, making it suitable for complex software applications like game development, GUI applications, and large-scale enterprise systems.
   * POP works well for small programs but becomes hard to manage for larger projects.
9. Explain the steps involved in setting up a C++ development environment

Ans. Setting up a **C++ development environment** involves several steps to ensure that you have the necessary tools and configurations to write, compile, and run C++ programs. Here's a step-by-step guide:

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

A compiler translates C++ code into executable machine code. Common options include:

* **GCC (GNU Compiler Collection)** – Available on Linux and macOS (via Xcode or Homebrew).
* **MinGW (Minimalist GNU for Windows)** – A GCC-based compiler for Windows.
* **MSVC (Microsoft Visual C++)** – Part of **Visual Studio**.
* **Clang** – A modern compiler for C++ that is often used on macOS.

👉 **Installation:**

* **Windows:** Install **MinGW-w64** from [Mingw-w64 website](https://www.mingw-w64.org/) or **Visual Studio with C++ build tools**.
* **Mac:** Install **Xcode Command Line Tools** using:

sh

CopyEdit

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

Ans. C++ provides several ways to handle **input** (taking data from the user) and **output** (displaying data to the user). The most commonly used methods are through the **iostream** library.

## ****1. Standard Output (****cout****)****

The cout (character output) stream is used to display output on the console.

📌 **Example: Printing a Message**

cpp

CopyEdit

#include <iostream> // Include the I/O library

using namespace std;

int main() {

cout << "Hello, World!" << endl; // Prints output

return 0;

}

🔹 **endl** moves the cursor to a new line (similar to \n).

## ****2. Standard Input (****cin****)****

The cin (character input) stream is used to take user input from the keyboard.

📌 **Example: Taking Integer Input**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age; // Takes input from the user

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

return 0;

}

🔹 **User enters a value, and cin stores it in the age variable**.

## ****3. Standard Error (****cerr****)****

The cerr stream is used to display error messages.

📌 **Example: Displaying an Error Message**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

cerr << "Error: Something went wrong!" << endl;

return 1;

}

🔹 **cerr is unbuffered, meaning it prints immediately.**

## ****4. Standard Log (****clog****)****

The clog stream is used for logging messages. Unlike cerr, it is **buffered**.

📌 **Example: Logging a Message**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

clog << "Logging: Program started successfully." << endl;

return 0;

}

🔹 **Useful for debugging and logging application events.**

## ****5. File I/O (****fstream****)****

For reading/writing files, C++ provides the <fstream> library.

📌 **Example: Writing to a File**

cpp

CopyEdit

#include <fstream>

using namespace std;

int main() {

ofstream file("output.txt");

file << "Hello, File!" << endl;

file.close();

return 0;

}

📌 **Example: Reading from a File**

cpp

CopyEdit

#include <fstream>

#include <iostream>

using namespace std;

int main() {

ifstream file("output.txt");

string content;

getline(file, content);

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

file.close();

return 0;

}

1. Variables, Data Types, and Operators

LAB EXERCISES

1. Variables and Constants
2. Write a C++ program that demonstrates the use of variables and constants. Create variables of different data types and perform operations on them.

Ans. #include <iostream>

using namespace std;

int main() {

int a = 10;

float b = 5.5;

double c = 12.3456;

char d = 'X';

bool isTrue = true;

const double PI = 3.14159;

int sum = a + 5;

float product = b \* 2.0;

double division = c / 2.0;

int mod = a % 3;

// Displaying results

cout << "Integer value (a): " << a << endl;

cout << "Float value (b): " << b << endl;

cout << "Double value (c): " << c << endl;

cout << "Character value (d): " << d << endl;

cout << "Boolean value (isTrue): " << isTrue << endl;

cout << "Constant PI: " << PI << endl;

cout << "Sum (a + 5): " << sum << endl;

cout << "Product (b \* 2.0): " << product << endl;

cout << "Division (c / 2.0): " << division << endl;

cout << "Modulus (a % 3): " << mod << endl;

return 0;

}

o Objective: Understand the difference between variables and constants.

Ans. #include <iostream>

using namespace std;

int main() {

int a = 10;

float b = 5.5;

double c = 12.3456;

char d = 'X';

bool isTrue = true;

// Declaring a constant

const double PI = 3.14159;

int sum = a + 5;

float product = b \* 2.0;

double division = c / 2.0;

int mod = a % 3;

cout << "Integer value (a): " << a << endl;

cout << "Float value (b): " << b << endl;

cout << "Double value (c): " << c << endl;

cout << "Character value (d): " << d << endl;

cout << "Boolean value (isTrue): " << isTrue << endl;

cout << "Constant PI: " << PI << endl;

cout << "Sum (a + 5): " << sum << endl;

cout << "Product (b \* 2.0): " << product << endl;

cout << "Division (c / 2.0): " << division << endl;

cout << "Modulus (a % 3): " << mod << endl;

return 0;

}

* 1. Type Conversion o

Write a C++ program that performs both implicit and explicit type conversions and prints the results.

Ans. #include <iostream>

using namespace std;

int main() {

int a = 10;

float b = 5.5;

double c = 12.3456;

char d = 'X';

bool isTrue = true;

const double PI = 3.14159;

int sum = a + 5;

float product = b \* 2.0;

double division = c / 2.0;

int mod = a % 3;

double implicitConversion = a + b;

int explicitConversion = (int)c;

cout << "Integer value (a): " << a << endl;

cout << "Float value (b): " << b << endl;

cout << "Double value (c): " << c << endl;

cout << "Character value (d): " << d << endl;

cout << "Boolean value (isTrue): " << isTrue << endl;

cout << "Constant PI: " << PI << endl;

cout << "Sum (a + 5): " << sum << endl;

cout << "Product (b \* 2.0): " << product << endl;

cout << "Division (c / 2.0): " << division << endl;

cout << "Modulus (a % 3): " << mod << endl;

cout << "Implicit conversion (a + b): " << implicitConversion << endl;

cout << "Explicit conversion (int)c: " << explicitConversion << endl;

return 0;

}

o Objective: Practice type casting in C++.

Ans. #include <iostream>

using namespace std;

int main() {

int a = 10;

float b = 5.5;

double c = 12.3456;

char d = 'X';

bool isTrue = true;

const double PI = 3.14159;

int sum = a + 5;

float product = b \* 2.0;

double division = c / 2.0;

int mod = a % 3;

double implicitConversion = a + b;

int explicitConversion = (int)c;

cout << "Integer value (a): " << a << endl;

cout << "Float value (b): " << b << endl;

cout << "Double value (c): " << c << endl;

cout << "Character value (d): " << d << endl;

cout << "Boolean value (isTrue): " << isTrue << endl;

cout << "Constant PI: " << PI << endl;

cout << "Sum (a + 5): " << sum << endl;

cout << "Product (b \* 2.0): " << product << endl;

cout << "Division (c / 2.0): " << division << endl;

cout << "Modulus (a % 3): " << mod << endl;

cout << "Implicit conversion (a + b): " << implicitConversion << endl;

cout << "Explicit conversion (int)c: " << explicitConversion << endl;

return 0;

}

1. Operator Demonstration

o Write a C++ program that demonstrates arithmetic, relational, logical, and bitwise operators. Perform operations using each type of operator and display the results.

Ans. #include <iostream>

using namespace std;

// Objective: Demonstrate arithmetic, relational, logical, and bitwise operators in C++.

int main() {

// Declaring variables of different data types

int a = 10, b = 5;

bool isTrue = true, isFalse = false;

// Declaring a constant

const double PI = 3.14159;

// Arithmetic operations

int sum = a + b;

int difference = a - b;

int product = a \* b;

double quotient = (double)a / b;

int mod = a % b;

// Relational operations

bool isEqual = (a == b);

bool isNotEqual = (a != b);

bool isGreater = (a > b);

bool isLesser = (a < b);

// Logical operations

bool logicalAnd = isTrue && isFalse;

bool logicalOr = isTrue || isFalse;

bool logicalNot = !isTrue;

// Bitwise operations

int bitwiseAnd = a & b;

int bitwiseOr = a | b;

int bitwiseXor = a ^ b;

int leftShift = a << 1;

int rightShift = a >> 1;

// Displaying results

cout << "Arithmetic Operations:" << endl;

cout << "Sum: " << sum << ", Difference: " << difference << ", Product: " << product << endl;

cout << "Quotient: " << quotient << ", Modulus: " << mod << endl;

cout << "Relational Operations:" << endl;

cout << "Equal: " << isEqual << ", Not Equal: " << isNotEqual << endl;

cout << "Greater: " << isGreater << ", Lesser: " << isLesser << endl;

cout << "Logical Operations:" << endl;

cout << "AND: " << logicalAnd << ", OR: " << logicalOr << ", NOT: " << logicalNot << endl;

cout << "Bitwise Operations:" << endl;

cout << "AND: " << bitwiseAnd << ", OR: " << bitwiseOr << ", XOR: " << bitwiseXor << endl;

cout << "Left Shift: " << leftShift << ", Right Shift: " << rightShift << endl;

return 0;

}

o Objective: Reinforce understanding of different types of operatorsin C++

Ans. #include <iostream>

using namespace std;

// Objective: Reinforce understanding of different types of operators in C++.

int main() {

// Declaring variables of different data types

int a = 10, b = 5;

bool isTrue = true, isFalse = false;

// Declaring a constant

const double PI = 3.14159;

// Arithmetic operations

int sum = a + b;

int difference = a - b;

int product = a \* b;

double quotient = (double)a / b;

int mod = a % b;

// Relational operations

bool isEqual = (a == b);

bool isNotEqual = (a != b);

bool isGreater = (a > b);

bool isLesser = (a < b);

// Logical operations

bool logicalAnd = isTrue && isFalse;

bool logicalOr = isTrue || isFalse;

bool logicalNot = !isTrue;

// Bitwise operations

int bitwiseAnd = a & b;

int bitwiseOr = a | b;

int bitwiseXor = a ^ b;

int leftShift = a << 1;

int rightShift = a >> 1;

// Displaying results

cout << "Arithmetic Operations:" << endl;

cout << "Sum: " << sum << ", Difference: " << difference << ", Product: " << product << endl;

cout << "Quotient: " << quotient << ", Modulus: " << mod << endl;

cout << "Relational Operations:" << endl;

cout << "Equal: " << isEqual << ", Not Equal: " << isNotEqual << endl;

cout << "Greater: " << isGreater << ", Lesser: " << isLesser << endl;

cout << "Logical Operations:" << endl;

cout << "AND: " << logicalAnd << ", OR: " << logicalOr << ", NOT: " << logicalNot << endl;

cout << "Bitwise Operations:" << endl;

cout << "AND: " << bitwiseAnd << ", OR: " << bitwiseOr << ", XOR: " << bitwiseXor << endl;

cout << "Left Shift: " << leftShift << ", Right Shift: " << rightShift << endl;

return 0;

}

THEORY EXERCISE:

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

Ans. C++ provides several fundamental data types, categorized as follows:

**1. Integer Types**

* Used to store whole numbers (positive, negative, or zero).
* Example:

cpp

CopyEdit

int num = 10; // Integer type

long bigNum = 1000000; // Long integer

short smallNum = 500; // Short integer

**2. Floating-Point Types**

* Used to store numbers with decimal points.
* Example:

cpp

CopyEdit

float pi = 3.14f; // Single precision

double precisePi = 3.141592653589793; // Double precision

**3. Character Type**

* Stores a single character enclosed in single quotes.
* Example:

cpp

CopyEdit

char letter = 'A';

**4. Boolean Type**

* Stores true or false.
* Example:

cpp

CopyEdit

bool isCodingFun = true;

**5. String Type (Using string from <string> library)**

* Stores sequences of characters.
* Example:

cpp

CopyEdit

#include <string>

string name = "John";

**6. Void Type**

* Used for functions that do not return a value.
* Example:

cpp

CopyEdit

void printMessage() {

cout << "Hello, World!";

}

**7. Wide Character Type (wchar\_t)**

* Used for extended character sets.
* Example:

cpp

CopyEdit

wchar\_t wideChar = L'A';

Each type serves a different purpose in programming. Let me know if you need more details or examples!

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

### Ans. Implicit vs. Explicit Type Conversion in C++

In C++, type conversion occurs when a value of one data type is converted into another. There are two main types of type conversion:

**1. Implicit Type Conversion (Type Promotion)**

* Done automatically by the compiler.
* Converts a smaller data type to a larger data type to prevent data loss.
* Example:

cpp

CopyEdit

int num = 10;

double result = num + 2.5; // int is implicitly converted to double

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

* **Why is it useful?**  
  Ensures precision and avoids data loss in operations involving mixed data types.

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

* Manually performed by the programmer using type casting.
* Used to force a specific type conversion.
* Example:

cpp

CopyEdit

double pi = 3.14159;

int intPi = (int)pi; // Explicitly converting double to int

cout << "Explicit Conversion: " << intPi; // Output: 3

* **Why is it useful?**  
  Helps control how values are converted and prevents unexpected behavior.

**Key Differences**

| **Feature** | **Implicit Conversion** | **Explicit Conversion** |
| --- | --- | --- |
| Performed by | Compiler | Programmer |
| Safety | Generally safe | Can lead to data loss |
| Syntax | Automatic | Requires type casting |
| Example | int -> double automatically | (int)doubleValue |

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

### Ans. Types of Operators in C++ with Examples

C++ provides various types of operators to perform different kinds of operations:

**1. Arithmetic Operators**

Used for mathematical operations.

| **Operator** | **Description** | **Example (a = 10, b = 5)** |
| --- | --- | --- |
| + | Addition | a + b → 15 |
| - | Subtraction | a - b → 5 |
| \* | Multiplication | a \* b → 50 |
| / | Division | a / b → 2 |
| % | Modulus (remainder) | a % b → 0 |

**2. Relational (Comparison) Operators**

Used to compare values.

| **Operator** | **Description** | **Example (a = 10, b = 5)** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | false |
| != | Not equal to | a != b | true |
| > | Greater than | a > b | true |
| < | Less than | a < b | false |
| >= | Greater than or equal to | a >= b | true |
| <= | Less than or equal to | a <= b | false |

**3. Logical Operators**

Used to combine multiple conditions.

| **Operator** | **Description** | **Example (isTrue = true, isFalse = false)** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND | isTrue && isFalse | False |
| ` |  | ` | Logical OR |
| ! | Logical NOT | !isTrue | False |

**4. Bitwise Operators**

Perform operations at the binary level.

| **Operator** | **Description** | **Example (a = 10 (1010), b = 5 (0101))** | **Result (Binary)** | **Result (Decimal)** |
| --- | --- | --- | --- | --- |
| & | AND | a & b | 0000 | 0 |
| ` | ` | OR | `a | b` |
| ^ | XOR | a ^ b | 1111 | 15 |
| << | Left Shift | a << 1 | 10100 | 20 |
| >> | Right Shift | a >> 1 | 0101 | 5 |

**5. Assignment Operators**

Used to assign values.

| **Operator** | **Description** | **Example (a = 10)** | **Equivalent To** |
| --- | --- | --- | --- |
| = | Assignment | a = b | a = 5 |
| += | Add and assign | a += b | a = a + b |
| -= | Subtract and assign | a -= b | a = a - b |
| \*= | Multiply and assign | a \*= b | a = a \* b |
| /= | Divide and assign | a /= b | a = a / b |
| %= | Modulus and assign | a %= b | a = a % b |

**6. Increment and Decrement Operators**

Used to increase or decrease a value by 1.

| **Operator** | **Description** | **Example (a = 10)** | **Result** |
| --- | --- | --- | --- |
| ++ | Increment | ++a (Pre-increment) | 11 |
| ++ | Increment | a++ (Post-increment) | 10 (then 11) |
| -- | Decrement | --a (Pre-decrement) | 9 |
| -- | Decrement | a-- (Post-decrement) | 10 (then 9) |

**7. Ternary Operator (?:)**

A shorthand for if-else.

cpp

CopyEdit

int x = 10, y = 5;

int min = (x < y) ? x : y; // Assigns the smaller value

cout << "Minimum: " << min; // Output: 5

**8. Type Casting Operator**

Used for explicit type conversion.

cpp

CopyEdit

double pi = 3.14159;

int intPi = (int)pi; // Explicit conversion

cout << intPi; // Output: 3

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

### Ans. ****Constants and Literals in C++****

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

A **constant** is a value that does not change throughout the execution of a program. Constants are useful for defining fixed values that should not be modified, improving code readability and maintainability.

##### **Types of Constants**

1. **const Keyword (Constant Variables)**
   * Declares a variable whose value cannot be changed.
   * Example:

cpp

CopyEdit

const double PI = 3.14159;

PI = 3.14; // ❌ Error: Cannot modify a constant variable

1. **#define Preprocessor Directive**
   * Used to define a macro (constant) before compilation.
   * Example:

cpp

CopyEdit

#define GRAVITY 9.81

cout << "Gravity: " << GRAVITY;

1. **constexpr (Compile-Time Constants)**
   * Ensures that the value is constant at compile-time.
   * Example:

cpp

CopyEdit

constexpr int MAX\_LIMIT = 100;

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

A **literal** is a fixed value assigned directly in the code. Literals do not require a variable to store them.

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

1. **Integer Literals** (Numbers without decimal)

cpp

CopyEdit

int num = 10; // 10 is an integer literal

1. **Floating-Point Literals** (Numbers with decimal)

cpp

CopyEdit

double pi = 3.14; // 3.14 is a floating-point literal

1. **Character Literals** (Single character in single quotes)

cpp

CopyEdit

char letter = 'A'; // 'A' is a character literal

1. **String Literals** (Sequence of characters in double quotes)

cpp

CopyEdit

string name = "John"; // "John" is a string literal

1. **Boolean Literals** (true or false)

cpp

CopyEdit

bool isCodingFun = true; // true is a boolean literal

* 1. Control Flow Statements

LAB EXERCISES:

1. Grade Calculator

Write a C++ program that takes a student’s marks as input and calculates the grade based on if-else conditions

Ans. #include <iostream>

using namespace std;

int main() {

float marks;

cout << "Enter the student's marks (0-100): ";

cin >> marks;

if (marks < 0 || marks > 100) {

cout << "Invalid input. Marks should be between 0 and 100." << endl;

} else {

// Calculate grade based on marks

if (marks >= 90) {

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

} else if (marks >= 80) {

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

} else if (marks >= 70) {

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

} else if (marks >= 60) {

cout << "Grade: D" << endl;

} else {

cout << "Grade: F" << endl;

}

}

return 0;

}

o Objective: Practice conditional statements (if-else).

Ans. The objective of practicing conditional statements (if-else) is to understand how decisions are made in programming based on different conditions. The if-else statement allows the program to choose different actions based on whether a given condition is true or false. This helps the program respond dynamically to input or different states.

Here’s how the C++ program fulfills this objective:

**Explanation of the Objective:**

* **Conditional Statements**: The if-else statements enable the program to check conditions and execute specific blocks of code accordingly.
  + **If condition**: Executes when a condition is true.
  + **Else condition**: Executes when the previous condition is false.
  + **Else-if**: Used to check multiple conditions sequentially.

The provided C++ program demonstrates these concepts:

1. **Taking User Input**: The program takes marks as input from the user.
2. **Using if-else Statements**: It uses if-else conditions to compare the input marks and calculate the student's grade.
3. **Multiple Conditions**: It checks different ranges of marks (like greater than or equal to 90, between 80 and 89, etc.) to assign the appropriate grade.

**Here’s the C++ program again with comments to emphasize the if-else logic:**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declare a variable to hold the marks

float marks;

// Prompt the user to enter the student's marks

cout << "Enter the student's marks (0-100): ";

cin >> marks;

// Check for valid marks input (0 <= marks <= 100)

if (marks < 0 || marks > 100) {

// If marks are outside the valid range, print an error message

cout << "Invalid input. Marks should be between 0 and 100." << endl;

} else {

// Conditional statements to calculate the grade based on the marks entered

if (marks >= 90) {

cout << "Grade: A" << endl; // Marks 90 or above = Grade A

} else if (marks >= 80) {

cout << "Grade: B" << endl; // Marks between 80 and 89 = Grade B

} else if (marks >= 70) {

cout << "Grade: C" << endl; // Marks between 70 and 79 = Grade C

} else if (marks >= 60) {

cout << "Grade: D" << endl; // Marks between 60 and 69 = Grade D

} else {

cout << "Grade: F" << endl; // Marks below 60 = Grade F

}

}

return 0;

}

**Key Points in the Program:**

1. **Using if**: The program first checks if the marks are outside the valid range. This is an example of using if to check a condition (marks less than 0 or greater than 100).
2. **Using else-if**: After ensuring the marks are valid, the program uses else-if to check the grade based on the marks in different ranges. This is an example of using multiple conditions to choose from several possibilities.
3. **Using else**: Finally, if none of the above conditions are met, the program falls back to the else block, which assigns a grade of "F" for marks less than 60.
4. Number Guessing Game

Write a C++ program that asks the user to guess a number between 1 and 100. The program should provide hints if the guess is too high or too low. Use loops to allow the user multiple attempts.

Ans. #include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

int main() {

// Seed the random number generator

srand(time(0));

// Generate a random number between 1 and 100

int secretNumber = rand() % 100 + 1;

// Declare the user's guess and the number of attempts

int userGuess;

int attempts = 0;

// Introduction message

cout << "Welcome to the Guessing Game!" << endl;

cout << "I'm thinking of a number between 1 and 100." << endl;

// Loop to allow multiple attempts

do {

// Ask the user to input their guess

cout << "Enter your guess: ";

cin >> userGuess;

// Increment the number of attempts

attempts++;

// Provide feedback based on the guess

if (userGuess < secretNumber) {

cout << "Your guess is too low! Try again." << endl;

} else if (userGuess > secretNumber) {

cout << "Your guess is too high! Try again." << endl;

} else {

cout << "Congratulations! You've guessed the correct number "

<< secretNumber << " in " << attempts << " attempts!" << endl;

}

} while (userGuess != secretNumber); // Repeat until the guess is correct

return 0;

}

o Objective: Understand while loops and conditional logic.

Ans. The objective of this program is to **understand while loops** and **conditional logic**.

**Key Concepts Covered:**

1. **While Loops**:
   * A **while loop** in C++ repeatedly executes a block of code as long as a specified condition is true.
   * In the context of the guessing game, the do-while loop (a variant of the while loop) allows the user to keep making guesses until they guess the correct number. The loop ensures the program keeps prompting the user for input without requiring extra code or manual repetition.

**Why a loop here?**

* + The loop allows the user to guess multiple times until they arrive at the correct answer. The loop only stops when the user guesses the correct number.
  + Without the loop, the program would only allow one guess and exit, but the do-while ensures the user can keep guessing.

1. **Conditional Logic**:
   * **Conditional statements (if-else)** are used to make decisions in the program based on the user's input (the guess).
   * The program compares the guessed number with the secret number using **if-else** conditions:
     + If the guess is too low, the program prints a message saying "Your guess is too low! Try again."
     + If the guess is too high, the program prints "Your guess is too high! Try again."
     + If the guess is correct, the program congratulates the user and ends the loop.

**Breakdown of the Program:**

1. **Random Number Generation**:
   * A random number between 1 and 100 is generated. This simulates a hidden number the user has to guess.
2. **Do-While Loop**:
   * The do-while loop runs the block of code where the user is prompted to guess, and it continues until the user's guess matches the secret number.
   * Inside the loop, after the user guesses:
     + The program checks if the guess is too low, too high, or correct using **if-else** statements.
     + The loop only exits when the guess is correct.
3. **Conditional Statements** (if-else):
   * **If**: Checks if the guess is less than the secret number and prints a hint.
   * **Else-If**: Checks if the guess is greater than the secret number and prints a different hint.
   * **Else**: If the guess is correct, prints a congratulatory message and the loop stops.

**Program Example with Explanation:**

cpp

CopyEdit

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

int main() {

// Seed the random number generator

srand(time(0));

// Generate a random number between 1 and 100

int secretNumber = rand() % 100 + 1;

// Declare the user's guess and the number of attempts

int userGuess;

int attempts = 0;

// Introduction message

cout << "Welcome to the Guessing Game!" << endl;

cout << "I'm thinking of a number between 1 and 100." << endl;

// Loop to allow multiple attempts

do {

// Ask the user to input their guess

cout << "Enter your guess: ";

cin >> userGuess;

// Increment the number of attempts

attempts++;

// Provide feedback based on the guess

if (userGuess < secretNumber) {

cout << "Your guess is too low! Try again." << endl;

} else if (userGuess > secretNumber) {

cout << "Your guess is too high! Try again." << endl;

} else {

cout << "Congratulations! You've guessed the correct number "

<< secretNumber << " in " << attempts << " attempts!" << endl;

}

} while (userGuess != secretNumber); // Repeat until the guess is correct

return 0;

}

**Objective Breakdown:**

1. **While Loops**:
   * The program uses a **do-while loop**. This type of loop is useful when you want the loop to run at least once before checking the condition (in this case, checking if the guess is correct).
   * The loop will keep running until userGuess != secretNumber (i.e., the user guesses correctly).
2. **Conditional Logic (If-Else)**:
   * The program uses **if** to check if the guess is too low or too high and gives the user feedback based on those conditions.
   * If neither of those conditions is true (i.e., the guess matches the secret number), the program prints the congratulatory message.

**Why is this a good exercise for learning while loops and conditional logic?**

1. **While Loop**: The program runs in a loop, demonstrating the concept of repeating actions until a condition (the correct guess) is met.
2. **Conditional Logic**: The program uses conditional statements to compare the guess with the secret number and make decisions (too high, too low, correct) based on those comparisons. This is a practical example of how conditions control the flow of the program.

By practicing with this game, you'll get a good grasp of how both loops and conditions work together to control program behavior!

1. Multiplication Table

Write a C++ program to display the multiplication table of a given number using a for loop.

Ans. Certainly! Below is a simple C++ program that takes a number as input from the user and displays its multiplication table using a **for loop**:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declare the variable to hold the number

int number;

// Ask the user to input the number

cout << "Enter a number to display its multiplication table: ";

cin >> number;

// Display the multiplication table using a for loop

cout << "Multiplication Table of " << number << ":\n";

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

cout << number << " x " << i << " = " << number \* i << endl;

}

return 0;

}

**Explanation:**

1. **Input**: The program prompts the user to enter a number.
2. **For Loop**:
   * The for loop iterates from i = 1 to i = 10. This covers all the numbers in the multiplication table (1 to 10).
   * On each iteration, the program multiplies the entered number by i and prints the result in the format "number x i = result".
3. **Output**: The multiplication table of the entered number is displayed on the screen.

**Example Output:**

css

CopyEdit

Enter a number to display its multiplication table: 5

Multiplication Table of 5:

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

5 x 10 = 50

**How the program works:**

* The user enters a number (e.g., 5).
* The for loop runs 10 times (from 1 to 10), and in each iteration, it calculates the product of the given number and the loop counter i, displaying the result in the multiplication format.

This is a great exercise to practice using loops for repetitive tasks, in this case generating a sequence of calculations!

o Objective: Practice using loops.

Ans. The objective of this program is to **practice using loops**, specifically the **for loop**.

**Key Concepts Covered:**

1. **For Loop**:
   * A **for loop** in C++ is used when you know beforehand how many times you need to repeat an action. In this case, the loop will run exactly 10 times to display the multiplication table of the given number.
   * The structure of a for loop is:

cpp

CopyEdit

for (initialization; condition; increment) {

// Code to be executed

}

1. **Repeating Actions**:
   * The multiplication table requires you to perform a repetitive task — multiplying a number by each of the integers from 1 to 10. A loop is ideal for this repetitive task because it avoids the need to manually write the multiplication for each number.

**Breakdown of the Program:**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declare a variable to hold the number

int number;

// Ask the user for the number

cout << "Enter a number to display its multiplication table: ";

cin >> number;

// Display the multiplication table using a for loop

cout << "Multiplication Table of " << number << ":\n";

// The for loop runs 10 times to display the table

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

// Display the result of multiplying the number by the current value of i

cout << number << " x " << i << " = " << number \* i << endl;

}

return 0;

}

**Explanation of Key Parts:**

* **For Loop**:
  + **Initialization**: int i = 1 — the loop starts with i equal to 1.
  + **Condition**: i <= 10 — the loop continues as long as i is less than or equal to 10 (i.e., until we've gone through the first 10 multiples).
  + **Increment**: ++i — after each iteration, i is increased by 1.
* **Repeating Action**:
  + The multiplication of number by i is performed in each iteration of the loop, and the result is displayed with cout.

**Why Use a For Loop Here?**

* **Repetition**: A for loop is perfect here because you want to repeat the multiplication 10 times (for the numbers 1 through 10).
* **Compact Code**: Instead of writing out the multiplication table manually, the for loop handles the repetitive task in a clean and efficient way.
* **Control**: You can easily modify the loop to generate tables for more or fewer numbers by changing the 10 in the loop condition to any other value.

**Example Output:**

css

CopyEdit

Enter a number to display its multiplication table: 7

Multiplication Table of 7:

7 x 1 = 7

7 x 2 = 14

7 x 3 = 21

7 x 4 = 28

7 x 5 = 35

7 x 6 = 42

7 x 7 = 49

7 x 8 = 56

7 x 9 = 63

7 x 10 = 70

**Objective of Practicing Loops:**

* The goal of this exercise is to practice the use of loops, which are essential for repeating tasks efficiently in programming.
* By using the **for loop**, you gain familiarity with how to control repetition, make your code more concise, and avoid redundancy.
  1. Nested Control Structures

o Write a program that prints a right-angled triangle using stars(\*) with a nested loop.

Ans. Sure! Below is a C++ program that prints a right-angled triangle using stars (\*), and it uses a **nested loop** for this task:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declare the number of rows for the triangle

int rows;

// Ask the user to input the number of rows

cout << "Enter the number of rows for the right-angled triangle: ";

cin >> rows;

// Outer loop for each row

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

// Inner loop for printing stars in each row

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

cout << "\*";

}

// Move to the next line after printing each row

cout << endl;

}

return 0;

}

**Explanation:**

1. **Outer Loop**:
   * The outer loop (for (int i = 1; i <= rows; ++i)) controls the number of rows. The variable i represents the current row number (starting from 1).
   * It runs rows times, which is the number of rows in the triangle.
2. **Inner Loop**:
   * The inner loop (for (int j = 1; j <= i; ++j)) prints the stars (\*) in each row. The number of stars printed in each row is determined by the current row number i.
   * For example, in the 1st row, it prints 1 star, in the 2nd row it prints 2 stars, and so on until the number of stars equals the row number.
3. **Newline**:
   * After printing the stars for each row, the cout << endl; statement moves the cursor to the next line, which ensures the triangle is printed correctly.

**Example Output:**

If the user inputs 5 for the number of rows, the output would look like this:

markdown

CopyEdit

Enter the number of rows for the right-angled triangle: 5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

**How the Nested Loop Works:**

* The outer loop runs 5 times (for 5 rows).
* For each iteration of the outer loop, the inner loop prints a number of stars equal to the current row number:
  + Row 1: 1 star
  + Row 2: 2 stars
  + Row 3: 3 stars
  + Row 4: 4 stars
  + Row 5: 5 stars

**Objective of Using Nested Loops:**

* The **nested loop** is useful for handling tasks where you need to perform repetitive actions in two dimensions (like rows and columns).
* In this case, the outer loop handles the number of rows, and the inner loop controls how many stars are printed in each row.

This is a great exercise for practicing the use of loops in a practical scenario like printing patterns

o Objective: Learn nested control structures.

Ans. The objective of this program is to **learn nested control structures**, specifically **nested loops**. Nested loops are loops inside of other loops, allowing us to perform more complex repetitive tasks.

**Key Concepts Covered:**

1. **Nested Loops**:
   * A **nested loop** refers to a loop inside another loop. The outer loop controls the number of times the inner loop will run.
   * In this program, the outer loop is responsible for handling the rows of the triangle, while the inner loop is responsible for printing the stars (\*) in each row.
2. **Control Structures**:
   * **Outer Loop**: This loop runs once for each row of the triangle. It iterates from 1 to rows (the user-defined number of rows).
   * **Inner Loop**: For each iteration of the outer loop (each row), the inner loop runs, printing a number of stars corresponding to the current row number.
   * **cout << endl**: After printing the stars for a row, the endl statement moves the output to the next line, so the stars print correctly in the shape of a triangle.

**Breakdown of the Program:**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declare the number of rows for the triangle

int rows;

// Ask the user to input the number of rows

cout << "Enter the number of rows for the right-angled triangle: ";

cin >> rows;

// Outer loop for each row

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

// Inner loop for printing stars in each row

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

cout << "\*";

}

// Move to the next line after printing each row

cout << endl;

}

return 0;

}

**Explanation:**

1. **Outer Loop**:
   * The outer loop iterates rows times (from 1 to the number of rows input by the user).
   * On each iteration of the outer loop, we are generating a new row of stars.
2. **Inner Loop**:
   * The inner loop prints stars (\*) for each row.
   * The number of stars printed corresponds to the current row number (i.e., on row 1, 1 star; on row 2, 2 stars; and so on).
3. **cout << endl**:
   * After printing all stars in a row, the endl is used to move the cursor to the next line to start a new row.

**Example Output:**

If the user inputs 5 for the number of rows, the output would look like this:

markdown

CopyEdit

Enter the number of rows for the right-angled triangle: 5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

**Objective of Learning Nested Control Structures:**

1. **Control Flow**:
   * The program demonstrates how nested control structures work, with one loop inside another.
   * The outer loop controls how many times the inner loop should execute. In this case, the outer loop controls how many rows are printed, and the inner loop controls how many stars are printed in each row.
2. **Handling Two Dimensions**:
   * Nested loops are useful when working with tasks that involve two-dimensional data or patterns, such as printing patterns, processing matrices, or creating grid-like structures.
   * Here, the outer loop controls the rows, and the inner loop controls the number of columns (stars) in each row.
3. **Efficiency**:
   * Understanding nested loops allows you to solve more complex problems efficiently with fewer lines of code compared to manually writing out repetitive tasks.

**Why is this a good exercise for learning nested control structures?**

* **Repetition with Variation**: The outer loop is repeated for every row, while the inner loop is repeated for a variable number of stars depending on the current row.
* **Two Levels of Control**: You get to practice controlling two different loops to achieve the desired result. Nested loops are often used in problems where tasks need to be repeated within other repetitive tasks, such as in games, simulations, or table generation.
* **Pattern Creation**: The nested loop is ideal for printing patterns, which is a common exercise to practice loops.

By working with nested loops in this triangle program, you'll gain a deeper understanding of how multiple loops can work together to create more complex patterns or solutions.

THEORY EXERCISE:

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

Ans. In C++, **conditional statements** allow you to control the flow of execution of your program based on certain conditions. These statements help you decide which block of code should be executed when certain conditions are met or not met.

C++ provides several types of conditional statements:

1. **if statement**
2. **if-else statement**
3. **else-if ladder**
4. **switch statement**

Let's dive into the details of the **if-else** and **switch** statements.

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

The if-else statement is the most basic form of conditional control. It allows you to execute one block of code if a condition is true, and another block if the condition is false.

#### Syntax:

cpp

CopyEdit

if (condition) {

// Code block to be executed if condition is true

} else {

// Code block to be executed if condition is false

}

* **condition**: A boolean expression that evaluates to either true or false.
* **if block**: This block of code executes if the condition evaluates to true.
* **else block**: This block of code executes if the condition evaluates to false.

#### Example:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter a number: ";

cin >> number;

if (number > 0) {

cout << "The number is positive." << endl;

} else {

cout << "The number is not positive." << endl;

}

return 0;

}

* **Explanation**:
  + If the number entered is greater than 0, it prints "The number is positive."
  + If the number is 0 or negative, it prints "The number is not positive."

### 2. ****switch Statement****

The switch statement is another form of conditional control, but it is used when you need to compare a single variable (or expression) against multiple values. It's an alternative to using multiple if-else conditions when the conditions are based on specific values (like integers, characters, or enums).

#### Syntax:

cpp

CopyEdit

switch (expression) {

case value1:

// Code block for value1

break;

case value2:

// Code block for value2

break;

default:

// Code block if none of the cases match

}

* **expression**: The value to be checked. It can be an integer, character, or an expression that evaluates to a value.
* **case value**: These are the possible values for which the program will execute a specific block of code.
* **break**: The break statement is used to exit the switch block once a match is found. If break is omitted, the program will continue checking subsequent case statements, which is called "fall through."
* **default**: This block executes if no case matches the expression. It’s optional, but it can be helpful to handle unexpected cases.

#### Example:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int day;

cout << "Enter a number (1-7) to find the corresponding day of the week: ";

cin >> day;

switch (day) {

case 1:

cout << "Sunday" << endl;

break;

case 2:

cout << "Monday" << endl;

break;

case 3:

cout << "Tuesday" << endl;

break;

case 4:

cout << "Wednesday" << endl;

break;

case 5:

cout << "Thursday" << endl;

break;

case 6:

cout << "Friday" << endl;

break;

case 7:

cout << "Saturday" << endl;

break;

default:

cout << "Invalid input. Please enter a number between 1 and 7." << endl;

}

return 0;

}

* **Explanation**:
  + The program asks the user to enter a number (from 1 to 7), corresponding to a day of the week.
  + The switch statement checks the entered value (day) against the case values.
  + If a match is found, the corresponding day is printed.
  + If no valid day is entered (i.e., a number not between 1 and 7), the default case prints an error message.

### Comparison: ****if-else vs. switch****

* **Use if-else**:
  + When you have complex conditions that involve relational or logical operators (e.g., x > 10 && x < 20).
  + When you need to check conditions on ranges or floating-point numbers.
* **Use switch**:
  + When you're checking a single expression against many possible constant values.
  + For handling discrete, non-range values (like integers, characters, or enums).
  + switch statements are generally more readable and efficient when dealing with multiple possible values of a single variable.

### Summary:

1. **if-else**:
   * Best for conditions that evaluate true/false based on comparisons (e.g., number > 10).
   * More flexible but can become cumbersome with many conditions.
2. **switch**:
   * Best for selecting one of many possible values of a variable (e.g., case 1, case 2,...).
   * Efficient for handling multiple options based on one variable but limited to discrete values and constants.

Both conditional structures are fundamental in controlling program flow based on conditions, and they help you write logic that adapts to different inputs.

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

Ans In C++, the **for**, **while**, and **do-while** loops are all used to execute a block of code repeatedly, but they differ in terms of **how and when** the condition is evaluated and how the loops are structured. Let's go through the differences:

### 1. ****for loop****

#### Structure:

cpp

CopyEdit

for (initialization; condition; increment/decrement) {

// Code to be executed repeatedly

}

* **Initialization**: This part initializes a loop counter or variable.
* **Condition**: This part checks if the loop should continue. If the condition evaluates to true, the loop continues; otherwise, it stops.
* **Increment/Decrement**: This part modifies the loop variable (or counter) after each iteration, making the loop progress toward a stopping condition.

#### Characteristics:

* The **for loop** is best when you know **how many times** you want the loop to run beforehand.
* The condition is checked **before** entering the loop (i.e., the loop may not run at all if the condition is false initially).
* It is often used when the number of iterations is predetermined (e.g., iterating through an array or a fixed number of repetitions).

#### Example:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

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

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

}

return 0;

}

* **Output**:

nginx

CopyEdit

Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5

### 2. ****while loop****

#### Structure:

cpp

CopyEdit

while (condition) {

// Code to be executed repeatedly

}

* The **while loop** evaluates the **condition before** each iteration. If the condition is true, the code inside the loop executes. If it is false initially, the loop will not execute at all.

#### Characteristics:

* The **while loop** is best when you want the loop to run **as long as a condition remains true**.
* It is typically used when the number of iterations is **not known beforehand**, and the loop continues based on some condition.
* The condition is evaluated **before** the body of the loop, so if the condition is false at the start, the body may never execute.

#### Example:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int i = 1;

while (i <= 5) {

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

++i;

}

return 0;

}

* **Output**:

nginx

CopyEdit

Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5

### 3. ****do-while loop****

#### Structure:

cpp

CopyEdit

do {

// Code to be executed repeatedly

} while (condition);

* The **do-while loop** is similar to the **while loop**, but it evaluates the **condition after** executing the code block. Therefore, the code inside the loop will always execute at least once, regardless of whether the condition is true or false at the beginning.

#### Characteristics:

* The **do-while loop** is best when you want the loop to **run at least once** before checking the condition.
* It is typically used when the loop body needs to execute **before** the condition is checked, ensuring that the body executes at least once even if the condition is initially false.

#### Example:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int i = 1;

do {

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

++i;

} while (i <= 5);

return 0;

}

* **Output**:

nginx

CopyEdit

Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5

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

| **Aspect** | **for loop** | **while loop** | **do-while loop** |
| --- | --- | --- | --- |
| **Condition Check** | Before entering the loop. | Before each iteration. | After each iteration. |
| **Loop Guarantee** | May not execute at all if the condition is false initially. | May not execute if the condition is false initially. | Always executes at least once, regardless of the condition. |
| **Best Used When** | You know how many times you want to iterate (fixed iterations). | You don’t know how many times but want to continue while a condition holds true. | You want the loop to run at least once, regardless of the condition. |
| **Example Use** | Iterating over an array, or a known number of iterations. | Reading input until a valid value is entered. | Asking for user input at least once. |

### Summary:

* **for loop**: Ideal when the number of iterations is known beforehand. The condition is checked **before** the loop starts.
* **while loop**: Useful when the number of iterations is unknown, but the loop continues as long as the condition is true. The condition is checked **before** each iteration, and if it's false initially, the loop won't run.
* **do-while loop**: Guarantees that the loop will execute at least once, as the condition is checked **after** the loop body. This is useful when you want to perform an action at least once, even if the condition is initially false.

These three loops give you flexibility in how you manage repeating tasks in your C++ programs, depending on whether you need to control when the condition is evaluated and how many times the loop runs.

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

Ans. In C++, the **break** and **continue** statements are used to control the flow of execution within loops. They modify the normal execution of the loop in different ways.

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

The **break** statement is used to **exit** a loop prematurely, regardless of whether the loop's condition has been satisfied. When a break statement is encountered inside a loop, the loop terminates immediately, and the program continues executing from the statement that follows the loop.

#### Use Cases for break:

* **Exit a loop early** when a certain condition is met.
* **Terminate a loop** when further iterations are unnecessary (e.g., finding a result in a search).

#### Syntax:

cpp

CopyEdit

for (/\* initialization \*/; /\* condition \*/; /\* increment \*/) {

if (/\* some condition \*/) {

break; // Exit the loop

}

// Other code

}

#### Example 1: Using break in a for loop

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

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

if (i == 5) {

break; // Exit the loop when i equals 5

}

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

}

cout << "Loop exited." << endl;

return 0;

}

* **Explanation**:
  + The loop will run for values of i from 1 to 10.
  + When i reaches 5, the break statement is encountered, and the loop exits early.
  + The output will be:

makefile

CopyEdit

i = 1

i = 2

i = 3

i = 4

Loop exited.

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

The **continue** statement is used to **skip the current iteration** of a loop and proceed with the next iteration. When continue is encountered, the remaining code inside the loop is skipped for the current iteration, and the loop moves to the next iteration.

#### Use Cases for continue:

* **Skip certain iterations** of a loop without terminating the entire loop.
* **Skip unwanted conditions** inside a loop (e.g., when processing specific elements but ignoring others).

#### Syntax:

cpp

CopyEdit

for (/\* initialization \*/; /\* condition \*/; /\* increment \*/) {

if (/\* some condition \*/) {

continue; // Skip this iteration and proceed to the next

}

// Other code

}

#### Example 2: Using continue in a for loop

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

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

if (i == 5) {

continue; // Skip the iteration when i equals 5

}

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

}

return 0;

}

* **Explanation**:
  + The loop will run for values of i from 1 to 10.
  + When i equals 5, the continue statement is encountered, which skips the print statement for that iteration and moves to the next iteration.
  + The output will be:

ini

CopyEdit

i = 1

i = 2

i = 3

i = 4

i = 6

i = 7

i = 8

i = 9

i = 10

### ****Key Differences between**** break ****and**** continue

| **Aspect** | **break** | **continue** |
| --- | --- | --- |
| **Purpose** | Exits the loop entirely, terminating it. | Skips the current iteration and moves to the next one. |
| **Flow of Execution** | The program exits the loop completely. | The program skips the rest of the loop body for that iteration. |
| **Where It Is Used** | Can be used in for, while, or do-while loops. | Can be used in for, while, or do-while loops. |
| **Effect on Iteration** | The loop stops entirely. | The loop continues with the next iteration. |

### Example 3: Using break and continue Together

In a scenario where we need to both skip certain iterations and exit the loop early, we can use both break and continue in a single loop.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

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

if (i == 3) {

continue; // Skip when i equals 3

}

if (i == 8) {

break; // Exit the loop when i equals 8

}

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

}

return 0;

}

* **Explanation**:
  + The loop prints the value of i for all numbers from 1 to 10.
  + When i equals 3, the continue statement skips the rest of the loop body and moves to the next iteration.
  + When i equals 8, the break statement causes the loop to terminate completely.
  + The output will be:

ini

CopyEdit

i = 1

i = 2

i = 4

i = 5

i = 6

i = 7

### ****Summary****

* **break**: Exits the loop entirely, stopping any further iterations.
  + **Example**: Use it to stop the loop early when a specific condition is met.
* **continue**: Skips the rest of the current iteration and moves to the next iteration of the loop.
  + **Example**: Use it to skip over certain elements (like skipping odd numbers, or skipping unwanted conditions) without stopping the loop.

Both break and continue help provide more control over the flow of loops, allowing you to skip iterations or terminate loops early when certain conditions are met.

1. Explain nested control structures with an example.

### Ans. Nested Control Structures in C++

A **nested control structure** refers to a situation where one control structure (such as a loop or a conditional statement) is placed inside another. Essentially, a control structure can contain other control structures, allowing for more complex decision-making and iteration processes.

There are two primary types of control structures that can be nested in C++:

1. **Nested if-else statements**
2. **Nested loops** (e.g., for, while, do-while)

These allow you to implement more intricate decision-making processes and repetitive actions, where one set of decisions or actions depends on the outcome of a previous one.

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

In a **nested if-else** structure, one if or else block contains another if-else structure. This allows you to evaluate multiple conditions in a hierarchical manner.

#### Syntax:

cpp

CopyEdit

if (condition1) {

if (condition2) {

// Code to be executed if both conditions are true

} else {

// Code to be executed if condition1 is true but condition2 is false

}

} else {

// Code to be executed if condition1 is false

}

#### Example: Nested if-else to check multiple conditions

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

if (age >= 18) {

if (age >= 65) {

cout << "You are a senior citizen." << endl;

} else {

cout << "You are an adult." << endl;

}

} else {

cout << "You are a minor." << endl;

}

return 0;

}

* **Explanation**:
  + If the age is 18 or above, a second check determines whether the person is 65 or older.
  + If the age is less than 18, the program states that the person is a minor.
  + The **output** will depend on the value entered for age.

#### Example Output:

* Input: 20

sql

CopyEdit

You are an adult.

* Input: 70

css

CopyEdit

You are a senior citizen.

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

In a **nested loop**, one loop (inner loop) is placed inside another loop (outer loop). The inner loop completes all its iterations for each single iteration of the outer loop.

#### Example: Nested for loop to print a pattern

Let's say you want to print a right-angled triangle of stars (\*).

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int rows = 5; // Number of rows in the triangle

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

for (int j = 1; j <= i; ++j) { // Inner loop for columns (stars in each row)

cout << "\*"; // Print star

}

cout << endl; // Move to the next line after each row

}

return 0;

}

* **Explanation**:
  + The outer loop controls the number of rows.
  + The inner loop controls the number of stars printed in each row.
  + In each iteration of the outer loop, the inner loop runs until the number of stars printed matches the current row number (i).
  + **Output**:

markdown

CopyEdit

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

### ****3. Nested**** while ****Loop****

Similarly, you can nest while loops within each other. Here's an example of printing the same right-angled triangle using nested while loops.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int rows = 5;

int i = 1; // Outer loop counter

while (i <= rows) { // Outer while loop for rows

int j = 1; // Inner loop counter for columns

while (j <= i) { // Inner while loop for stars

cout << "\*";

++j;

}

cout << endl; // Move to the next line after printing stars

++i;

}

return 0;

}

* **Explanation**:
  + The outer while loop controls the rows (like in the previous for loop example).
  + The inner while loop controls how many stars (\*) are printed in each row.
  + **Output**:

markdown

CopyEdit

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

### ****4. Nested Loops with**** break ****and**** continue

You can also combine nested loops with break and continue statements. Here's an example that prints a number pattern, but skips certain rows using continue:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int rows = 5;

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

if (i == 3) {

continue; // Skip row 3

}

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

cout << j << " "; // Print numbers in each row

}

cout << endl;

}

return 0;

}

* **Explanation**:
  + The outer loop iterates through the rows.
  + If the row number is 3, the continue statement skips that iteration, so no numbers will be printed in row 3.
  + **Output**:

CopyEdit

1

1 2

1 2 3

1 2 4

### Summary of Nested Control Structures:

1. **Nested if-else Statements**:
   * One if or else block is placed inside another if-else structure.
   * Allows for hierarchical decision-making based on multiple conditions.
   * Useful when there are multiple levels of checks.
2. **Nested Loops**:
   * One loop is placed inside another loop, where the inner loop executes completely for every iteration of the outer loop.
   * Typically used when working with multi-dimensional structures like matrices or when generating patterns.

**Examples**:

* + Printing patterns, tables, grids.
  + Iterating through multi-dimensional arrays (like matrices).

1. **Combining Nested Loops with break and continue**:
   * Nested loops can be controlled with break (to exit) or continue (to skip an iteration).
   * Allows for more dynamic control within loops.

### Final Thoughts:

* Nested control structures are powerful tools for handling complex logic in C++.
* Proper use of nested loops and conditionals enables you to handle multi-level decision-making, repetitive tasks, and even generate complex output patterns. However, excessive nesting can make code harder to understand and maintain, so it's essential to use them thoughtfully and when necessary.

1. Functions and Scope

LAB EXERCISES:

1. Simple Calculator Using Functions

Write a C++ program that defines functions for basic arithmetic operations (add, subtract, multiply, divide). The main function should call these based on user input.

Ans. Here’s a C++ program that defines functions for basic arithmetic operations (addition, subtraction, multiplication, and division). The main function asks the user to input the operation they wish to perform, and then calls the appropriate function.

**C++ Program:**

cpp

CopyEdit

#include <iostream>

using namespace std;

// Function to add two numbers

double add(double a, double b) {

return a + b;

}

// Function to subtract two numbers

double subtract(double a, double b) {

return a - b;

}

// Function to multiply two numbers

double multiply(double a, double b) {

return a \* b;

}

// Function to divide two numbers

double divide(double a, double b) {

if (b != 0) {

return a / b;

} else {

cout << "Error! Division by zero." << endl;

return 0;

}

}

int main() {

double num1, num2;

char operation;

// Prompt user for input

cout << "Enter first number: ";

cin >> num1;

cout << "Enter second number: ";

cin >> num2;

// Ask for the operation

cout << "Enter operation (+, -, \*, /): ";

cin >> operation;

// Perform the operation based on the user's input

switch (operation) {

case '+':

cout << "Result: " << add(num1, num2) << endl;

break;

case '-':

cout << "Result: " << subtract(num1, num2) << endl;

break;

case '\*':

cout << "Result: " << multiply(num1, num2) << endl;

break;

case '/':

cout << "Result: " << divide(num1, num2) << endl;

break;

default:

cout << "Invalid operation!" << endl;

break;

}

return 0;

}

**Explanation:**

1. **Function Definitions:**
   * add(): Adds two numbers and returns the result.
   * subtract(): Subtracts the second number from the first and returns the result.
   * multiply(): Multiplies two numbers and returns the result.
   * divide(): Divides the first number by the second and checks for division by zero (if the second number is zero, it prints an error message).
2. **Main Function:**
   * Prompts the user for two numbers and the desired arithmetic operation (+, -, \*, /).
   * Uses a switch statement to call the corresponding arithmetic function based on the user's input.
3. **Error Handling:**
   * The divide() function checks if the denominator (b) is zero to prevent division by zero errors.

**Example Output:**

* **Input:**

mathematica

CopyEdit

Enter first number: 10

Enter second number: 5

Enter operation (+, -, \*, /): +

* **Output:**

makefile

CopyEdit

Result: 15

* **Input:**

mathematica

CopyEdit

Enter first number: 10

Enter second number: 5

Enter operation (+, -, \*, /): /

* **Output:**

makefile

CopyEdit

Result: 2

* **Input:**

mathematica

CopyEdit

Enter first number: 10

Enter second number: 0

Enter operation (+, -, \*, /): /

* **Output:**

vbnet

CopyEdit

Error! Division by zero.

This program provides a simple, interactive calculator using functions and conditional logic for the four basic arithmetic operations.

o Objective: Practice defining and using functions in C++.

Ans. A function in C++ consists of:

1. **Return Type** - Specifies the type of value the function returns.
2. **Function Name** - The identifier of the function.
3. **Parameters (optional)** - Input values to the function.
4. **Function Body** - The block of code that executes when the function is called.

**Basic Function Example**

cpp

CopyEdit

#include <iostream>

using namespace std;

// Function definition

int add(int a, int b) {

return a + b; // Returns the sum of a and b

}

int main() {

int num1 = 5, num2 = 10;

// Function call

int result = add(num1, num2);

cout << "Sum: " << result << endl;

return 0;

}

**Types of Functions**

1. **Functions with No Return Type (void)**

cpp

CopyEdit

void greet() {

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

}

1. **Functions with Parameters**

cpp

CopyEdit

int multiply(int x, int y) {

return x \* y;

}

1. **Functions with Default Arguments**

cpp

CopyEdit

int subtract(int a, int b = 5) {

return a - b;

}

1. **Inline Functions (For Small Functions)**

cpp

CopyEdit

inline int square(int x) {

return x \* x;

}

1. **Function Overloading (Same Name, Different Parameters)**

cpp

CopyEdit

int area(int side) { return side \* side; }

int area(int length, int width) { return length \* width; }

1. Factorial Calculation Using Recursion

o Write a C++ program that calculates the factorial of a number using recursion.

Ans. #include <iostream>

using namespace std;

// Function to calculate factorial using recursion

long long factorial(int n) {

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

return 1;

else

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

}

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

if (num < 0)

cout << "Factorial is not defined for negative numbers." << endl;

else

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

return 0;

}

o Objective: Understand recursion in functions.

Ans. Recursion is a programming technique where a function **calls itself** to solve a smaller version of the original problem. Every recursive function must have:

1. **Base Case** - The condition that stops recursion.
2. **Recursive Case** - The part where the function calls itself.
3. Variable Scope

Write a program that demonstrates the difference between local and global variables in C++. Use functions to show scope.

Ans. #include <iostream>

using namespace std;

// Global variable (accessible in all functions)

int globalVar = 10;

void showVariables() {

int localVar = 5; // Local variable (only inside this function)

cout << "Inside showVariables function:" << endl;

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

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

}

int main() {

int localVar = 20; // Local variable in main (different from showVariables' localVar)

cout << "Inside main function:" << endl;

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

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

showVariables(); // Calling function to see scope in action

// Modifying the global variable

globalVar = 50;

cout << "Global Variable after modification in main: " << globalVar << endl;

return 0;

}

o Objective: Reinforce the concept of variable scope.

Ans. #include <iostream>

using namespace std;

// Global variable (accessible anywhere)

int globalVar = 10;

void functionA() {

int localVar = 5; // Local to functionA

static int staticVar = 0; // Static variable (retains value across calls)

cout << "Inside functionA:" << endl;

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

cout << "Static Variable: " << staticVar << " (Retains value)" << endl;

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

// Modify static and global variables

staticVar++;

globalVar += 5;

}

int main() {

int localVar = 20; // Local to main

cout << "Inside main function:" << endl;

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

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

functionA(); // Call functionA

functionA(); // Call again to see static variable in action

cout << "Global Variable after functionA calls: " << globalVar << endl;

return 0;

}

THEORY EXERCISE:

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

Ans. A **function** in C++ is a block of code that performs a specific task. It helps in **code reusability, modularity, and readability** by breaking a program into smaller, manageable parts.

## ****Function Components in C++****

A function has three key components:

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

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

A function must be **declared** before it is used. The function declaration (prototype) tells the compiler about:

* The function's **name**.
* The **return type**.
* The **parameters** (if any).

💡 **Example of Function Declaration:**

cpp

CopyEdit

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

Here, add is a function that takes two int parameters and returns an int.

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

This is where the **actual code** of the function is written. The definition includes:

* The **return type**.
* The **function name**.
* The **parameters**.
* The **body** (code to execute).

💡 **Example of Function Definition:**

cpp

CopyEdit

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

return a + b;

}

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

To execute a function, you need to call it from the main() function (or another function). The function call uses:

* The **function name**.
* The **arguments** (if required).

💡 **Example of Function Call:**

cpp

CopyEdit

int sum = add(5, 10); // Function call with arguments

Here, 5 and 10 are passed to the add function, which returns 15.

## ****Complete C++ Example with Functions****

cpp

CopyEdit

#include <iostream>

using namespace std;

// Function Declaration (Prototype)

int add(int, int);

int main() {

int num1 = 5, num2 = 10;

// Function Call

int result = add(num1, num2);

cout << "Sum: " << result << endl;

return 0;

}

// Function Definition

int add(int a, int b) {

return a + b;

}

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

Ans. The **scope** of a variable in C++ determines **where** in the program a variable can be accessed. There are two main types of scope:

1. **Local Scope (Local Variables)**
2. **Global Scope (Global Variables)**

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

* **Declared inside a function or block {}.**
* **Accessible only within that function/block.**
* **Destroyed when the function/block execution ends.**

### ****Example of Local Variable:****

cpp

CopyEdit

#include <iostream>

using namespace std;

void display() {

int localVar = 10; // Local variable

cout << "Inside display function, localVar: " << localVar << endl;

}

int main() {

// cout << localVar; // ERROR: localVar is not accessible here

display();

return 0;

}

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

scss

CopyEdit

Inside display function, localVar: 10

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

* **Declared outside all functions.**
* **Accessible from any function in the program.**
* **Retains value throughout the program execution.**

### ****Example of Global Variable:****

cpp

CopyEdit

#include <iostream>

using namespace std;

int globalVar = 20; // Global variable

void display() {

cout << "Inside display function, globalVar: " << globalVar << endl;

}

int main() {

cout << "Inside main function, globalVar: " << globalVar << endl;

display();

return 0;

}

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

bash

CopyEdit

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

Ans. **Recursion** is a programming technique where a function **calls itself** to solve a problem. Instead of using loops, recursion breaks the problem into smaller subproblems.

### ****Key Components of Recursion****

1. **Base Case** → The condition that stops recursion.
2. **Recursive Case** → The function calls itself with a smaller input.

## ****Example: Factorial Using Recursion****

Factorial of n (**n!**) is calculated as:

n!=n×(n−1)×(n−2)×...×1n! = n \times (n - 1) \times (n - 2) \times ... \times 1n!=n×(n−1)×(n−2)×...×1

With **Base Case**:

0!=10! = 10!=1

### ****C++ Program for Factorial Using Recursion****

cpp

CopyEdit

#include <iostream>

using namespace std;

// Recursive function to calculate factorial

long long factorial(int n) {

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

return 1;

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

}

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

if (num < 0)

cout << "Factorial is not defined for negative numbers." << endl;

else

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

return 0;

}

## ****How Recursion Works in This Case****

For factorial(5), the function calls itself as follows:

scss

CopyEdit

factorial(5) → 5 \* factorial(4)

factorial(4) → 4 \* factorial(3)

factorial(3) → 3 \* factorial(2)

factorial(2) → 2 \* factorial(1)

factorial(1) → 1 (Base Case)

Now, resolving back:

matlab

CopyEdit

factorial(2) = 2 \* 1 = 2

factorial(3) = 3 \* 2 = 6

factorial(4) = 4 \* 6 = 24

factorial(5) = 5 \* 24 = 120

So, factorial(5) = 120.

## ****Another Example: Fibonacci Series Using Recursion****

The Fibonacci sequence is:

CopyEdit

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Where:

F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2)

With base cases:

F(0)=0,F(1)=1F(0) = 0, \quad F(1) = 1F(0)=0,F(1)=1

### ****C++ Program for Fibonacci Using Recursion****

cpp

CopyEdit

#include <iostream>

using namespace std;

int fibonacci(int n) {

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

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

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

}

int main() {

int num;

cout << "Enter the position in Fibonacci series: ";

cin >> num;

cout << "Fibonacci number at position " << num << " is: " << fibonacci(num) << endl;

return 0;

}

### ****Example Output****

mathematica

CopyEdit

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

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

* The **function name**
* The **return type**
* The **parameters (type and order)**

It is placed **before the main function** to inform the compiler about a function that will be **defined later**.

5. Arrays and Strings

1. Array Sum and Average

Write a C++ program that accepts an array of integers, calculates the sum and average, and displays the results.

Ans. #include <iostream>

using namespace std;

void calculateSumAndAverage(int arr[], int size, int &sum, double &average) {

sum = 0;

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

sum += arr[i];

}

average = static\_cast<double>(sum) / size; // Convert to double for precision

}

int main() {

int n;

// User input for array size

cout << "Enter the number of elements: ";

o Objective: Understand basic array manipulation.

Ans. Arrays are collections of **elements of the same type** stored in contiguous memory locations.

**Syntax:**

cpp

CopyEdit

type arrayName[arraySize];

* **type**: The data type of the elements (e.g., int, double).
* **arrayName**: The name of the array.
* **arraySize**: The number of elements in the array.

Example:

cpp

CopyEdit

int arr[5]; // Array of 5 integers

**2. Initializing Arrays**

Arrays can be initialized at the time of declaration or later.

**Example (at declaration):**

cpp

CopyEdit

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

**Example (later):**

cpp

CopyEdit

int arr[5]; // Declare

arr[0] = 1; // Initialize individual elements

arr[1] = 2;

**3. Accessing Array Elements**

Array elements are accessed using their **index** (starting from 0).

**Example:**

cpp

CopyEdit

int firstElement = arr[0]; // Access the first element

**4. Manipulating Arrays (Example in the Program)**

In our **sum and average program**, we used an array to store **integers** input by the user. Here’s how we manipulated the array:

1. **Input**:
   * We used a loop to accept **user input** for each element of the array.

cpp

CopyEdit

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

cin >> arr[i]; // Accept input for each element

}

1. **Sum Calculation**:
   * We used a loop to calculate the **sum** of the array's elements by **iterating** through them.

cpp

CopyEdit

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

sum += arr[i]; // Sum the elements

}

1. **Average Calculation**:
   * We calculated the **average** by dividing the sum by the number of elements in the array.

cpp

CopyEdit

average = static\_cast<double>(sum) / size; // Calculate average

**5. Modifying Arrays**

Arrays can be modified at any index, and their values can be updated.

**Example:**

cpp

CopyEdit

arr[2] = 100; // Change the third element (index 2) to 100

**6. Passing Arrays to Functions**

In C++, arrays are **passed by reference** to functions, meaning the function can modify the array's contents.

In the earlier program, we passed the array arr to the function calculateSumAndAverage:

cpp

CopyEdit

calculateSumAndAverage(arr, n, sum, average);

**Key Takeaways:**

* Arrays store **multiple values** of the same type.
* **Manipulation** involves accessing, modifying, and processing array elements using their index.
* Arrays can be passed to functions to **perform operations** like calculating sums or averages.

2. Matrix Addition

o Write a C++ program to perform matrix addition on two 2x2 matrices.

Ans. Matrix addition is done by adding corresponding elements of two matrices of the same size.

For example, if you have two 2x2 matrices:

Matrix A:

nginx

CopyEdit

a11 a12

a21 a22

Matrix B:

nginx

CopyEdit

b11 b12

b21 b22

The sum of these matrices will be:

Matrix C (Result):

ini

CopyEdit

c11 = a11 + b11 c12 = a12 + b12

c21 = a21 + b21 c22 = a22 + b22

**C++ Program: Matrix Addition**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declare two 2x2 matrices and a result matrix

int A[2][2], B[2][2], C[2][2];

// Input for Matrix A

cout << "Enter elements of Matrix A (2x2):" << endl;

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

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

cout << "A[" << i << "][" << j << "]: ";

cin >> A[i][j];

}

}

// Input for Matrix B

cout << "Enter elements of Matrix B (2x2):" << endl;

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

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

cout << "B[" << i << "][" << j << "]: ";

cin >> B[i][j];

}

}

// Matrix addition (C = A + B)

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

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

C[i][j] = A[i][j] + B[i][j];

}

}

// Output the result matrix C

cout << "\nThe result of Matrix A + Matrix B is:" << endl;

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

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

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

}

cout << endl;

}

return 0;

}

o Objective: Practice multi-dimensional arrays.

Ans. In C++, a **multi-dimensional array** is an array of arrays. The most common type is a **2D array** (which you have already seen), but you can also work with **3D** or even higher-dimensional arrays.

3. String Palindrome Check

Write a C++ program to check if a given string is a palindrome (reads the same forwards and backwards).

Ans. Here's a **C++ program** that checks if a given string is a **palindrome** (reads the same forwards and backwards).

**Palindrome Check:**

A **palindrome** is a word, phrase, number, or other sequence of characters that reads the same forward as backward (ignoring spaces, punctuation, and case).

For example:

* "madam" is a palindrome.
* "racecar" is a palindrome.
* "hello" is **not** a palindrome.

**C++ Program: Palindrome Check**

cpp

CopyEdit

#include <iostream>

#include <string>

#include <algorithm> // For transform() and reverse()

using namespace std;

bool isPalindrome(string str) {

// Convert string to lowercase for case-insensitive comparison

transform(str.begin(), str.end(), str.begin(), ::tolower);

// Remove spaces or non-alphanumeric characters (optional)

str.erase(remove\_if(str.begin(), str.end(), ::isspace), str.end());

// Check if string is equal to its reverse

string reversedStr = str;

reverse(reversedStr.begin(), reversedStr.end());

return str == reversedStr;

}

int main() {

string str;

cout << "Enter a string: ";

getline(cin, str); // Read full line including spaces

if (isPalindrome(str)) {

cout << "The string is a palindrome." << endl;

} else {

cout << "The string is not a palindrome." << endl;

}

return 0;

}

**Explanation:**

1. **Input String**: The program uses getline() to capture the entire input string, including spaces.
2. **String Processing**:
   * The string is **converted to lowercase** using transform() to make the palindrome check case-insensitive.
   * **Spaces** are removed using erase() and remove\_if() (optional for cleaner palindromes).
3. **Palindrome Check**:
   * We reverse the string using reverse() and check if the original string is equal to its reversed version.
4. **Output**: The program outputs whether the string is a palindrome.

**Example Output:**

less

CopyEdit

Enter a string: A man a plan a canal Panama

The string is a palindrome.

**Key Takeaways:**

* **Palindrome** check is case-insensitive and space-insensitive in this program.
* We used **transform()** to convert to lowercase and **reverse()** to reverse the string.
* **erase()** and **remove\_if()** are used to remove spaces (optional).

o Objective: Practice string operations.

Ans. Great! Practicing **string operations** is a crucial skill in C++ as strings are frequently used for data manipulation, text processing, and many other applications.

In the palindrome check program we just discussed, we’ve already touched on some key string operations like:

* **Reading input** with getline().
* **Converting case** with transform().
* **Reversing a string** using reverse().
* **Removing characters** using erase() and remove\_if().

Now, let's expand on **other common string operations** and their practical uses in C++.

THEORY EXERCISE:

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

Ans. An **array** in C++ is a collection of elements of the **same data type** that are stored in **contiguous memory locations**. Arrays are used to store multiple values in a single variable, instead of declaring individual variables for each value.

### ****Declaration of Arrays:****

cpp

CopyEdit

type arrayName[arraySize];

* **type**: Specifies the data type of the elements (e.g., int, float).
* **arrayName**: The name of the array.
* **arraySize**: The number of elements the array can hold.

### ****Example (Single-Dimensional Array)****:

cpp

CopyEdit

int arr[5]; // Declares an array of 5 integers

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

1. **Single-Dimensional Arrays (1D Arrays)**
2. **Multi-Dimensional Arrays (2D, 3D, etc.)**

## ****1. Single-Dimensional Arrays (1D Arrays)****

A **single-dimensional array** is like a list or a vector, where all the elements are arranged in a single row or column.

### ****Declaration:****

cpp

CopyEdit

type arr[arraySize];

**Example**:

cpp

CopyEdit

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

### ****Accessing Elements in 1D Array****:

You access individual elements using **indices**. In a 1D array, the index starts at 0.

cpp

CopyEdit

cout << arr[0]; // Accesses the first element (1)

cout << arr[4]; // Accesses the fifth element (5)

### ****Example of Using 1D Array****:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int arr[5] = {1, 2, 3, 4, 5}; // Declaring a 1D array

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

cout << arr[i] << " "; // Output: 1 2 3 4 5

}

return 0;

}

## ****2. Multi-Dimensional Arrays****

A **multi-dimensional array** is an array of arrays. It is used when you need to store data in a **grid-like structure** (e.g., a matrix).

### ****Types of Multi-Dimensional Arrays:****

1. **2D Arrays** (Matrix)
2. **3D Arrays**
3. Arrays of higher dimensions (nD arrays)

### ****2D Arrays (Matrix)****:

A **2D array** is a collection of 1D arrays, where data is organized in **rows and columns** (like a matrix).

### ****Declaration of 2D Array****:

cpp

CopyEdit

type arrayName[rows][columns];

**Example**:

cpp

CopyEdit

int matrix[3][3]; // A 2D array with 3 rows and 3 columns

### ****Accessing Elements in 2D Array****:

You access elements using two indices: the first for the row and the second for the column.

cpp

CopyEdit

matrix[0][0] = 1; // First element in first row

matrix[2][1] = 5; // Second element in third row

### ****Example of Using 2D Array****:

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Declaring and initializing a 2D array (3x3 matrix)

int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

// Outputting elements of the 2D array

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

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

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

}

cout << endl; // Output each row in a new line

}

return 0;

}

**Output:**

CopyEdit

1 2 3

4 5 6

7 8 9

### ****3D Arrays**** (and beyond):

A **3D array** adds a third dimension to the array, often used to represent cubes or other complex data structures.

### ****Declaration of 3D Array****:

cpp

CopyEdit

type arrayName[depth][rows][columns];

**Example**:

cpp

CopyEdit

int arr[2][3][4]; // A 3D array with 2 blocks, 3 rows, and 4 columns

## ****Difference Between Single-Dimensional and Multi-Dimensional Arrays****

| **Aspect** | **Single-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| **Structure** | A simple list of elements. | A collection of arrays (e.g., a matrix or grid). |
| **Indexing** | Uses one index (0 to n-1). | Uses multiple indices (e.g., two indices for 2D arrays). |
| **Example** | int arr[5] = {1, 2, 3, 4, 5}; | int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; |
| **Memory Layout** | Contiguous block of memory. | Block of memory that represents arrays within arrays. |
| **Use Cases** | Storing linear data. | Representing grids, tables, matrices, and higher-dimensional data. |

## ****Key Takeaways****:

* **Single-dimensional arrays** are used to store a series of elements in a single line (e.g., a list).
* **Multi-dimensional arrays** store data in a grid-like structure, making them useful for operations on matrices and more complex data arrangements.
* **2D arrays** are often used to represent matrices, while **3D arrays** are used for data that requires a third dimension (e.g., for representing a cube).

2. Explain string handling in C++ with examples.

Ans. In C++, **string handling** refers to the manipulation of sequences of characters, which is a core aspect of many programming tasks. C++ provides two primary ways to work with strings:

1. **C-style Strings** (Null-terminated character arrays)
2. **C++ string Class** (from the Standard Library)

We'll cover both approaches, but C++ string is generally easier and more efficient, so we’ll focus more on that.

### ****1. C-style Strings****

C-style strings are arrays of characters terminated by a special **null character** ('\0'). Each element of the array represents one character of the string.

#### ****Example of C-style String:****

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// C-style string: Array of characters

char str[] = "Hello, World!";

// Printing the C-style string

cout << "C-style String: " << str << endl;

return 0;

}

#### ****Key Operations with C-style Strings****:

1. **Length of a string**: Use the strlen() function from <cstring> to get the length of a string (excluding the null character).

cpp

CopyEdit

int len = strlen(str); // Gives length of the string (not including the null terminator)

1. **Copying strings**: Use strcpy() to copy one C-style string to another.

cpp

CopyEdit

char str2[20];

strcpy(str2, str); // Copies contents of str to str2

1. **Concatenating strings**: Use strcat() to concatenate (combine) two C-style strings.

cpp

CopyEdit

char str3[40];

strcpy(str3, "Hello, ");

strcat(str3, "World!"); // Concatenates "World!" to str3

cout << str3; // Output: "Hello, World!"

### ****2. C++**** string ****Class****

The **C++ string class** (found in the <string> header) provides a more **powerful and flexible** way to handle strings than C-style strings. It automatically manages memory and provides a wide range of functions for manipulating text.

#### ****Basic Operations with C++**** string ****Class****:

1. **Creating and Initializing Strings**:

cpp

CopyEdit

string str1 = "Hello, World!"; // Initialize with a string literal

string str2 = str1; // Copy initialization

string str3(5, 'A'); // Create string of length 5 with 'A' repeated: "AAAAA"

1. **Accessing Characters**:
   * You can access individual characters using [] or .at().

cpp

CopyEdit

cout << str1[0]; // Output: H

cout << str1.at(1); // Output: e

1. **Length of a String**:
   * Use the .length() or .size() function to get the length of the string.

cpp

CopyEdit

cout << "Length of str1: " << str1.length() << endl; // Output: 13

1. **Concatenating Strings**:
   * You can concatenate strings using the + operator or .append() method.

cpp

CopyEdit

string str4 = "Hello";

str4 += ", World!"; // Concatenate using `+=`

cout << str4; // Output: "Hello, World!"

1. **Finding Substrings**:
   * Use .find() to search for a substring within a string.

cpp

CopyEdit

size\_t pos = str1.find("World");

if (pos != string::npos) {

cout << "'World' found at position: " << pos << endl;

} else {

cout << "'World' not found" << endl;

}

1. **Replacing Substrings**:
   * Use .replace() to replace part of the string.

cpp

CopyEdit

str1.replace(7, 5, "C++"); // Replaces "World" with "C++"

cout << str1; // Output: "Hello, C++!"

1. **Substring Extraction**:
   * Use .substr() to extract a substring from a string.

cpp

CopyEdit

string sub = str1.substr(7, 3); // Extracts 3 characters starting from index 7

cout << sub; // Output: "C++"

1. **Clearing a String**:
   * Use .clear() to empty the contents of a string.

cpp

CopyEdit

str1.clear();

cout << "str1 after clear: " << str1 << endl; // Output: ""

1. **String Comparison**:
   * You can compare strings using ==, !=, <, >, etc.

cpp

CopyEdit

if (str1 == str2) {

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

} else {

cout << "Strings are not equal" << endl;

}

### ****Example of C++ String Handling:****

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

// Creating and initializing strings

string str1 = "Hello, ";

string str2 = "World!";

// Concatenation

string result = str1 + str2; // Using + operator for concatenation

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

// Length of string

cout << "Length of str1: " << str1.length() << endl;

// Substring extraction

string sub = result.substr(7, 5); // Extracts substring "World"

cout << "Extracted Substring: " << sub << endl;

// Finding a substring

size\_t pos = result.find("World");

if (pos != string::npos) {

cout << "'World' found at index: " << pos << endl;

}

// Replacing a part of the string

result.replace(7, 5, "C++"); // Replace "World" with "C++"

cout << "After Replace: " << result << endl;

// Comparing strings

if (str1 == "Hello, ") {

cout << "str1 is equal to 'Hello, '" << endl;

}

// Clearing the string

str1.clear();

cout << "After Clear: " << str1 << endl; // Output: ""

return 0;

}

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

mathematica

CopyEdit

Concatenated String: Hello, World!

Length of str1: 7

Extracted Substring: World

'World' found at index: 7

After Replace: Hello, C++!

str1 is equal to 'Hello, '

After Clear:

### ****Summary of String Handling in C++:****

* **C-style strings** are arrays of characters, often used in older C programs, requiring manual memory management and null-termination.
* **C++ string class** is safer and more flexible, offering built-in functions for string manipulation like concatenation, substring extraction, replacement, and comparison.
* Common operations on strings include **length**, **concatenation**, **substrings**, **finding** substrings, **replacing**, and **comparing**.

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

Ans. In C++, arrays can be initialized in several ways. Initialization refers to assigning values to the elements of an array at the time of its creation. The method you use depends on the type of array (single-dimensional or multi-dimensional) and whether you're initializing the array with specific values.

### ****1. Single-Dimensional (1D) Array Initialization****

A **1D array** is a simple collection of elements stored in contiguous memory locations. Here's how you can initialize a single-dimensional array:

#### ****1.1. Static Initialization (Direct Initialization)****

You can directly initialize a 1D array by specifying values at the time of declaration.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Static initialization with predefined values

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

// Displaying array elements

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

cout << arr[i] << " "; // Output: 1 2 3 4 5

}

cout << endl;

return 0;

}

In this case, the values {1, 2, 3, 4, 5} are assigned to the array arr.

#### ****1.2. Default Initialization (Partial Initialization)****

If you initialize a 1D array with fewer values than its size, the remaining elements are initialized to zero (for basic types like int, float, etc.).

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Default initialization with partial values

int arr[5] = {1, 2}; // The remaining 3 elements will be 0

// Displaying array elements

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

cout << arr[i] << " "; // Output: 1 2 0 0 0

}

cout << endl;

return 0;

}

#### ****1.3. Zero Initialization****

You can initialize all elements of a 1D array to zero by using {} without specifying any values.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Zero initialization of all elements

int arr[5] = {}; // All elements will be set to 0

// Displaying array elements

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

cout << arr[i] << " "; // Output: 0 0 0 0 0

}

cout << endl;

return 0;

}

### ****2. Multi-Dimensional (2D) Array Initialization****

A **2D array** is an array of arrays (think of a table or matrix). Here's how you can initialize a two-dimensional array:

#### ****2.1. Static Initialization****

You can initialize a 2D array by providing values for each row and column at the time of declaration.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Static initialization of a 2D array (3x3 matrix)

int matrix[3][3] = {

{1, 2, 3}, // Row 1

{4, 5, 6}, // Row 2

{7, 8, 9} // Row 3

};

// Displaying elements of the 2D array

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

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

cout << matrix[i][j] << " "; // Output: 1 2 3 4 5 6 7 8 9

}

cout << endl;

}

return 0;

}

In this example, we initialize a 2D array with 3 rows and 3 columns. Each row has three integers, and we populate all elements of the 2D array.

#### ****2.2. Partial Initialization****

If you don't provide values for all elements, the remaining elements are automatically initialized to zero.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Partial initialization of a 2D array

int matrix[3][3] = {

{1, 2}, // First row with two elements, the third will be 0

{4}, // Second row with one element, the others will be 0

{} // Third row is empty, all elements will be 0

};

// Displaying elements of the 2D array

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

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

cout << matrix[i][j] << " "; // Output: 1 2 0 4 0 0 0 0 0

}

cout << endl;

}

return 0;

}

#### ****2.3. Zero Initialization****

Similar to 1D arrays, you can use {} to zero-initialize all elements of a 2D array.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

// Zero initialization of all elements in the 2D array

int matrix[3][3] = {}; // All elements will be set to 0

// Displaying elements of the 2D array

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

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

cout << matrix[i][j] << " "; // Output: 0 0 0 0 0 0 0 0 0

}

cout << endl;

}

return 0;

}

### ****3. Using Loops for Initialization****

Sometimes, arrays are initialized dynamically (during runtime) using loops. Here's an example of how to initialize a 1D and a 2D array using loops:

#### ****3.1. Initializing 1D Array with a Loop****

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int arr[5];

// Initialize the array using a loop

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

arr[i] = i + 1; // Assign values 1 to 5

}

// Displaying the array

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

cout << arr[i] << " "; // Output: 1 2 3 4 5

}

cout << endl;

return 0;

}

#### ****3.2. Initializing 2D Array with a Loop****

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int matrix[3][3];

// Initialize the 2D array using nested loops

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

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

matrix[i][j] = (i \* 3) + (j + 1); // Filling with values from 1 to 9

}

}

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

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

cout << matrix[i][j] << " "; // Output: 1 2 3 4 5 6 7 8 9

}

cout << endl;

}

return 0;

}

### ****Summary of Array Initialization in C++:****

* **1D Arrays** can be initialized with direct values, partial values (zero initialization), or a loop.
* **2D Arrays** (and higher-dimensional arrays) can be initialized similarly, with values for rows and columns, or with default zero values if no specific values are given.
* Arrays can also be initialized dynamically using loops, which is useful when the values are computed during runtime.

4. Explain string operations and functions in C++.

Ans. In C++, string handling can be done using two types of string representations:

1. **C-style strings** (arrays of characters)
2. **C++ string class** (from the C++ Standard Library)

The C++ string class offers a much more powerful and flexible way to manipulate strings compared to C-style strings, which require manual memory management.

Here, we'll cover the **C++ string class**, which is the most common approach for string operations. We'll also briefly mention some C-style string functions for completeness.

### ****C++**** string ****Class Operations****

The string class is defined in the <string> header and provides various functions for performing string operations like concatenation, comparison, searching, and more.

#### ****1. String Initialization and Assignment****

You can initialize and assign strings in several ways:

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

// Initializing strings

string str1 = "Hello, ";

string str2 = "World!";

// String assignment

string str3;

str3 = str1 + str2; // Concatenates str1 and str2

cout << str3 << endl; // Output: Hello, World!

return 0;

}

#### ****2. String Concatenation****

Strings can be concatenated using the + operator or the .append() method:

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str1 = "Hello, ";

string str2 = "World!";

// Using the + operator

string result = str1 + str2;

cout << result << endl; // Output: Hello, World!

// Using .append() method

str1.append(str2);

cout << str1 << endl; // Output: Hello, World!

return 0;

}

#### ****3. String Length****

The .length() or .size() function returns the length of a string (the number of characters in the string):

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, World!";

cout << "Length of the string: " << str.length() << endl; // Output: 13

return 0;

}

#### ****4. Accessing Characters****

You can access individual characters of a string using the [] operator or the .at() function. The .at() function provides bounds checking, meaning it will throw an exception if you try to access an invalid index.

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello";

// Accessing characters using []

cout << "First character: " << str[0] << endl; // Output: H

// Accessing characters using .at()

cout << "Second character: " << str.at(1) << endl; // Output: e

return 0;

}

#### ****5. String Comparison****

You can compare two strings using relational operators or the .compare() function. The relational operators return true or false, while .compare() returns an integer value:

* **< 0**: str1 is less than str2
* **0**: str1 is equal to str2
* **> 0**: str1 is greater than str2

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str1 = "Hello";

string str2 = "World";

// Using relational operators

if (str1 == str2) {

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

} else {

cout << "Strings are not equal!" << endl; // Output: Strings are not equal!

}

// Using compare()

int result = str1.compare(str2);

if (result < 0) {

cout << "str1 is less than str2" << endl;

} else if (result == 0) {

cout << "str1 is equal to str2" << endl;

} else {

cout << "str1 is greater than str2" << endl;

}

return 0;

}

#### ****6. Finding Substrings****

You can find a substring within a string using the .find() function. It returns the index of the first occurrence of the substring or string::npos if the substring is not found.

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, World!";

// Finding a substring

size\_t pos = str.find("World");

if (pos != string::npos) {

cout << "'World' found at position: " << pos << endl; // Output: 7

} else {

cout << "'World' not found!" << endl;

}

return 0;

}

#### ****7. Substring Extraction****

You can extract a substring from a string using the .substr() function. This function takes two arguments:

1. The starting position (index).
2. The length of the substring to extract (optional).

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, World!";

// Extracting a substring

string sub = str.substr(7, 5); // Starting at index 7, extract 5 characters

cout << "Extracted substring: " << sub << endl; // Output: World

return 0;

}

#### ****8. String Insertion****

You can insert a string at any position using the .insert() function. This function takes two arguments: the position where you want to insert the string and the string you want to insert.

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, !";

// Inserting a string at a specific position

str.insert(7, "World");

cout << "String after insertion: " << str << endl; // Output: Hello, World!

return 0;

}

#### ****9. String Replacing****

You can replace a part of a string using the .replace() function, which takes three arguments:

1. The starting position.
2. The number of characters to replace.
3. The string to insert.

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, World!";

// Replacing part of the string

str.replace(7, 5, "C++");

cout << "String after replace: " << str << endl; // Output: Hello, C++!

return 0;

}

#### ****10. String Clearing****

You can clear the contents of a string using the .clear() function, which removes all characters from the string.

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, World!";

// Clearing the string

str.clear();

cout << "String after clear: " << str << endl; // Output: (empty string)

return 0;

}

### ****C-style String Functions****

Though C++ string class is more powerful, it is still useful to know how C-style strings are handled. Here are some functions for C-style strings (using the <cstring> header):

1. **strlen()**: Returns the length of a C-style string (excluding the null-terminator).

cpp

CopyEdit

#include <cstring>

size\_t len = strlen("Hello");

1. **strcpy()**: Copies one C-style string to another.

cpp

CopyEdit

char str1[10];

strcpy(str1, "Hello");

1. **strcat()**: Concatenates two C-style strings.

cpp

CopyEdit

char str1[20] = "Hello, ";

strcat(str1, "World!");

1. **strcmp()**: Compares two C-style strings.

cpp

CopyEdit

int result = strcmp("Hello", "World");

### ****Summary of String Operations in C++:****

* **String Initialization**: You can initialize strings directly or through assignment.
* **Concatenation**: Use + or .append() for combining strings.
* **Length**: Use .length() or .size() to get the number of characters.
* **Character Access**: Use [] or .at() to access individual characters.
* **Comparison**: Use ==, <, >, or .compare() to compare strings.
* **Substring Operations**: Use .find(), .substr(), and .replace() for substring operations.
* **Modification**: Insert or replace parts of strings using .insert() and .replace().
* **Clearing**: Clear the string with .clear().

The **C++ string class** provides a more modern and flexible way to handle string operations compared to traditional C-style strings. Most modern C++ code uses the string class due to its ease of use and memory management capabilities.

6. Introduction to Object-Oriented Programming

LAB EXERCISES:

1. Class for a Simple Calculator

Write a C++ program that defines a class Calculator with functions for addition, subtraction, multiplication, and division. Create objects to use these functions.

Ans. #include <iostream>

using namespace std;

// Calculator class definition

class Calculator {

public:

// Function for addition

double add(double a, double b) {

return a + b;

}

// Function for subtraction

double subtract(double a, double b) {

return a - b;

}

// Function for multiplication

double multiply(double a, double b) {

return a \* b;

}

// Function for division

double divide(double a, double b) {

if (b != 0) {

return a / b; // Division is safe if b is not zero

} else {

cout << "Error: Division by zero!" << endl;

return 0; // Return 0 if division by zero occurs

}

}

};

int main() {

// Create an object of the Calculator class

Calculator calc;

double num1, num2;

char op;

cout << "Enter first number: ";

cin >> num1;

cout << "Enter second number: ";

cin >> num2;

cout << "Enter operation (+, -, \*, /): ";

cin >> op;

// Perform the chosen operation

switch (op) {

case '+':

cout << "Result: " << calc.add(num1, num2) << endl;

break;

case '-':

cout << "Result: " << calc.subtract(num1, num2) << endl;

break;

case '\*':

cout << "Result: " << calc.multiply(num1, num2) << endl;

break;

case '/':

cout << "Result: " << calc.divide(num1, num2) << endl;

break;

default:

cout << "Invalid operation!" << endl;

break;

}

return 0;

}

o Objective: Introduce basic class structure.

Ans. The objective of this program is to introduce the basic structure of a class in C++ by defining a Calculator class that encapsulates various functions (addition, subtraction, multiplication, and division). This example demonstrates:

* **Class Declaration:** How to define a class with member functions.
* **Object Creation:** How to create an object of the class and use its functions.
* **Member Functions:** How to define functions inside a class to perform specific tasks.
* **Encapsulation:** How the class encapsulates related functionalities together, making the code modular and easier to manage.

2. Class for Bank Account

Create a class BankAccount with data members like balance and member functions like deposit and withdraw. Implement encapsulation by keeping the data members private

Ans. #include <iostream>

using namespace std;

// BankAccount class definition with encapsulation

class BankAccount {

private:

double balance; // Private data member for balance

public:

// Constructor to initialize the balance

BankAccount(double initialBalance) {

if (initialBalance >= 0) {

balance = initialBalance; // Set balance to the provided initial value

} else {

balance = 0; // Ensure balance is never negative

cout << "Initial balance can't be negative. Setting balance to 0." << endl;

}

}

// Member function to deposit an amount

void deposit(double amount) {

if (amount > 0) {

balance += amount; // Add the deposit amount to balance

cout << "Deposited: $" << amount << endl;

} else {

cout << "Deposit amount must be positive!" << endl;

}

}

// Member function to withdraw an amount

void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount; // Subtract the withdrawal amount from balance

cout << "Withdrew: $" << amount << endl;

} else if (amount <= 0) {

cout << "Withdrawal amount must be positive!" << endl;

} else {

cout << "Insufficient balance for withdrawal!" << endl;

}

}

// Member function to display the balance

void displayBalance() const {

cout << "Current balance: $" << balance << endl;

}

};

int main() {

// Create a BankAccount object with an initial balance of 1000

BankAccount myAccount(1000.0);

// Display initial balance

myAccount.displayBalance();

// Deposit some money

myAccount.deposit(500.0);

myAccount.displayBalance();

// Attempt to withdraw some money

myAccount.withdraw(200.0);

myAccount.displayBalance();

// Attempt to withdraw more money than available

myAccount.withdraw(1500.0);

myAccount.displayBalance();

return 0;

}

o Objective: Understand encapsulation in classes.

Ans. The objective of the program is to **understand encapsulation in classes**. Encapsulation is one of the core principles of Object-Oriented Programming (OOP) that helps in bundling the data (attributes) and the methods (functions) that operate on the data into a single unit called a **class**. Additionally, encapsulation restricts direct access to certain components of an object, which is achieved by **hiding** the object's internal state and allowing access through **public methods** (also called getters and setters).

**Key Concepts of Encapsulation:**

1. **Private Data Members**:
   * In C++, we make certain data members of a class private to prevent direct access from outside the class. This ensures that the object's state cannot be modified arbitrarily, thus protecting the integrity of the data.
   * In the example BankAccount, the balance data member is **private**, meaning it cannot be accessed directly by code outside of the class.
2. **Public Member Functions (Methods)**:
   * Public methods are the interface through which we interact with an object. These methods provide controlled access to the private data members.
   * For example, methods like deposit, withdraw, and displayBalance in the BankAccount class are public and provide a way to safely interact with the balance.
3. **Access Control**:
   * **Private members** are not directly accessible from outside the class.
   * **Public members** can be accessed from outside the class.
4. **Getter and Setter Methods** (Optional in this example):
   * Sometimes, we use getter and setter methods to access or modify private data. This provides more control and validation when accessing or modifying the private data.
   * In this example, instead of direct access to balance, we used functions (deposit and withdraw) to change its value, ensuring validation is done (e.g., for deposit/withdraw amounts).

**Why Encapsulation is Important:**

* **Data Integrity:** By controlling how the data is accessed and modified, encapsulation ensures that objects remain in a valid state. For example, it prevents an account balance from being directly set to an invalid value like a negative number (unless through a method that checks such conditions).
* **Ease of Maintenance:** Encapsulation hides the internal implementation of an object. This means that the implementation can change without affecting the code that uses the object, making it easier to maintain and update the code.
* **Code Reusability and Flexibility:** Since the internal details are hidden and only necessary functionality is exposed, the class can be reused in various contexts without worrying about its internal workings.

**Summary:**

* **Encapsulation** in C++ helps in hiding the internal state of an object (private data members) and only exposes methods (public functions) to interact with the object. This prevents direct modification of the internal state and ensures the object remains in a valid and consistent state.

The BankAccount class example demonstrates this by making the balance a private data member and allowing deposits and withdrawals only through public methods that handle validation.

This practice of hiding implementation details and exposing only essential functionality is fundamental to building robust, maintainable, and reusable object-oriented code.

3. Inheritance Example

Write a program that implements inheritance using a base class Person and derived classes Student and Teacher. Demonstrate reusability through inheritance.

Ans. #include <iostream>

#include <string>

using namespace std;

// Base class: Person

class Person {

protected:

string name;

int age;

public:

Person(string name, int age) {

this->name = name;

this->age = age;

}

void displayInfo() {

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

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

}

};

class Student : public Person {

private:

string studentID;

public:

// Constructor to initialize student's data, using the base class constructor for name and age

Student(string name, int age, string studentID) : Person(name, age) {

this->studentID = studentID;

}

void displayStudentInfo() {

displayInfo(); // Reusing displayInfo from Person class

cout << "Student ID: " << studentID << endl;

}

};

class Teacher : public Person {

private:

string subject;

public:

// Constructor to initialize teacher's data, using the base class constructor for name and age

Teacher(string name, int age, string subject) : Person(name, age) {

this->subject = subject;

}

// Member function to display teacher-specific information

void displayTeacherInfo() {

displayInfo(); // Reusing displayInfo from Person class

cout << "Subject: " << subject << endl;

}

};

int main() {

// Creating a Student object

Student student("John Doe", 20, "S12345");

cout << "Student Information:" << endl;

student.displayStudentInfo(); // Display student info

cout << endl;

// Creating a Teacher object

Teacher teacher("Jane Smith", 40, "Mathematics");

cout << "Teacher Information:" << endl;

teacher.displayTeacherInfo(); // Display teacher info

return 0;

}

o Objective: Learn the concept of inheritance.

Ans. The objective of the program is to **learn the concept of inheritance** in C++. Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called the **derived class**) to inherit properties and behaviors (data members and member functions) from another class (called the **base class**).

**Key Concepts of Inheritance:**

1. **Base Class**:
   * The **base class** (also known as the **parent class** or **superclass**) is the class that contains common attributes and behaviors (methods) that can be shared by multiple derived classes.
   * In the example program, the Person class is the base class. It has common data members (name and age) and a method displayInfo() that can be inherited by any class that derives from Person.
2. **Derived Class**:
   * A **derived class** (also known as a **child class** or **subclass**) is a class that inherits properties and behaviors from a base class. It can have additional data members and member functions specific to it.
   * In the example program, both Student and Teacher are derived classes that inherit from the Person base class.
   * The derived class can:
     + **Extend** the functionality of the base class by adding new attributes and methods.
     + **Override** base class methods if needed (not shown in this example, but possible).
     + **Reuse** methods from the base class, as demonstrated with displayInfo().
3. **Constructor Inheritance**:
   * The constructor of the base class is inherited by the derived class, and the derived class can call the base class constructor to initialize the inherited attributes.
   * In the program, both Student and Teacher constructors call the Person constructor to initialize name and age.
4. **Reusability**:
   * One of the key advantages of inheritance is **reusability**. Inheritance allows the derived class to reuse the code from the base class, reducing code duplication.
   * In the example, both Student and Teacher reuse the displayInfo() method from the Person class, which displays common information like name and age.

**Benefits of Inheritance:**

1. **Code Reusability**:
   * With inheritance, common functionality is written only once in the base class and reused in all derived classes. This eliminates code duplication and keeps the code DRY (Don't Repeat Yourself).
2. **Modularity**:
   * By organizing related classes into a hierarchy (base and derived), the program becomes more modular. Each class has a clear role, and this modularity improves maintainability.
3. **Extensibility**:
   * Inheritance allows you to easily extend the functionality of a class by creating new derived classes. For example, you can create additional derived classes (e.g., Principal, Staff) based on the Person class without modifying the existing Student or Teacher classes.
4. **Flexibility**:
   * With inheritance, you can modify or extend the base class while minimizing the changes required in the derived classes. This makes it easier to update code as requirements change.

**Types of Inheritance in C++:**

* **Single Inheritance**: A derived class inherits from only one base class (as in our example with Student and Teacher).
* **Multiple Inheritance**: A derived class can inherit from more than one base class (not demonstrated here but possible).
* **Multilevel Inheritance**: A derived class can inherit from another derived class, forming a chain (not demonstrated here but possible).
* **Hierarchical Inheritance**: Multiple derived classes inherit from the same base class (as in our example with Student and Teacher both inheriting from Person).

THEORY EXERCISE:

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

Ans. Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects** rather than functions and logic. The primary goal of OOP is to increase the modularity, reusability, and maintainability of software.

Here are the key concepts of OOP:

**1. Classes and Objects**

* **Class**: A class is a blueprint or template for creating objects. It defines the attributes (data members) and behaviors (methods or functions) that objects created from the class will have.
* **Object**: An object is an instance of a class. When a class is defined, no memory is allocated. Memory is allocated only when an object of that class is created. Objects have specific values for their attributes and can perform behaviors defined in the class.

**Example**:

cpp

CopyEdit

class Car {

public:

string color;

int speed;

void drive() {

cout << "Driving at speed " << speed << endl;

}

};

Car myCar; // Object of class Car

myCar.color = "Red";

myCar.speed = 100;

myCar.drive();

**2. Encapsulation**

* **Encapsulation** is the concept of bundling the data (attributes) and the methods (functions) that operate on the data within a single unit, the **class**. This hides the internal state of an object from the outside world and allows interaction only through well-defined methods.
* It also involves **data hiding**, where sensitive data is protected from outside access. This can be achieved using **private** and **protected** access modifiers.

**Example**:

cpp

CopyEdit

class Account {

private:

double balance; // Private data member

public:

void setBalance(double b) { balance = b; } // Setter method

double getBalance() { return balance; } // Getter method

};

In this example, the balance attribute is hidden from direct access, and can only be modified or retrieved using the setBalance and getBalance methods.

**3. Inheritance**

* **Inheritance** is a mechanism where one class (the derived class) inherits the attributes and behaviors (methods) of another class (the base class). This allows for **code reuse** and the creation of a class hierarchy.
* The derived class can extend the functionality of the base class by adding new attributes and methods or modifying existing methods.

**Example**:

cpp

CopyEdit

class Animal {

public:

void eat() {

cout << "Eating food" << endl;

}

};

class Dog : public Animal { // Dog inherits from Animal

public:

void bark() {

cout << "Barking!" << endl;

}

};

Dog myDog;

myDog.eat(); // Inherited from Animal

myDog.bark(); // Defined in Dog

In this example, the Dog class inherits from the Animal class, so Dog objects can call the eat() method, even though it is defined in the Animal class.

**4. Polymorphism**

* **Polymorphism** allows objects of different classes to be treated as objects of a common base class. The most common use of polymorphism is when a base class reference or pointer is used to refer to objects of derived classes.
* **Method Overloading** and **Method Overriding** are two types of polymorphism:
  + **Method Overloading**: Multiple functions can have the same name but different parameters.
  + **Method Overriding**: A derived class can provide a specific implementation of a method that is already defined in the base class.

**Example** (Method Overriding):

cpp

CopyEdit

class Animal {

public:

virtual void sound() {

cout << "Animal sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Barking!" << endl;

}

};

Animal\* animal = new Dog();

animal->sound(); // Calls Dog's sound() due to polymorphism

In this example, even though animal is a pointer to Animal, it will call the sound() method from Dog because of **method overriding**.

**5. Abstraction**

* **Abstraction** involves simplifying complex systems by exposing only the essential features while hiding the unnecessary details. It is achieved by defining **abstract classes** and **interfaces** in some languages, which provide a blueprint for other classes.
* **Abstract classes** cannot be instantiated and may contain **abstract methods** (methods that are declared but not defined).

**Example**:

cpp

CopyEdit

class Shape {

public:

virtual void draw() = 0; // Abstract method

};

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing Circle" << endl;

}

};

In this example, the Shape class is abstract because it contains a pure virtual function draw(). The Circle class provides an implementation for this method.

**Summary of Key OOP Concepts:**

* **Classes and Objects**: Defines the blueprint for creating objects, which are instances of classes.
* **Encapsulation**: Bundles data and methods inside classes, hiding internal state and ensuring controlled access.
* **Inheritance**: Allows derived classes to inherit from base classes, enabling code reuse and the creation of class hierarchies.
* **Polymorphism**: Allows different classes to provide different implementations of the same method, enhancing flexibility and reusability.
* **Abstraction**: Hides complex implementation details and only exposes the essential parts of an object or system.

**Benefits of OOP:**

* **Modularity**: The code is divided into classes, making it easier to manage and understand.
* **Code Reusability**: Inheritance allows the reuse of code across classes.
* **Maintainability**: OOP encourages organized code with better structure, making it easier to maintain and modify.
* **Flexibility and Extensibility**: Polymorphism and inheritance make it easier to extend and modify existing code without affecting other parts of the system.

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

Ans. A **class** is a user-defined blueprint or template for creating objects. It defines the attributes (data members) and methods (functions) that the objects created from the class will have. A class encapsulates data and functions into a single unit and specifies the properties and behaviors of objects that belong to that class.

A class does not allocate memory for its attributes until an object is created.

#### ****Syntax for Defining a Class:****

cpp

CopyEdit

class ClassName {

public:

// Data Members (Attributes)

type member1;

type member2;

// Member Functions (Methods)

void function1() {

// Function definition

}

// Constructor (Optional)

ClassName() {

// Constructor definition

}

};

* **Data Members**: Variables that hold the data for an object.
* **Member Functions**: Functions that define the behaviors or actions for objects of the class.
* **Constructor**: A special function that is called when an object of the class is created.

### ****2. Object in C++****

An **object** is an instance of a class. When a class is defined, no memory is allocated. Memory is allocated only when an object is created based on the class definition. Each object has its own copy of the data members defined in the class and can access the class's member functions.

#### ****Syntax for Creating an Object:****

cpp

CopyEdit

ClassName objectName;

An object can be created either by declaring it directly or dynamically using new.

### ****Example:****

Let's create a simple class called Car that has attributes like brand and speed, and a method to display the car's details.

cpp

CopyEdit

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

// Data members (attributes)

string brand;

int speed;

// Member function (method)

void displayInfo() {

cout << "Car Brand: " << brand << endl;

cout << "Speed: " << speed << " km/h" << endl;

}

// Constructor to initialize the attributes

Car(string b, int s) {

brand = b;

speed = s;

}

};

int main() {

// Creating an object of class Car

Car myCar("Toyota", 120); // object creation with constructor

// Calling the member function to display car details

myCar.displayInfo(); // Output: Car Brand: Toyota, Speed: 120 km/h

return 0;

}

### ****Explanation of the Example:****

1. **Class Definition (Car)**:
   * **Data members**: The class Car has two attributes: brand (a string) and speed (an integer).
   * **Member function (displayInfo)**: This function is responsible for displaying the brand and speed of the car object.
   * **Constructor**: The constructor initializes the data members brand and speed when an object is created.
2. **Object Creation**:
   * In the main() function, we create an object myCar of class Car and initialize it with the brand "Toyota" and speed 120.
   * The constructor of Car is called automatically when myCar is created, and it initializes the brand and speed attributes of myCar.
3. **Function Call**:
   * We use myCar.displayInfo() to call the displayInfo() method of the Car class, which prints out the car's brand and speed.

### ****Key Points to Remember:****

* A **class** is a blueprint, while an **object** is an instance of that blueprint.
* A class defines the **attributes** (data members) and **behaviors** (methods) of the objects created from it.
* When you create an object, the constructor is called to initialize the object’s attributes.
* **Objects** interact with the class methods to perform actions based on their data.

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

yaml

CopyEdit

Car Brand: Toyota

Speed: 120 km/h

### ****Summary:****

* **Classes** in C++ provide a way to group related data and functions together.
* **Objects** are instances of these classes, which have their own unique set of attributes and behaviors.
* This combination of classes and objects forms the backbone of object-oriented programming in C++.

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

Ans. **Inheritance** is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (called the **derived class**) to acquire the properties and behaviors (data members and methods) of an existing class (called the **base class**). Inheritance allows for **code reuse**, meaning that the derived class can use the functionality of the base class and extend or modify it as needed.

There are several types of inheritance in C++:

* **Single Inheritance**: A derived class inherits from a single base class.
* **Multiple Inheritance**: A derived class inherits from more than one base class.
* **Multilevel Inheritance**: A class derives from another derived class.
* **Hierarchical Inheritance**: Multiple derived classes inherit from the same base class.

Inheritance allows us to create a hierarchy of classes and facilitates **reusability** and **extensibility**.

**Syntax of Inheritance in C++:**

cpp

CopyEdit

class BaseClass {

public:

// Base class members

};

class DerivedClass : public BaseClass {

public:

// Derived class members

};

* BaseClass is the class from which other classes inherit.
* DerivedClass is the class that inherits from BaseClass.
* The **public** keyword specifies the type of inheritance, meaning that public members of the base class will be accessible to the derived class.

**Example of Inheritance in C++**

Let's consider a **base class Animal** that contains common attributes and methods for animals, and a **derived class Dog** that inherits from Animal and adds additional behavior specific to dogs.

cpp

CopyEdit

#include <iostream>

using namespace std;

// Base Class

class Animal {

public:

// Data members (attributes)

string name;

int age;

// Member function

void eat() {

cout << name << " is eating." << endl;

}

// Constructor for the base class

Animal(string n, int a) {

name = n;

age = a;

}

};

// Derived Class

class Dog : public Animal {

public:

// Additional member function for Dog

void bark() {

cout << name << " is barking!" << endl;

}

Dog(string n, int a) : Animal(n, a) {

}

};

int main() {

// Creating an object of the derived class Dog

Dog myDog("Buddy", 3);

myDog.eat(); // Inherited from Animal class

myDog.bark(); // Defined in Dog class

return 0;

}

**Explanation of the Example:**

1. **Base Class (Animal)**:
   * The Animal class has two data members (name and age) and a member function eat() that prints out a message indicating the animal is eating.
   * The constructor of the Animal class initializes name and age when an Animal (or derived) object is created.
2. **Derived Class (Dog)**:
   * The Dog class is derived from the Animal class using the public keyword, meaning it inherits all the public members of Animal (such as name, age, and the eat() method).
   * The Dog class has an additional member function bark(), which prints out a message that the dog is barking.
   * The Dog class constructor calls the base class constructor using an initializer list (: Animal(n, a)) to initialize the name and age of the Dog object.
3. **Object Creation**:
   * In the main() function, an object myDog of the Dog class is created with the name "Buddy" and age 3.
   * The eat() method is called on myDog, which is inherited from the Animal class.
   * The bark() method is called on myDog, which is specific to the Dog class.

**Output:**

csharp

Buddy is eating.

Buddy is barking!

**Key Points About Inheritance:**

1. **Reusability**:
   * The Dog class reuses the functionality of the Animal class, specifically the eat() method and the name and age attributes, avoiding the need to duplicate this code.
2. **Extensibility**:
   * The Dog class can add new functionality, such as the bark() method, without changing the base class Animal. This makes the code more flexible and easier to extend.
3. **Access Modifiers**:
   * The access level of inherited members can be controlled using access modifiers (public, protected, and private):
     + **Public inheritance** (public keyword): Public members of the base class remain accessible in the derived class.
     + **Private inheritance**: Public and protected members of the base class become private members of the derived class.
     + **Protected inheritance**: Public members of the base class become protected members of the derived class.
4. **Constructor and Destructor**:
   * The constructor of the derived class can call the constructor of the base class to initialize inherited data members.
   * The base class constructor is called before the derived class constructor, and the base class destructor is called after the derived class destructor.

**Types of Inheritance:**

1. **Single Inheritance**: A derived class inherits from a single base class (as shown in the example above).
2. **Multiple Inheritance**: A derived class inherits from more than one base class.
   * This can be useful when you want a class to have characteristics of multiple classes.
3. **Multilevel Inheritance**: A class is derived from another derived class.
4. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.

**Summary:**

* **Inheritance** allows one class to acquire the properties and methods of another class, promoting **code reuse** and **extensibility**.
* **Derived classes** can reuse, modify, or extend the functionality of **base classes**, and this enables creating a **class hierarchy**.
* In the example, the Dog class inherits from the Animal class, demonstrating **single inheritance**. The Dog class can use the inherited eat() method and also has its own bark() method.

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

Ans. **Encapsulation** is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a **class**. Encapsulation also involves **restricting access** to the internal state of an object and only allowing interaction through well-defined methods. This is often called **data hiding**, and it helps prevent unintended interference and misuse of the data.

In C++, encapsulation is achieved through the use of **access modifiers**, which control how the members of a class can be accessed.

**Key Aspects of Encapsulation:**

1. **Data Members (Attributes)**: The variables that hold the state or properties of an object.
2. **Member Functions (Methods)**: The functions that define the behaviors or actions that can be performed on the object's data.
3. **Access Modifiers**: These are keywords used to specify the accessibility of data members and member functions:
   * **public**: Members are accessible from outside the class.
   * **private**: Members are not accessible from outside the class; only accessible within the class's member functions.
   * **protected**: Members are not accessible from outside the class, but they are accessible to derived classes (in case of inheritance).