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

Procedural Programming (PP):--

* Focuses on a linear and structured approach.
* Programs are divided into procedures or functions, which operate on data.
* Data and functions are separate.
* Functions manipulate global or shared data, which may lead to tight coupling.
* Less emphasis on data protection.
* Reusability is achieved by creating reusable functions.
* Limited support for abstraction and reuse compared to OOP.
* Functions, loops, conditionals (e.g., for, while, if).

Object-Oriented Programming:----

* Focuses on objects that encapsulate data and behavior.
* Programs are structured around objects and classes
* Data is encapsulated within objects, promoting data security.
* Access to data is controlled through methods and access modifiers (e.g., public, private).
* Encourages loose coupling between components.
* Strong emphasis on reusability through inheritance, polymorphism, and abstraction.
* Promotes code modularity and reuse across projects.
* Classes, objects, methods, inheritance, polymorphism, encapsulation

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

**1. Modularity through Classes and Objects**

* **Explanation**:
  + In OOP, code is organized into classes and objects, making it easier to manage and understand.
  + Each class encapsulates related data (attributes) and behavior (methods), promoting modularity.
* **Advantage**:
  + Facilitates working on large projects by dividing them into smaller, manageable units.
  + Changes in one module (class) often do not affect others, reducing ripple effects.

**2. Code Reusability**

* **Explanation**:
  + OOP allows the reuse of code through **inheritance**, where a new class can inherit attributes and methods from an existing class.
* **Advantage**:
  + Saves development time and reduces code duplication.
  + Promotes the use of well-tested and proven components.

**3. Abstraction**

* **Explanation**:
  + OOP hides complex implementation details and exposes only essential features to the user through abstract classes and interfaces.
* **Advantage**:
  + Simplifies development and maintenance by focusing only on relevant details.
  + Reduces complexity for end users and developers.

**4. Encapsulation**

* **Explanation**:
  + Data and methods are bundled together within an object, and access to data is controlled using access modifiers like private, protected, and public.
* **Advantage**:
  + Protects data from unauthorized access and unintended interference.
  + Enhances data security and integrity.

**5. Polymorphism**

* **Explanation**:
  + OOP supports polymorphism, allowing objects to take on multiple forms. For example, a method can behave differently based on the object calling it (method overriding).
* **Advantage**:
  + Simplifies code and enhances flexibility by enabling generic programming.
  + Makes systems more adaptable to change.

**6. Real-World Modeling**

* **Explanation**:
  + OOP mimics real-world entities and their interactions using objects and classes.
* **Advantage**:
  + Makes it easier to design and implement complex systems.
  + Promotes intuitive problem-solving by modeling real-world relationships.

**7. Scalability**

* **Explanation**:
  + OOP supports scalable system design by breaking down applications into modular, reusable components.
* **Advantage**:
  + Simplifies adding new features or scaling the application without affecting existing code significantly.

**8. Maintainability**

* **Explanation**:
  + OOP encourages a structured approach with clear separation of concerns, reducing dependencies between different parts of the code.
* **Advantage**:
  + Easier to debug, update, and extend systems.
  + Reduces long-term costs of software development.

**9. Flexibility with Dynamic Binding**

* **Explanation**:
  + OOP uses dynamic (runtime) binding, which allows decisions about object behavior to be deferred until runtime.
* **Advantage**:
  + Enhances system flexibility and supports late changes without modifying core code.

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

**1. Choose Your Platform**

Decide on the platform where you'll be developing C++ code:

* **Windows**
* **macOS**
* **Linux**

**2. Install a Compiler**

C++ requires a compiler to translate code into machine-executable instructions.

**Common Compilers**

* **GCC (GNU Compiler Collection)**:
  + Available for Linux, macOS, and Windows (via MinGW or Cygwin).
* **MSVC (Microsoft C++ Compiler)**:
  + Comes with Microsoft Visual Studio (Windows).
* **Clang**:
  + Available for Linux, macOS, and Windows.

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

input/output operations are primarily handled using streams provided by the **<iostream>** library.

The two main types of operations are:

1. **Input Operations**: Reading data into a program.
2. **Output Operations**: Displaying or writing data from a program.
3. **Input Operations**

C++ uses std::cin to perform input operations. It reads data from the standard input (typically the keyboard).

**Examples of Input Operations**

**Single Input:**

#include <iostream>

using namespace std;

main() {

int num;

cout << "Enter a number: ";

cin >> num; // Input from the user

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

}

**Output Operations**

C++ uses std::cout to perform output operations. It writes data to the standard output (typically the console).

**Examples of Output Operations**

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl; // Outputs text and moves to the next line

}

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

**Integer Types (int)**

* Used to store whole numbers.

Ex:==

int age = 25;

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

**Floating-Point Types (float, double)**

* Used to store decimal numbers.

**float**: Typically 4 bytes, less precision.

**double**: Typically 8 bytes, more precision.

**Example:**

float pi = 3.14f;

double precisePi = 3.141592653589;

cout << "Pi (float): " << pi << ", Pi (double): " << precisePi << endl;

**c. Character Type (char)**

* Used to store a single character.

**Example:**

char grade = 'A';

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

**d. Boolean Type (bool)**

* Used to store true or false.

**Example:**

bool isPassed = true;

cout << "Passed: " << isPassed << endl; // Outputs: 1 (true)

**2. Derived Data Types**

**a. Array**

* Used to store multiple values of the same type.

**Example:**

int scores[3] = {90, 85, 88};

cout << "First score: " << scores[0] << endl;

**b. Pointer**

* Used to store the address of a variable.

int num = 42;

int\* ptr = &num;

cout << "Pointer Address: " << ptr << ", Value: " << \*ptr << endl;

**Structure (struct)**

* Groups variables of different types.

**Example:**

struct Student {

string name;

int age;

};

Student s = {"Alice", 20};

cout << "Name: " << s.name << ", Age: " << s.age << endl;

**Strings**

Though not a primitive type, std::string (from <string>) is widely used for text.

**Example:**

#include <string>

using namespace std;

string name = "Alice";

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

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

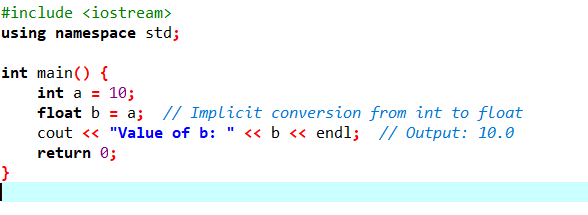
**. Implicit Type Conversion (Type Promotion or Coercion)**

Implicit type conversion occurs automatically by the compiler when there is no risk of data loss or when the conversion is logical and safe. It happens without the programmer’s intervention.

**Features:**

* Automatically performed by the compiler.
* Converts a smaller data type to a larger data type or a less precise type to a more precise type.
* Common in arithmetic expressions and assignments.

Example : Converting int to float

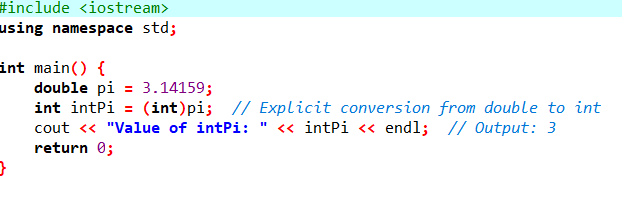


**Explicit Type Conversion (Type Casting)**

Explicit type conversion is performed manually by the programmer using cast operators. It is used when the programmer needs to override the compiler's default behavior, such as when a conversion may lead to data loss or when a specific type is required.

**Features:**

* Requires explicit syntax by the programmer.
* Done using either C-style casting, C++ casting operators, or function-style casting.
* Useful in scenarios where type mismatches occur.



| **Aspect** | **Implicit Conversion** | **Explicit Conversion** |
| --- | --- | --- |
| **Initiated by** | Automatically by the compiler. | Manually by the programmer. |
| **Risk of Data Loss** | Usually none (safe conversions only). | May result in data loss if not used carefully. |
| **Syntax** | No special syntax; happens automatically. | Requires cast operators or functions. |
| **Control** | Less control, determined by the compiler. | Full control over the conversion process. |
| **Use Cases** | Converting smaller types to larger types. | When the default behavior is inadequate or incorrect. |
| **Example** | int to float, float to double. | double to int, pointer type changes. |

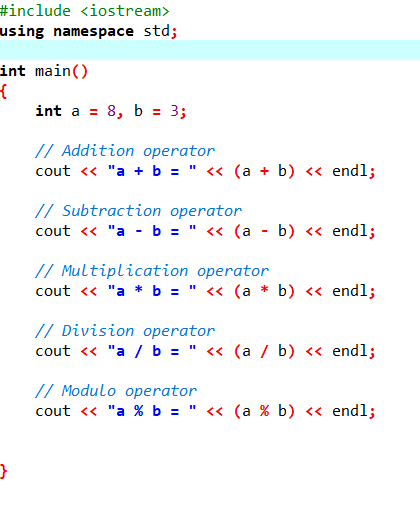
1. What are the different types of operators in C++? Provide examples of ?

**Operators in C++ can be classified into 6 types:**

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Ternary or Conditional Operators

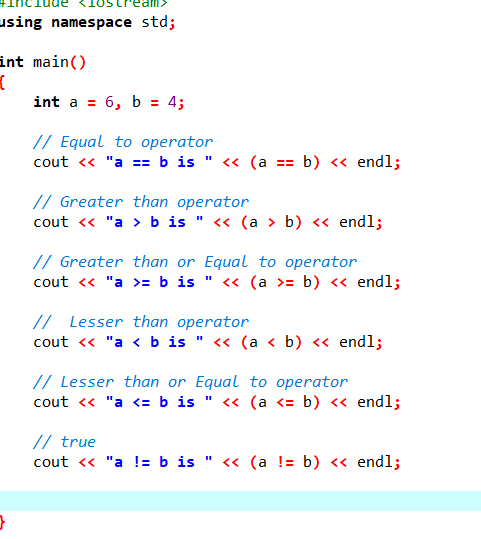
**1) Arithmetic Operators**

These operators are used to perform arithmetic or mathematical operations on the operands. For example, ‘+’ is used for addition, ‘-‘ is used for subtraction ‘\*’ is used for multiplication, etc.



**2) Relational Operators**

These operators are used for the comparison of the values of two operands. For example, ‘>’ checks if one operand is greater than the other operand or not, etc. The result returns a Boolean value, i.e.,**true** or **false.**



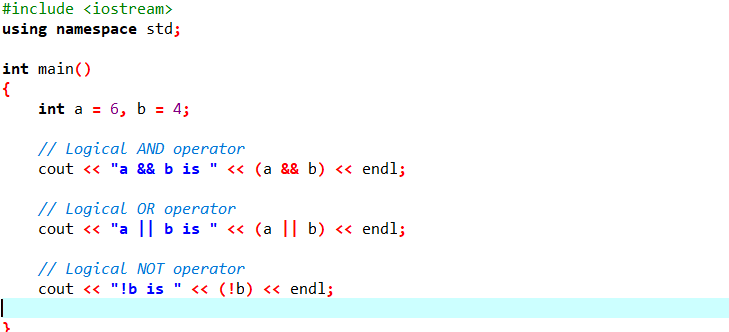
**3) Logical Operators**

These operators are used to combine two or more conditions or constraints or to complement the evaluation of the original condition in consideration. The result returns a Boolean value, i.e., **true** or **false**.

Logical AND && Returns true only if all the operands are true or non-zero.

Logical OR || Returns true if either of the operands is true or non-zero.

Logical NOT ! Returns true if the operand is false or zero.



**4) Bitwise Operators**

These operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc.

Binary AND & Copies a bit to the evaluated result if it exists in both operands

Binary OR | Copies a bit to the evaluated result if it exists in any of the opera

Binary XOR ^ Copies the bit to the evaluated result if it is present in either of

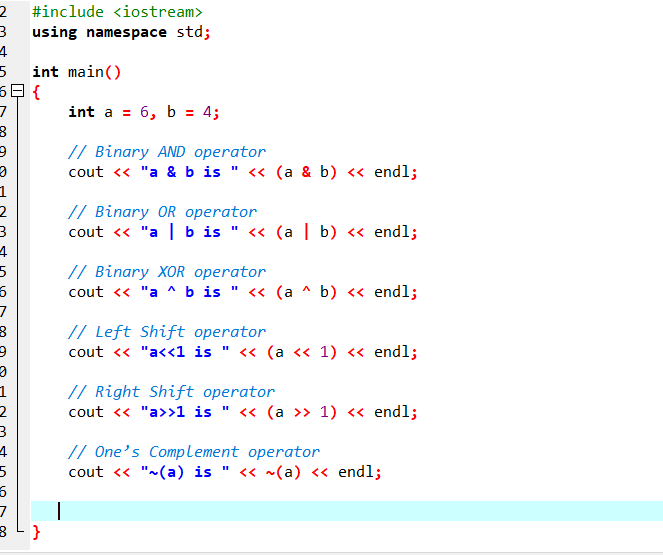
the operands but not bot

Left Shift << Shifts the value to left by the number of bits specified by the

right operand

Right Shift >> Shifts the value to right by the number of bits specified by the

right operand

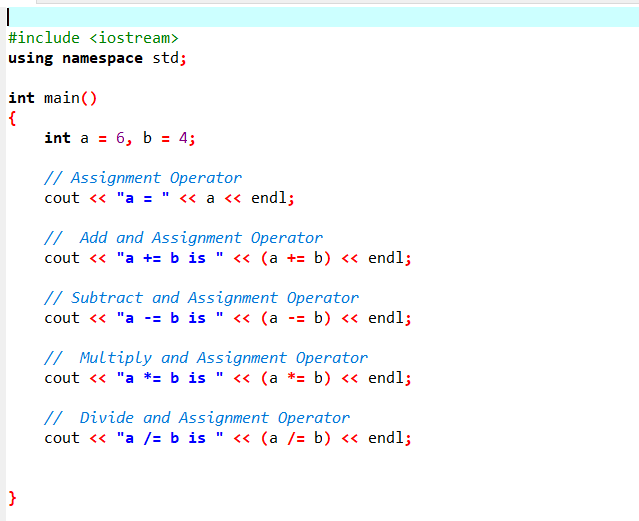


**5) Assignment Operators**

These operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

|  |  |  |
| --- | --- | --- |
| Assignment Operator | = | Assigns the value on the right to the variable on the left |

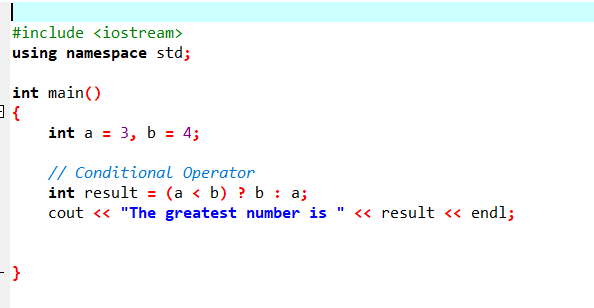
|  |  |  |
| --- | --- | --- |
| Add and Assignment Operator | += | First adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left |
| Subtract and Assignment Operator | -= | First subtracts the value on the right from the current value of the variable on left and then assign the result to the variable on the left |
| Multiply and Assignment Operator | \*= | First multiplies the current value of the variable on left to the value on the right and then assign the result to the variable on the left |
| Divide and Assignment Operator | /= | First divides the current value of the variable on left by the value on the right and then assign the result to the variable on the lef |



**6) Ternary or Conditional Operators(?:)**

This operator returns the value based on the condition.

The ternary operator **?** determines the answer on the basis of the evaluation of **Expression1**. If it is **true**, then **Expression2** gets evaluated and is used as the answer for the expression. If **Expression1** is **false**, then **Expression3** gets evaluated and is used as the answer for the expression.



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

**Constants**

Constants are fixed values that cannot be altered during the execution of a program. They are used to define values that remain unchanged throughout the program, improving readability and reducing errors.

const double PI = 3.14159;

cout << PI; // Output: 3.14159

// PI = 3.14; // Error: Assignment to const variable

**Literals**

Literals represent fixed values used directly in code. Unlike constants, literals don't require a name. They are directly written into the program and are the actual data.

**Purpose of Literals:**

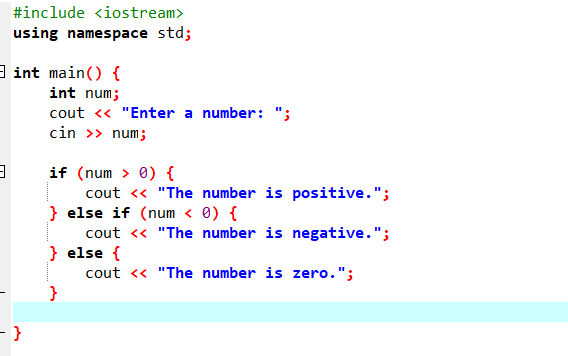
1. **Direct Representation:** Provide values directly in the code without needing variables.
2. **Efficiency:** Simple and clear way to represent data.
3. What are conditional statements in C++? Explain the if-else and switch statements.

Conditional Statements

Conditional statements in C++ are used to make decisions in the code based on certain conditions. They control the flow of the program by executing different blocks of code depending on whether a condition is true or false.

**if-else Statement**

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

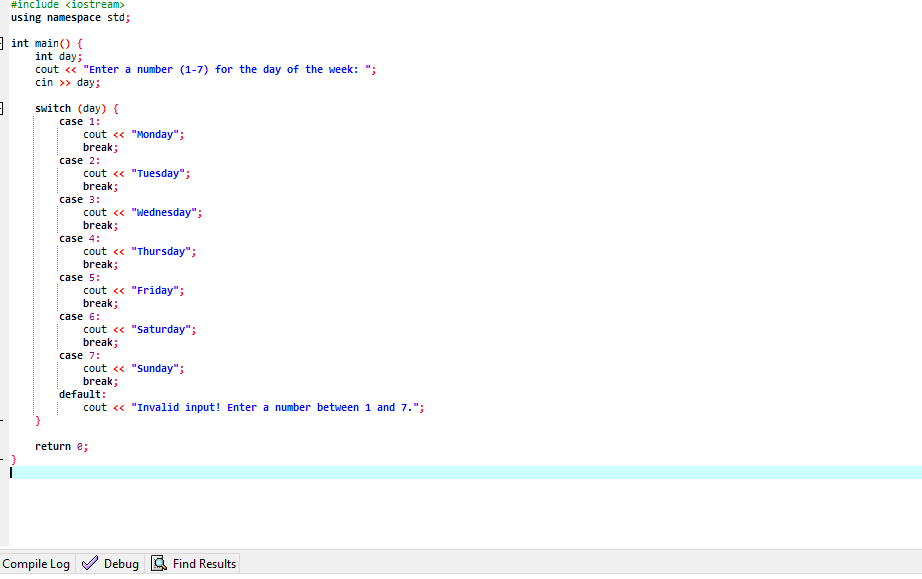


**2. switch Statement**

The switch statement is used when you want to select one of many possible options based on a single variable or expression.

The break statement prevents the execution from "falling through" to the next case.

The default block is optional and executes if none of the cases match



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

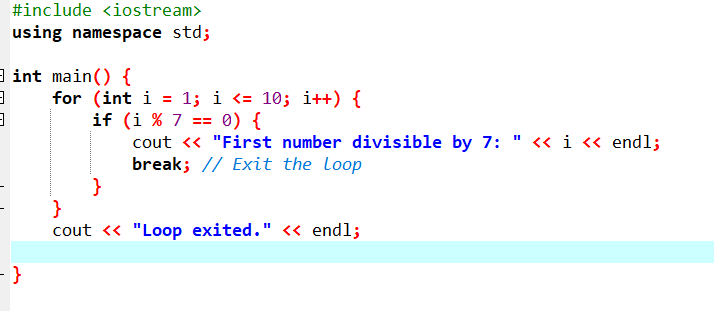
**Differences**

| **Feature** | **for Loop** | **while Loop** | **do-while Loop** |
| --- | --- | --- | --- |
| **Initialization** | Done in the loop header. | Done outside the loop. | Done outside the loop. |
|  |  |  |  |
| **Condition Check** | Checked **before** each iteration. | Checked **before** each iteration. | Checked **after** each iteration. |
| **Execution Guarantee** | May not execute if the condition is false initially. | May not execute if the condition is false initially. | Executes at least **once**, even if the condition is false. |
| **Use Case** | Best for loops with a known number of iterations or counters. | Best for loops with a condition that depends on external factors. | Best when the block needs to execute at least once. |

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

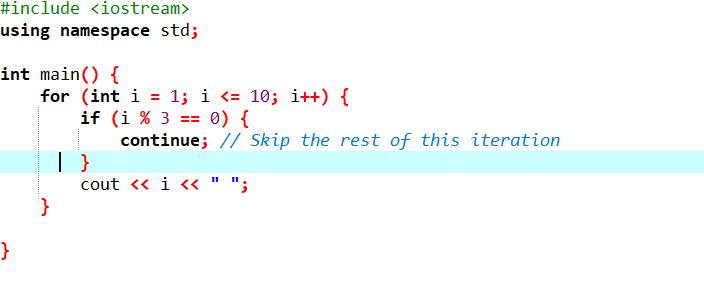
**1. break Statement**

The break statement is used to exit a loop prematurely. When a break is encountered, the control is transferred to the statement immediately after the loop.



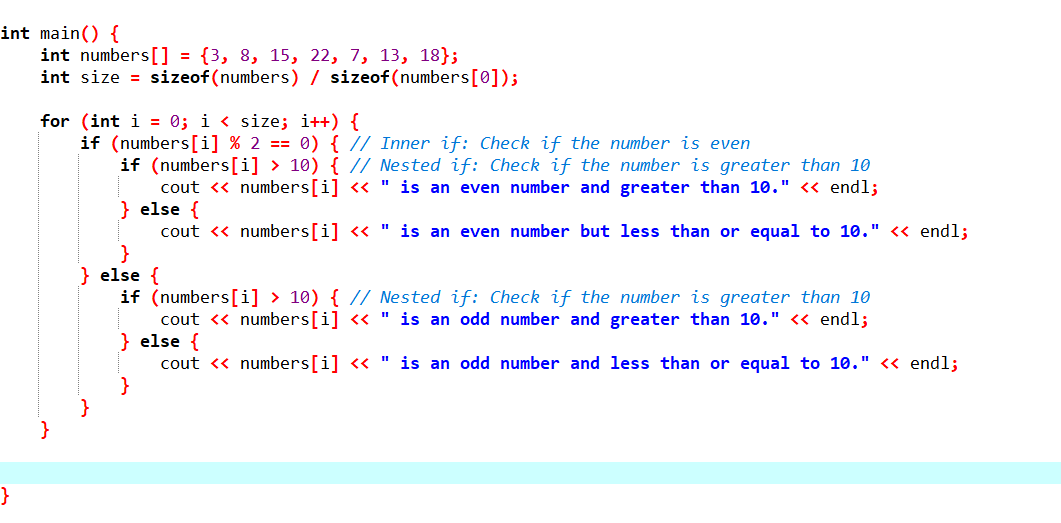
**2. continue Statement**

The continue statement is used to skip the rest of the loop's body for the current iteration and move to the next iteration.



1. Explain nested control structures with an example.

**Nested control structures** occur when one control structure, such as loops or conditional statements, is placed inside another. These are useful for solving problems that require multiple layers of logic or iterations



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

A function in C++ is a block of code designed to perform a specific task. Functions allow for better modularity and code reuse by enabling developers to divide the program into smaller, manageable, and reusable

1. **Function Declaration**:
   * int add(int a, int b); is the prototype of the function add, declaring its return type (int), name (add), and parameters (int a, int b).
2. **Function Definition**:
   * The block int add(int a, int b) { return a + b; } provides the logic for adding two integers.
3. **Function Calling**:
   * The function is invoked with the arguments num1 and num2. The returned value is stored in sum.
4. What is the scope of variables in C++? Differentiate between local and global scope.

The **scope of a variable** in C++ refers to the region of the program where the variable is accessible. Variables can be classified based on their scope as **local**, **global**,

**. Local Scope**

A variable with **local scope** is defined within a function or block and can only be accessed from within that specific function or block.

**Characteristics:**

* Declared inside a function or block (e.g., loops or conditionals).
* Not accessible outside the block in which they are declared.
* Memory is allocated when the block is entered and deallocated when the block is exited.

**2.Global Scope**

A variable with **global scope** is defined outside of all functions and is accessible from any function in the program.

**Characteristics:**

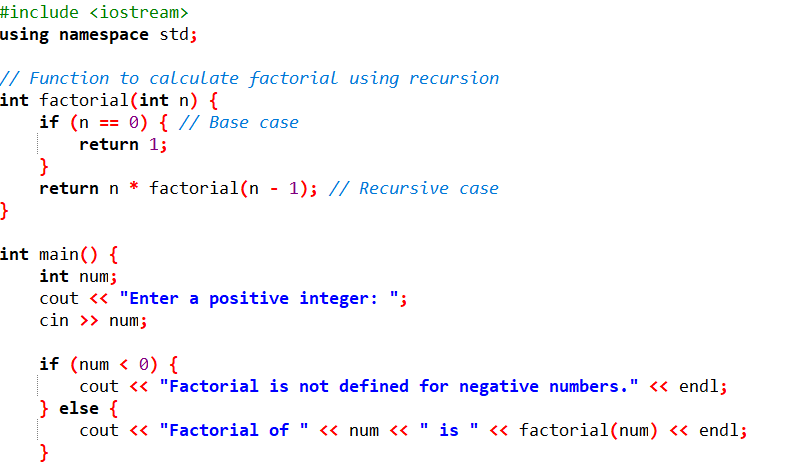
* Declared outside all functions, usually at the top of the program.
* Accessible by all functions within the program.
* Lifetime extends throughout the program execution.
* Can lead to naming conflicts and unintended side effects if not carefully managed.

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

**Recursion** is a programming technique where a function calls itself directly or indirectly to solve a problem. It breaks down a problem into smaller sub-problems of the same type, solving each sub-problem until a base condition is met.

**Key Components of Recursion**

1. **Base Case**:
   * The condition at which the recursion stops.
   * Ensures that the function does not call itself indefinitely.
2. **Recursive Case**:
   * Defines how the problem is broken down and the function calls itself



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

A **function prototype** in C++ is a declaration of a function that specifies its return type, name, and parameters, without providing the actual implementation. It serves as a forward declaration to inform the compiler about the function's existence before it is used in the code

 **Allow Function Calls Before Definition**:

* Function prototypes let you call a function before its full definition is provided in the program.

 **Enable Modular Code**:

* Prototypes enable a clean separation between declaration and definition, making code easier to read and maintain.

 **Type Checking**:

* The compiler uses the prototype to ensure that function calls match the declared parameters and return type.
* Mismatches (e.g., incorrect number of arguments or wrong types) result in a compile-time error.

 **Avoid Implicit Declarations**:

* Without a prototype, the compiler might assume a function returns int by default, which can lead to errors in modern C++.

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

An **array** in C++ is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store and manage multiple values under a single name, with each value being accessible using an index.

| **Single-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- |

|  |  |  |
| --- | --- | --- |
|  | Stores data in a linear format | Stores data in a tabular or matrix format |

|  |  |  |
| --- | --- | --- |
|  | Requires a single index (e.g., arr[i]) | Requires multiple indices (e.g., arr[i][j]) |

|  |  |  |
| --- | --- | --- |
|  | Stored in a single contiguous block | Stored as blocks for each dimension |

|  |  |  |
| --- | --- | --- |
|  | Best for lists, vectors, or sequences | Best for grids, tables, or matrices |

|  |  |  |
| --- | --- | --- |
|  | Simpler to implement and manage | More complex due to multiple indices |

1. Explain string handling in C++ with examples

In C++, strings can be handled in two primary ways:

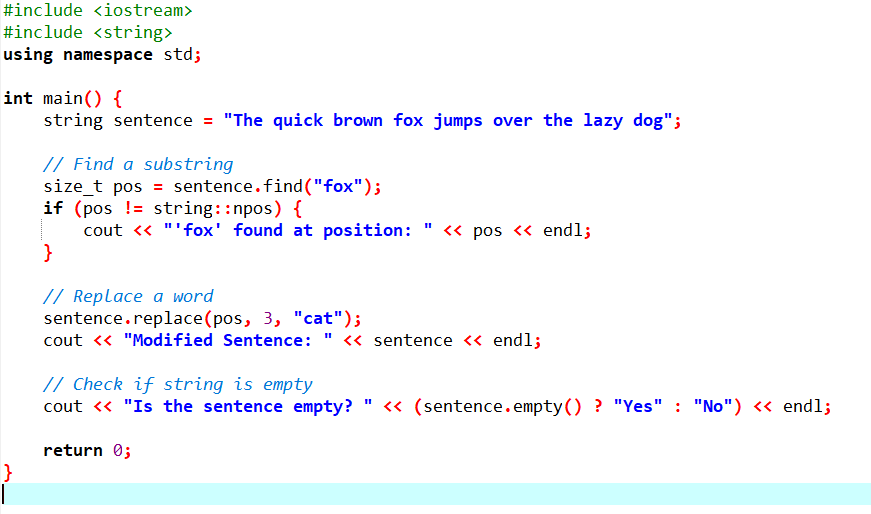
1. **Using C-style strings**: These are arrays of characters terminated by a null character (\0).
2. **Using the std::string class**: This is part of the Standard Template Library (STL) and offers a more convenient and powerful way to work with strings.

**1. C-Style Strings**

A C-style string is a character array where the last character is a null terminator (\0) to mark the end of the string

**2. std::string Class**

The std::string class provides a dynamic, safe, and feature-rich way to handle strings in C++



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

Arrays in can be initialized at the time of declaration.

**a. Full Initialization**

You can initialize all elements of the array explicitly.

**Partial Initialization**

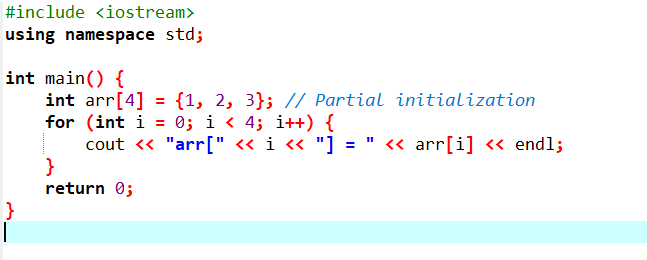
If fewer elements are provided, the remaining elements are automatically initialized to **0** (for numeric arrays).

**Default Initialization**

If no values are provided during declaration, the elements are **uninitialized**, meaning their values are indeterminate (garbage values

**d. Initializing All Elements to Zero**

Use empty curly braces {} to set all elements to zero



**2. Initializing 2D Arrays**

A **2D array** is an array of arrays (matrix) and can be initialized in several ways:

**a. Full Initialization**

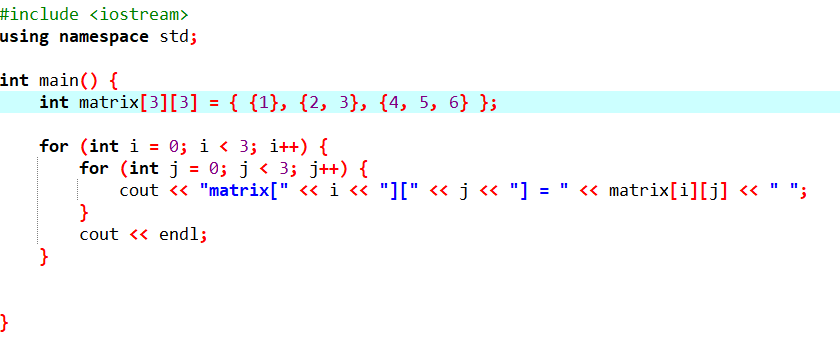
Specify all elements explicitly.

**Partial Initialization**

You can initialize some elements, and the rest are set to **0**.

**Default Initialization**

If no values are provided, the elements are uninitialized.



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

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods (functions) that operate on that data. The key concepts of OOP help in organizing and structuring code in a way that is modular, reusable, and easier to maintain. The four fundamental principles of OOP are Encapsulation, Abstraction, Inheritance, and Polymorphism.

Encapsulation

Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. This helps to hide the internal implementation details and protects the object’s state from unwanted access and modification. The access to the data is controlled via public and private access modifiers.

Private Members: Attributes and methods that cannot be accessed directly from outside the class.

Public Members: Methods that provide access to private members (getters and setters).

**Abstraction**

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of the object. It is achieved by defining interfaces (abstract classes) and only providing the necessary details to the user. The idea is to focus on what an object **does** rather than how it does it.

* **Abstract Class**: A class that cannot be instantiated and may contain abstract methods (pure virtual functions).
* **Interface**: A set of pure virtual functions that define a contract for derived classes to implement.

**3. Inheritance**

Inheritance is a mechanism in which one class (the **derived class**) inherits the properties and behaviors (methods) of another class (the **base class**). This promotes **code reuse** and allows the creation of a new class based on an existing class, with additional features or modifications.

* **Base Class**: The class that provides common functionality.
* **Derived Class**: The class that inherits from the base class and may add or override functionality

**4. Polymorphism**

Polymorphism means "many shapes" and allows objects of different classes to be treated as objects of a common base class. There are two types of polymorphism in C++:

* **Compile-time Polymorphism** (Method Overloading and Operator Overloading)
* **Runtime Polymorphism** (Method Overriding using Virtual Functions)

**a. Compile-time Polymorphism (Method Overloading)**

Method overloading allows multiple methods with the same name but different parameters in the same class.

**b. Runtime Polymorphism (Method Overriding)**

Method overriding allows a derived class to provide a specific implementation of a method that is already defined in its base class.

16. What are classes and objectsin C++? Provide an example

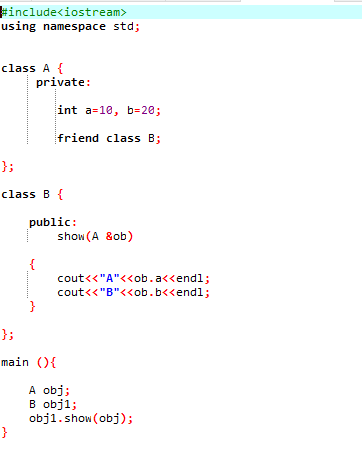
Classes and objects are the fundamental building blocks of Object-Oriented Programming (OOP) in C++. They provide a way to model real-world entities and their behavior in a program

**1. Class**

* A **class** is a user-defined data type that acts as a blueprint for creating objects.
* It defines the properties (**attributes**) and behaviors (**methods**) of the objects

**2. Object**

* An **object** is an instance of a class.
* When a class is defined, no memory is allocated until objects of the class are created.
* Objects use the properties and behaviors defined in the class

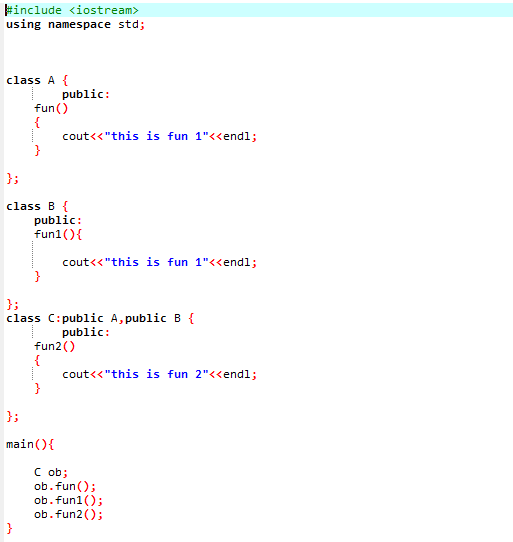


1. What isinheritance in C++? Explain with an example

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**). It promotes code reuse and establishes a relationship between classes

**Types of Inheritance in C++**

1. **Single Inheritance**: A derived class inherits from a single base class.
2. **Multiple Inheritance**: A derived class inherits from more than one base class.
3. **Multilevel Inheritance**: A class inherits from a derived class.
4. **Hierarchical Inheritance**: Multiple classes inherit from a single base class.
5. **Hybrid Inheritance**: A combination of two or more types of inheritance.



1. What is encapsulation in C++? How isit achieved in classes?

It refers to bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, called a class. Encapsulation also ensures data hiding, meaning the internal state of an object is protected from unintended or unauthorized access

Encapsulation is achieved using **access specifiers** in a class:

1. **Private**: Members declared as private can only be accessed within the class.
2. **Public**: Members declared as public can be accessed from outside the class.
3. **Protected**: Members declared as protected can be accessed in derived classes (used with inheritance)

