Module 3 - Introduction to OOPS Programming

Overview of C++ Programming

1. Introduction to C++

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

Ans: Procedural Programming and Object-Oriented Programming (OOP) are two fundamental paradigms of software development. Here are their key differences:

1. Core Concept

- **Procedural Programming**: Focuses on procedures (functions) and the sequence of actions to perform tasks. It organizes code into reusable functions.
- **OOP**: Focuses on objects, which are instances of classes. Objects combine data (attributes) and behavior (methods) into a single unit.

2. Modularity

- **Procedural Programming**: Code is organized into functions and modules. The separation of data and logic can make managing complex systems challenging.
- **OOP**: Code is organized into classes and objects. Encapsulation groups data and methods together, improving modularity and code reuse.

3. Data Handling

- **Procedural Programming**: Data and functions are separate. Functions operate on external data passed to them.
- **OOP**: Data is encapsulated within objects, and methods operate on the data inside the object. This promotes data integrity and security.

4. Abstraction

- **Procedural Programming**: Relies on functions and procedures for abstraction. Abstraction is typically less comprehensive.
- **OOP**: Supports abstraction through classes and interfaces, allowing the creation of complex models and reducing system complexity.

5. Reusability

- **Procedural Programming**: Functions can be reused, but reusability is limited compared to OOP.
- **OOP**: Supports inheritance and polymorphism, enabling significant code reuse and extension of functionality.

6. Flexibility and Scalability

- **Procedural Programming**: Works well for smaller, straightforward programs but can become unwieldy as complexity grows.
- **OOP**: Designed to handle complex systems with better scalability due to its structure and use of design principles.

7. Examples

- **Procedural Programming**: C, Fortran, Pascal
- **OOP**: Java, Python, C++, C#

8. Real-World Modeling

- **Procedural Programming**: Does not directly model real-world entities; focuses more on tasks and operations.
- **OOP**: Models real-world entities naturally using classes and objects.

9. Methodology

- **Procedural Programming**: Follows a linear, top-down approach where tasks are broken down into smaller functions.
- **OOP**: Follows a bottom-up approach where objects are built first, and interactions between them are defined.

In summary, Procedural Programming emphasizes processes and functions, while OOP emphasizes objects and their interactions, making OOP more suitable for large, complex, and scalable software systems.

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

Ans: Object-Oriented Programming (OOP) offers several advantages over Procedural-Oriented Programming (POP). These advantages arise from OOP's focus on organizing code around objects and their interactions. Here are the main advantages:

1. Modularity and Encapsulation

- **Explanation**: OOP encapsulates data and behavior into objects, which are self-contained units. This reduces dependencies between different parts of the code.
- **Advantage**: Encapsulation improves modularity, making it easier to isolate, debug, and maintain specific parts of the code without affecting the rest of the system.

2. Reusability

- **Explanation**: OOP supports inheritance, allowing developers to create new classes based on existing ones. Polymorphism enables the use of a single interface for different underlying forms.

- **Advantage**: Code can be reused across multiple projects or components, reducing redundancy and development effort.

3. Scalability

- **Explanation**: OOP's structure and design principles make it easier to manage and scale complex systems.
- **Advantage**: Adding new features or modifying existing ones becomes less error-prone because of the clear separation of concerns and organized code.

4. Abstraction

- **Explanation**: OOP allows abstraction through classes and interfaces. This hides unnecessary implementation details and focuses on the high-level structure.
- **Advantage**: Simplifies complex systems by allowing developers to work with a simplified model of an entity without worrying about underlying complexity.

5. Security

- **Explanation**: Data hiding in OOP restricts access to sensitive data using access modifiers (e.g., private, protected).
- **Advantage**: Ensures data integrity by preventing unauthorized access and unintended modifications.

6. Real-World Modeling

- **Explanation**: OOP is well-suited for modeling real-world entities and their interactions, using classes to represent objects and methods to define behavior.
- **Advantage**: Simplifies understanding and design of systems by closely mirroring real-world concepts and relationships.

7. Ease of Maintenance

- **Explanation**: OOP's modular approach and encapsulation simplify the process of updating or fixing code.
- **Advantage**: Developers can address specific issues or add new features without unintended side effects on other parts of the code.

8. Flexibility and Extensibility

- **Explanation**: Polymorphism and inheritance allow OOP systems to adapt and evolve with changing requirements.
- **Advantage**: Systems built with OOP can be extended with minimal changes to existing code.

9. Improved Collaboration

- **Explanation**: OOP's structure divides work into discrete, manageable units (objects or classes).

- **Advantage**: Facilitates team collaboration by enabling different team members to work on different parts of the system simultaneously.

10. Tool and Library Support

- **Explanation**: Most modern programming languages and frameworks are designed with OOP principles in mind, providing extensive libraries and tools.
- **Advantage**: Makes development faster and provides built-in solutions for common problems.

In summary, OOP's emphasis on structure, reusability, and abstraction makes it particularly advantageous for developing large, complex, and maintainable software systems compared to Procedural-Oriented Programming.

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

Ans: Setting up a C++ development environment involves installing the necessary tools and configuring them to write, compile, and run C++ programs effectively. Here's a step-by-step guide:

1. Choose and Install a C++ Compiler

- A C++ compiler translates your C++ code into executable machine code.
- **Popular C++ Compilers**:
  - GCC (GNU Compiler Collection): Common on Linux and available for Windows (via MinGW) and macOS.
  - Clang: Often used on macOS and also available for Linux and Windows.
  - MSVC (Microsoft Visual C++): Integrated into Visual Studio on Windows.
- **Steps**:
  1. On Linux: Use the package manager to install GCC or Clang (sudo apt install g++ on Ubuntu).
  2. On macOS: Install Xcode Command Line Tools (xcode-select --install).
  3. On Windows: Install MinGW or Visual Studio, which includes the MSVC compiler.

2. Install a Text Editor or Integrated Development Environment (IDE)

- A text editor or IDE is where you write your C++ code.
- **Popular Choices**:
  - **Text Editors**:
    - Visual Studio Code (VS Code): Lightweight and extensible.
    - Sublime Text or Notepad++: Simple editors with minimal setup.
  - **IDEs**:
    - Visual Studio: Feature-rich IDE for Windows, includes debugging tools and MSVC.
    - Code::Blocks: Lightweight, cross-platform C++ IDE.

        - CLion: A powerful IDE from JetBrains, supports many platforms.
- **Steps**:
    1. Download your preferred tool (e.g., VS Code or Visual Studio) from its official website.
    2. Install and configure extensions or plugins for C++ (e.g., "C/C++" extension for VS Code).

3. Configure the Compiler in the IDE or Editor

- Ensure the IDE or editor knows where to find the C++ compiler.
- **Steps**:
    - For VS Code:
        1. Install the "C/C++" extension from the Extensions Marketplace.
        2. Configure the compiler in the tasks.json file.
    - For Visual Studio:
        - Compiler is already integrated.
    - For other IDEs:
        - Specify the compiler's path in the settings (refer to the IDE's documentation).

4. Set Up a Debugger

- Debuggers help you find and fix errors in your code.
- **Steps**:
    - For GCC/Clang: Use GDB (GNU Debugger).
    - For MSVC: Debugger is built into Visual Studio.
- Ensure the debugger is properly configured in your IDE or text editor.

5. Verify the Setup

- Write a simple "Hello, World!" program to test your environment:

```
#include <iostream>
int main() {
   std::cout << "Hello, World!" << std::endl;
   return 0;
}
```

- Compile and run the program:
    - Using terminal/command line:

```
g++ -o hello hello.cpp
./hello
```

    - Using the IDE: Build and run the program using built-in buttons or menus.

6. Optional: Install Build Tools and Version Control

- **Build Tools**:
    - Use tools like CMake or Make for managing complex build processes.

- Install via your system's package manager or the official website.
- **Version Control**:
  - Install Git for version control and repository management.
  - Integrate with your IDE for seamless version control.

7. Additional Configurations

- **Set Environment Variables**:
  - Ensure the compiler's path is added to the system's PATH environment variable.
- **Install Libraries**:
  - Use libraries like Boost, SDL, or others for additional functionality.
  - Follow the library documentation for installation and linking.

Once these steps are complete, your C++ development environment will be ready, and you can start coding, debugging, and building C++ applications.

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

Ans: C++ provides a robust set of input/output (I/O) operations through its standard library, primarily using streams. The main I/O operations are handled by the following:

1. Input Operations

Input operations in C++ are typically performed using std::cin or file streams.

a. Using std::cin for Console Input

- **Purpose**: Reads data from the standard input (keyboard).
- **Example**:

```
#include <iostream>

int main() {
    int age;
    std::cout << "Enter your age: ";
    std::cin >> age;
    std::cout << "You entered: " << age << std::endl;
    return 0;
}
```

  - **Explanation**:
    - std::cin >> age; reads input from the user and stores it in the variable age.

b. Using getline() for String Input

- **Purpose**: Reads an entire line, including spaces.
- **Example**:

```
#include <iostream>
#include <string>

int main() {
    std::string name;
    std::cout << "Enter your full name: ";
    std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

  - **Explanation**:
    - std::getline(std::cin, name); reads a line of text into the name string.

2. Output Operations

Output operations are usually performed using std::cout or file streams.

a. Using std::cout for Console Output

- **Purpose**: Sends output to the standard output (screen).
- **Example**:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

  - **Explanation**:
    - std::cout outputs data.
    - std::endl inserts a newline and flushes the stream.

b. Formatted Output

- **Purpose**: Use manipulators for formatting.
- **Example**:

```
#include <iostream>
#include <iomanip> // For manipulators

int main() {
```

```
    double pi = 3.14159;
    std::cout << "Pi to 2 decimal places: " << std::fixed << std::setprecision(2)
<< pi << std::endl;
    return 0;
}
```

- **Explanation**:
  - std::fixed and std::setprecision(2) format the output to two decimal places.

3. File Input/Output

C++ supports file I/O using file stream classes from <fstream>.

a. File Output

- **Purpose**: Write data to a file.
- **Example**:

```
cpp
Copy code
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outFile("example.txt");
    if (outFile.is_open()) {
        outFile << "This is a line in a file." << std::endl;
        outFile.close();
        std::cout << "File written successfully." << std::endl;
    } else {
        std::cout << "Unable to open file." << std::endl;
    }
    return 0;
}
```

b. File Input

- **Purpose**: Read data from a file.
- **Example**:

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream inFile("example.txt");
    if (inFile.is_open()) {
        std::string line;
        while (std::getline(inFile, line)) {
```

```
            std::cout << line << std::endl;
        }
        inFile.close();
    } else {
        std::cout << "Unable to open file." << std::endl;
    }
    return 0;
}
```

4. Error Checking

- Always check if the stream is in a valid state (e.g., for file I/O):

```
if (!inFile) {
    std::cerr << "Error opening file." << std::endl;
}
```

Summary of Common I/O Operations

Operation	Method	Example
Console Input	std::cin	std::cin >> variable;
Line Input	std::getline	std::getline(std::cin, var);
Console Output	std::cout	std::cout << data;
File Input	std::ifstream	Read data from files
File Output	std::ofstream	Write data to files

By mastering these operations, you can effectively handle input and output in C++ for various applications.

2. Variables, Data

(Q1). What are the different data types available in C++? Explain with examples.

Ans: In C++, data types define the type of data a variable can hold. They are categorized into four main types:

1. Basic (or Primitive) Data Types:

These are the fundamental data types provided by C++.

Data Type	Description	Example
int	Integer numbers	int age = 25;
float	Floating-point numbers	float pi = 3.14f;
double	Double-precision floats	double g = 9.80665;
char	Single character	char grade = 'A';
bool	Boolean (true/false) values	bool isC++Fun = true;
void	No value (used in functions)	void display();

Example:

```
#include <iostream>
using namespace std;

int main() {
    int age = 30;
    float pi = 3.14f;
    double gravity = 9.80665;
    char grade = 'A';
    bool isFun = true;

    cout << "Age: " << age << ", Pi: " << pi << ", Gravity: " << gravity << endl;
    cout << "Grade: " << grade << ", Is C++ fun? " << isFun << endl;
    return 0;
}
```

2. Derived Data Types:

These are derived from the basic data types.

Data Type	Description	Example

Array	Collection of elements of the same type	int arr[5] = {1, 2, 3, 4, 5};
Pointer	Stores the address of another variable	int* ptr = &age;
Reference	Alias for another variable	int& ref = age;
Function	Code that performs a specific task	int add(int a, int b);

Example:

```
#include <iostream>
using namespace std;

int main() {
    int arr[3] = {10, 20, 30};
    int num = 50;
    int* ptr = &num;
    int& ref = num;

    cout << "Array element: " << arr[0] << endl;
    cout << "Pointer value: " << *ptr << endl;
    cout << "Reference value: " << ref << endl;
    return 0;
}
```

3. User-Defined Data Types:

These allow customization of data types.

Data Type	Description	Example
Class	Blueprint for creating objects	class Car { ... };
Structure	Groups related variables	struct Point { int x, y; };
Enumeration	Defines a set of named integer constants	enum Color { RED, GREEN };
Typedef/alias	Creates a new name for an existing type	typedef unsigned int uint;

Example:

```
#include <iostream>
using namespace std;
```

```
struct Point {
    int x, y;
};

enum Color { RED, GREEN, BLUE };

int main() {
    Point p = {10, 20};
    Color favorite = GREEN;

    cout << "Point: (" << p.x << ", " << p.y << ")" << endl;
    cout << "Favorite color: " << favorite << endl;
    return 0;
}
```

4. Abstract or Special Data Types:

These are more advanced data types.

Data Type	Description	Example
String	Represents a sequence of characters	string name = "Alice";
Vector	Dynamic array	vector<int> nums;
List	Doubly linked list	list<int> myList;
Map	Key-value pairs	map<int, string> myMap;

Example:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    string name = "Alice";
    vector<int> numbers = {1, 2, 3};

    cout << "Name: " << name << endl;
    cout << "Numbers: ";
    for (int n : numbers) {
        cout << n << " ";
    }
    cout << endl;
    return 0;
```

```
}
```

By combining these data types, you can represent a wide range of information and structures in C++.

(Q2). Explain the difference between implicit and explicit type conversion in C++.

Ans: In C++, **type conversion** is the process of converting a value from one data type to another. There are two types of type conversions: **implicit** and **explicit**.

1. Implicit Type Conversion (Type Promotion or Coercion):

- Implicit conversion is automatically performed by the compiler.
- It occurs when a value of one data type is converted to another compatible data type without explicit intervention by the programmer.
- This usually happens in operations involving mixed data types or during assignment to a larger or compatible data type.

Characteristics:

1. No special syntax is required; the conversion happens automatically.
2. Typically converts smaller types to larger types to avoid data loss (e.g., int to float).
3. May lead to precision loss or unexpected results in some cases.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int intVar = 42;
    double doubleVar = intVar;  // Implicit conversion from int to double

    cout << "Integer value: " << intVar << endl;
    cout << "Converted to double: " << doubleVar << endl;

    float result = intVar / 5;  // intVar is implicitly converted to float for division
    cout << "Result of division: " << result << endl;

    return 0;
}
```

2. Explicit Type Conversion (Type Casting):

- Explicit conversion is performed manually by the programmer using **type casting**.
- The programmer specifies the desired type, overriding the compiler's default behavior.
- Explicit conversion is used when implicit conversion is not available or when precision control is required.

Syntax:

- **C-Style Casting:** (new_type) value
- **C++ Casting Operators:**
  - static_cast<new_type>(value)
  - dynamic_cast<new_type>(value) (used for polymorphism)
  - const_cast<new_type>(value) (adds/removes const)
  - reinterpret_cast<new_type>(value) (for low-level conversions)

Characteristics:

1. Requires explicit syntax, ensuring clarity and programmer intent.
2. Used to convert incompatible types or for custom conversion.
3. Can lead to runtime errors if not used carefully.

Example:

```
#include <iostream>
using namespace std;

int main() {
   double doubleVar = 3.14159;
   int intVar;

   // Explicit conversion using C-style casting
   intVar = (int)doubleVar;
   cout << "Double to int (C-style cast): " << intVar << endl;

   // Explicit conversion using static_cast
   intVar = static_cast<int>(doubleVar);
   cout << "Double to int (static_cast): " << intVar << endl;

   return 0;
}
```

Key Differences:

Aspect	Implicit Conversion	Explicit Conversion
Trigger	Automatic by the compiler.	Performed manually by the programmer.

Syntax	No syntax required.	Requires casting syntax (e.g., (type) or static_cast).
Control	Compiler decides the conversion.	Programmer controls the conversion.
Safety	Relatively safe, but precision loss may occur.	Can lead to undefined behavior if done improperly.
Example	floatVar = intVar;	intVar = (int)floatVar;

Illustrative Example:

```
#include <iostream>
using namespace std;

int main() {
    int intVar = 7;
    double doubleVar = 3.2;

    // Implicit conversion: intVar is converted to double
    double resultImplicit = intVar + doubleVar;
    cout << "Implicit Conversion Result: " << resultImplicit << endl;

    // Explicit conversion: doubleVar is cast to int
    int resultExplicit = intVar + static_cast<int>(doubleVar);
    cout << "Explicit Conversion Result: " << resultExplicit << endl;

    return 0;
}
```

Output:

```
Implicit Conversion Result: 10.2
Explicit Conversion Result: 10
```

In the first case, intVar is implicitly converted to double. In the second case, doubleVar is explicitly converted to int before the addition.

(Q3). What are the different types of operators in C++? Provide examples of each.

Ans: C++ provides a wide range of operators for performing various operations. These operators can be classified into several categories based on their functionality.

1. Arithmetic Operators

Used to perform mathematical operations on numeric data types.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 10, b = 3;
    cout << "Addition: " << a + b << endl;
    cout << "Modulus: " << a % b << endl;
    return 0;
}
```

2. Relational (Comparison) Operators

Used to compare two values and return a boolean result.

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

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 10;
    cout << "a is less than b: " << (a < b) << endl;
    return 0;
}
```

3. Logical Operators

Used to combine multiple conditions or to perform logical operations.

Operator	Description	Example
&&	Logical AND	(a > b) && (c < d)
`		`
!	Logical NOT	!(a == b)

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 10;
    cout << "Logical AND: " << ((a < b) && (b > 0)) << endl;
    cout << "Logical OR: " << ((a > b) || (b > 0)) << endl;
    return 0;
}
```

4. Bitwise Operators

Operate at the bit level and are used for binary computations.

Operator	Description	Example
&	Bitwise AND	a & b
`	`	Bitwise OR

^	Bitwise XOR	a ^ b
~	Bitwise Complement	~a
<<	Left Shift	a << 2
>>	Right Shift	a >> 2

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 3;
    cout << "Bitwise AND: " << (a & b) << endl;
    cout << "Left Shift: " << (a << 1) << endl;
    return 0;
}
```

5. Assignment Operators

Used to assign values to variables.

Example:

Operator	Description	Example
=	Assign value	a = b
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;
```

```
    a += 5;
    cout << "Updated value of a: " << a << endl;
    return 0;
}
```

6. Increment and Decrement Operators

Used to increase or decrease a value by 1.

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

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5;
    cout << "Pre-increment: " << ++a << endl;
    cout << "Post-increment: " << a++ << endl;
    cout << "Current value of a: " << a << endl;
    return 0;
}
```

7. Conditional (Ternary) Operator

Used as a shorthand for an if-else condition.

Operator	Description	Example
? :	Condition evaluation	condition ? value1 : value2

Example:

```
#include <iostream>
using namespace std;
```

```
int main() {
    int a = 5, b = 10;
    int max = (a > b) ? a : b;
    cout << "Maximum value: " << max << endl;
    return 0;
}
```

8. Special Operators

Operator	Description	Example
sizeof	Returns the size of a data type	sizeof(int)
&	Address of a variable	&a
*	Pointer dereference	*ptr
->	Member access via pointer	ptr->member
.	Member access	obj.member

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    cout << "Size of int: " << sizeof(a) << endl;
    cout << "Address of a: " << &a << endl;
    return 0;
}
```

Summary:

C++ operators allow for a wide range of operations, from arithmetic and logical comparisons to advanced bitwise and pointer manipulations, making it a versatile and powerful language for diverse programming needs.

(Q4). Explain the purpose and use of constants and literals in C++.

Ans: In C++, **constants** and **literals** are used to represent fixed values that do not change during the execution of a program. They enhance the readability, reliability, and maintainability of the code by ensuring that some values remain constant and cannot be modified accidentally.

Constants in C++

Constants are variables whose values cannot be changed after initialization. They are declared using the const keyword or preprocessor directives.

Purpose of Constants:

1. **Protect Values:** Prevent accidental modification of values that should remain constant (e.g., Pi, tax rates).
2. **Improve Readability:** Make the code easier to understand by giving meaningful names to constant values.
3. **Enhance Maintainability:** Centralize constant values so they can be updated in one place if needed.

Types of Constants:

1. **Constant Variables:** Declared using the const keyword.

   const data_type variable_name = value;

   Example:

```
#include <iostream>
using namespace std;

int main() {
    const double pi = 3.14159;
    cout << "Value of Pi: " << pi << endl;

    // Uncommenting the next line will cause a compilation error
    // pi = 3.14;
    return 0;
}
```

2. **Macro Constants (Preprocessor Directives):** Defined using the #define preprocessor directive.

   #define constant_name value

   Example:

```
#include <iostream>
#define PI 3.14159

int main() {
   cout << "Value of PI: " << PI << endl;
   return 0;
}
```

Note: Macros are replaced during preprocessing and do not offer type checking.

3. **Enumerations (enum):** Used to define a set of related constant integer values.

   ```
   enum Color { RED, GREEN, BLUE };
   ```

   Example:

   ```
   #include <iostream>
   using namespace std;

   enum Color { RED = 1, GREEN = 2, BLUE = 3 };

   int main() {
      Color favoriteColor = GREEN;
      cout << "Favorite color code: " << favoriteColor << endl;
      return 0;
   }
   ```

4. **constexpr Constants:** Introduced in C++11, used for values that should be evaluated at compile time.

   ```
   constexpr data_type variable_name = value;
   ```

   Example:

   ```
   #include <iostream>
   using namespace std;

   constexpr int square(int x) {
      return x * x;
   }

   int main() {
      constexpr int result = square(5);
      cout << "Square of 5: " << result << endl;
      return 0;
   }
   ```

Literals in C++

Literals are fixed values that are directly written in the code and represent specific data types.

Purpose of Literals:

1. Represent constant values directly in the code.
2. Provide initialization values for variables.

Types of Literals:

1. **Integer Literals:** Represent whole numbers. They can be written in:
   - **Decimal:** Base 10 (e.g., 123)
   - **Octal:** Base 8 (prefix 0, e.g., 012)
   - **Hexadecimal:** Base 16 (prefix 0x or 0X, e.g., 0x7B)
   - **Binary (C++14):** Base 2 (prefix 0b or 0B, e.g., 0b1011) **Example:**

   ```
   int decimal = 123;
   int octal = 012;       // 10 in decimal
   int hex = 0x7B;        // 123 in decimal
   int binary = 0b1011;   // 11 in decimal
   ```

2. **Floating-Point Literals:** Represent decimal numbers or numbers in scientific notation. **Example:**

   ```
   double pi = 3.14;
   float g = 9.8f;      // 'f' specifies a float literal
   double e = 2.71e3;   // Scientific notation (2.71 × 10^3)
   ```

3. **Character Literals:** Represent a single character enclosed in single quotes. **Example:**

   ```
   char grade = 'A';
   char newline = '\n'; // Escape sequence for a new line
   ```

4. **String Literals:** Represent sequences of characters enclosed in double quotes. **Example:**

   ```
   const char* message = "Hello, C++!";
   ```

5. **Boolean Literals:** Represent true or false. **Example:**

   ```
   bool isFun = true;
   ```

6. **Null Pointer Literal:** Represents a null pointer using nullptr (introduced in C++11). **Example:**

   ```
   int* ptr = nullptr;
   ```

Key Differences Between Constants and Literals

Aspect	Constants	Literals
Definition	Named, fixed values defined in code.	Fixed values directly written in code.
Syntax	Requires const, constexpr, or #define.	Directly written (e.g., 42, 3.14).
Usage	Can be reused multiple times.	Used in expressions or assignments.
Example	const double pi = 3.14159;	3.14159 (used directly in the code).

Example Demonstrating Constants and Literals

```
#include <iostream>
#define PI 3.14159

using namespace std;

int main() {
    const int radius = 5;       // Constant variable
    constexpr double gravity = 9.8;  // Compile-time constant

    double circumference = 2 * PI * radius;  // Using literal and constant
    cout << "Circumference of circle: " << circumference << endl;

    double force = gravity * radius;  // Using constexpr constant
    cout << "Force: " << force << endl;

    return 0;
}
```

Output:

```
Circumference of circle: 31.4159
Force: 49
```

In this example, PI, radius, and gravity are constants, while 2, 5, and 9.8 are literals. Together, they provide a clear and maintainable way to define fixed values.

3.Control Flow Statements

(Q1). What are conditional statements in C++? Explain the if-else and switch statements.

Ans: Conditional statements in **C++** are used to make decisions based on certain conditions. These statements allow the program to choose a particular set of instructions to execute depending on whether a condition evaluates to **true** or **false**.

The two most common conditional statements are **if-else** and **switch**.

1. if-else Statement

The if-else statement executes a block of code based on whether a condition is true or false.

Syntax:

```
if (condition) {
   // Code block executed if condition is true
}
else {
   // Code block executed if condition is false
}
```

Explanation:

1. **condition**: A logical expression that evaluates to true or false.
2. If the condition evaluates to true, the code inside the if block is executed.
3. If the condition evaluates to false, the code inside the else block is executed.

Example:

```
#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;
}
```

Variations:

- **if-else if Ladder**: Used to check multiple conditions sequentially.

```
if (condition1) {
   // Executed if condition1 is true
}
else if (condition2) {
   // Executed if condition2 is true
}
else {
   // Executed if all conditions are false
}
```

- **Nested if**: if statements inside another if block.

2. switch Statement

The switch statement allows a variable or expression to be compared against a set of predefined constant values. It is often used as a cleaner alternative to a series of if-else if statements.

Syntax:

```
switch (expression) {
   case constant1:
      // Code to execute if expression == constant1
      break;
   case constant2:
      // Code to execute if expression == constant2
      break;
   ...
   default:
      // Code to execute if none of the above cases match
}
```

Explanation:

1. **expression**: This is evaluated and compared with each case constant.
2. **case**: If the expression matches a case, the corresponding block of code is executed.
3. **break**: The break statement prevents fall-through, which means it stops execution and exits the switch statement.
4. **default**: This is executed if none of the case values match the expression.

Example:

```
#include <iostream>
```

```
using namespace std;

int main() {
   int day;
   cout << "Enter a day number (1-7): ";
   cin >> day;

   switch (day) {
      case 1:
         cout << "Monday" << endl;
         break;
      case 2:
         cout << "Tuesday" << endl;
         break;
      case 3:
         cout << "Wednesday" << endl;
         break;
      case 4:
         cout << "Thursday" << endl;
         break;
      case 5:
         cout << "Friday" << endl;
         break;
      case 6:
         cout << "Saturday" << endl;
         break;
      case 7:
         cout << "Sunday" << endl;
         break;
      default:
         cout << "Invalid day number!" << endl;
   }

   return 0;
}
```

Key Differences Between if-else and switch:

if-else	switch
Works for any condition or range of values.	Works only for constant expressions.
Evaluates conditions sequentially.	Evaluates a single expression against cases.
More flexible but less readable for multiple checks.	Better for multiple values of the same variable.

When to Use:

- Use **if-else** when conditions involve relational or logical operators.
- Use **switch** when checking a variable against specific constants, as it provides better clarity and performance for such cases.

(Q2). What is the difference between for, while, and do-while loops in C++?

Ans: In **C++**, loops are used to execute a block of code repeatedly as long as a specified condition is true. The main types of loops are **for**, **while**, and **do-while** loops. Each has its use case depending on the situation.

1. for Loop

The `for` loop is used when the number of iterations is **known beforehand**. It is commonly used for counting or iterating through a range.

Syntax:

```
for (initialization; condition; update) {
    // Code to be executed
}
```

Explanation:

1. **initialization**: This step initializes the loop control variable (executed once at the beginning).
2. **condition**: This is checked before each iteration. If it evaluates to `true`, the loop body executes. If `false`, the loop terminates.
3. **update**: This step updates the loop control variable after each iteration.

Example:

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) { // Initialization: i=1, Condition: i<=5,
Update: i++
        cout << i << " "; // Output: 1 2 3 4 5
    }
    return 0;
}
```

2. while Loop

The `while` loop is used when the number of iterations is **not known beforehand** and depends on a condition being true. If the condition is `false` at the start, the loop body will not execute even once.

Syntax:

```
cpp
Copy code
while (condition) {
    // Code to be executed
}
```

Explanation:

1. **condition**: It is checked at the beginning of each iteration. If `true`, the loop body executes; otherwise, the loop terminates.
2. The loop continues until the condition becomes `false`.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int i = 1;
    while (i <= 5) { // Condition: i <= 5
        cout << i << " "; // Output: 1 2 3 4 5
        i++; // Increment i
    }
    return 0;
}
```

3. do-while Loop

The `do-while` loop is similar to the `while` loop, but it **guarantees that the loop body will execute at least once**. The condition is checked **after** the loop body is executed.

Syntax:

```
do {
    // Code to be executed
} while (condition);
```

Explanation:

1. The loop body executes **once** before the condition is checked.
2. After the first execution, if the condition is `true`, the loop continues; otherwise, it terminates.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int i = 1;
    do {
        cout << i << " "; // Output: 1 2 3 4 5
        i++;
    } while (i <= 5); // Condition: i <= 5

    return 0;
}
```

Key Differences Between `for`, `while`, and `do-while` Loops:

Feature	for Loop	while Loop	do-while Loop
When Used	When the number of iterations is known beforehand.	When the number of iterations depends on a condition.	When the loop body must execute at least once.
Condition Check	At the start of the loop.	At the start of the loop.	After the loop body executes.
Execution	The body may not execute if the condition is false initially.	The body may not execute if the condition is false initially.	The body executes at least once, regardless of condition.
Structure	Initialization, condition, and update in one line.	Initialization and update must be done manually.	Initialization and update must be done manually.
Common Use Case	Counting or iterating over a range.	Condition-based repetition.	Post-condition execution.

Example Demonstrating All Loops:

```
#include <iostream>
using namespace std;

int main() {
    // for loop example
    cout << "For loop: ";
    for (int i = 1; i <= 3; i++) {
        cout << i << " ";
    }

    // while loop example
    cout << "\nWhile loop: ";
    int j = 1;
    while (j <= 3) {
        cout << j << " ";
```

```
        j++;
    }

    // do-while loop example
    cout << "\nDo-while loop: ";
    int k = 1;
    do {
        cout << k << " ";
        k++;
    } while (k <= 3);

    return 0;
}
```

Output:

```
For loop: 1 2 3
While loop: 1 2 3
Do-while loop: 1 2 3
```

When to Use Each Loop:

1. **for Loop**: Use it when you know the number of iterations (e.g., iterating through arrays or ranges).
2. **while Loop**: Use it when the loop depends on a condition that is checked before execution.
3. **do-while Loop**: Use it when you want the loop to execute at least once regardless of the condition.

(Q3). How are break and continue statements used in loops? Provide examples.

Ans: The **break** and **continue** statements in C++ are used to control the flow of loops (like for, while, and do-while loops). They help manage how the loop executes under specific conditions.

1. break Statement

The break statement is used to **terminate** the loop immediately, regardless of the loop's condition. When break is encountered, the program exits the loop and continues executing the code after the loop.

Usage:

- To **exit** a loop when a certain condition is met.
- Often used in conjunction with conditional statements (if).

Example (Using break in a for loop):

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            cout << "Breaking at i = " << i << endl;
            break; // Exit the loop when i == 5
        }
        cout << "i = " << i << endl;
    }
    cout << "Loop terminated!" << endl;
    return 0;
}
```

Output:

```
i = 1
i = 2
i = 3
i = 4
Breaking at i = 5
Loop terminated!
```

Explanation:

- The loop runs until i == 5.
- When break is encountered, the loop terminates immediately, skipping the rest of the iterations.

2. continue Statement

The continue statement is used to **skip** the current iteration of the loop and move to the next iteration. It does not terminate the loop but skips the remaining code in the current iteration.

Usage:

- To **skip certain values** or iterations in the loop.
- Often used with conditional statements (if).

Example (Using continue in a for loop):

```
#include <iostream>
using namespace std;

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

```
        if (i == 3) {
            cout << "Skipping i = " << i << endl;
            continue; // Skip the rest of the code for i == 3
        }
        cout << "i = " << i << endl;
    }
    cout << "Loop finished!" << endl;
    return 0;
}
```

Output:

```
i = 1
i = 2
Skipping i = 3
i = 4
i = 5
Loop finished!
```

Explanation:

- When i == 3, the continue statement skips the rest of the loop body for that iteration.
- The loop then proceeds with the next iteration.

Differences Between break and continue

Feature	break	continue
Effect	Terminates the loop entirely.	Skips the current iteration of the loop and proceeds to the next iteration.
Where Used	Can be used in for, while, do-while, and switch statements.	Can be used in for, while, and do-while loops.
When Used	When you need to exit the loop early.	When you need to skip certain iterations.

Examples in a while Loop

Example with break:

```
#include <iostream>
using namespace std;

int main() {
    int i = 1;
```

```
    while (i <= 10) {
        if (i == 6) {
            cout << "Breaking at i = " << i << endl;
            break; // Exit the loop when i == 6
        }
        cout << "i = " << i << endl;
        i++;
    }
    return 0;
}
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
Breaking at i = 6
```

Example with continue:

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    while (i < 5) {
        i++;
        if (i == 3) {
            cout << "Skipping i = " << i << endl;
            continue; // Skip the rest of the code for i == 3
        }
        cout << "i = " << i << endl;
    }
    return 0;
}
```

Output:

```
i = 1
i = 2
Skipping i = 3
i = 4
i = 5
```

Conclusion

1. Use **break** to terminate the loop entirely when a condition is met.

2. Use **continue** to skip the current iteration and proceed to the next iteration.

Both are powerful tools for managing loops efficiently in C++.

(Q4).Explain nested control structures with an example.

Ans**: A** nested control structure **refers to placing one control structure (like a loop, if statement, or switch case)** inside another**. This is a common practice in programming to handle complex logic where multiple conditions or loops are involved.**

Types of Nested Control Structures

1. **Nested if statements**
2. **Nested loops (e.g., for inside a for, while inside a for, etc.)**
3. **Control structures inside loops or conditionals** (e.g., an if inside a for loop)

1. Nested if Statements

An if statement inside another if or else block is a nested if. It allows us to test multiple conditions in a hierarchical way.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;

    if (num > 0) { // Outer if
        if (num % 2 == 0) { // Inner if
            cout << "The number is positive and even." << endl;
        } else {
            cout << "The number is positive and odd." << endl;
        }
    } else {
        cout << "The number is not positive." << endl;
    }
    return 0;
}
```

Output:

- Input: 4 → "The number is positive and even."
- Input: 3 → "The number is positive and odd."
- Input: -2 → "The number is not positive."

---

2. Nested Loops

A **nested loop** is a loop inside another loop. It is commonly used for tasks involving grids, tables, or repetitive patterns.

Example (Printing a Number Pyramid):

```
#include <iostream>
using namespace std;

int main() {
    int rows;
    cout << "Enter the number of rows: ";
    cin >> rows;

    for (int i = 1; i <= rows; i++) { // Outer loop for rows
        for (int j = 1; j <= i; j++) { // Inner loop for printing numbers
            cout << j << " ";
        }
        cout << endl; // Move to the next line after each row
    }
    return 0;
}
```

Output (if rows = 4):

```
1
1 2
1 2 3
1 2 3 4
```

Explanation:

- The **outer loop** controls the number of rows (i increases from 1 to rows).
- The **inner loop** prints numbers from 1 to i in each row.

3. Control Structures Inside Loops

You can place conditionals like if-else or even another loop inside a loop to control the behavior dynamically.

Example (Finding Even and Odd Numbers in a Range):

```
#include <iostream>
using namespace std;

int main() {
    int start, end;
    cout << "Enter the start and end of the range: ";
    cin >> start >> end;

    for (int i = start; i <= end; i++) { // Loop through the range
        if (i % 2 == 0) { // Check if even
            cout << i << " is Even" << endl;
        } else { // If not even, it's odd
            cout << i << " is Odd" << endl;
        }
    }
    return 0;
}
```

Output:

- Input: start = 3, end = 6

```
3 is Odd
4 is Even
5 is Odd
6 is Even
```

Explanation:

- The for loop iterates through the range of numbers.
- The if statement checks if the number is divisible by 2 (even) or not (odd).

4. Combining Different Control Structures

You can combine loops, conditionals, and other control structures in complex ways.

Example (Nested Loops with an if Condition):

```
#include <iostream>
using namespace std;

int main() {
    int rows;
    cout << "Enter the number of rows: ";
    cin >> rows;
```

```
    for (int i = 1; i <= rows; i++) { // Outer loop for rows
        for (int j = 1; j <= rows; j++) { // Inner loop for columns
            if (j <= i) { // Condition to print stars
                cout << "* ";
            } else {
                cout << "  "; // Print spaces
            }
        }
        cout << endl;
    }
    return 0;
}
```

Output (if rows = 4):

```
*
* *
* * *
* * * *
```

Explanation:

- The outer loop controls the rows.
- The inner loop controls the columns.
- An if condition ensures stars (*) are printed only in the left triangle, and spaces fill the rest.

Summary

- **Nested if**: Allows multiple conditions to be checked in sequence.
- **Nested loops**: Repeats a loop inside another loop, ideal for grids or tables.
- **Control structures inside loops**: Combines loops and conditionals for complex behaviors.

Nested control structures make programs flexible and powerful for handling complex logic.

4.Functions and Scope

(Q1.) What is a function in C++? Explain the concept of function declaration, definition, and calling.

Ans: In **C++**, a **function** is a block of code that performs a specific task. Functions are used to modularize and organize code, making it reusable and easier to read and maintain.

A function typically consists of the following components:

1. **Function Declaration** (or Prototype):
   - It specifies the function's name, return type, and parameters, but it does not include the function body.
   - The purpose of the declaration is to inform the compiler about the function's existence before it is used in the program.
   - Example:

     ```
     int add(int a, int b); // Function declaration
     ```

2. **Function Definition**:
   - It provides the actual implementation of the function.
   - The definition includes the function's return type, name, parameters, and the body where the task is implemented.
   - Example:

     ```
     int add(int a, int b) { // Function definition
        return a + b;
     }
     ```

3. **Function Call**:
   - It is the process of invoking or executing the function by specifying its name and providing the necessary arguments.
   - Example:

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

Example Demonstrating Function Declaration, Definition, and Calling

```
#include <iostream>
using namespace std;

// Function Declaration
int multiply(int x, int y);

int main() {
   int a = 5, b = 3;

   // Function Call
   int product = multiply(a, b);

   // Output the result
   cout << "Product: " << product << endl;

   return 0;
}

// Function Definition
```

```
int multiply(int x, int y) {
   return x * y;
}
```

Key Points:

- The **declaration** helps the compiler know about the function before its actual definition or use.
- The **definition** contains the function's logic.
- The **call** executes the function and uses its return value if applicable.

This modular approach to coding improves program organization and readability.

(Q2.) What is the scope of variables in C++? Differentiate between local and global scope.

Ans: The **scope** of a variable defines the part of the program where the variable is accessible or visible. In C++, there are two primary types of scope:

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

Local Scope

- **Definition**: A variable declared inside a function or a block is said to have a local scope. It is accessible only within that function or block.
- **Characteristics**:
  - The variable is created when the function or block is entered and destroyed when it is exited.
  - It is not visible or accessible outside the function or block in which it is defined.
- **Example**:

```
void display() {
   int x = 10; // Local variable
   cout << "Value of x: " << x << endl;
}

int main() {
   display();
   // cout << x; // Error: x is not accessible here
   return 0;
}
```

Global Scope

- **Definition**: A variable declared outside all functions is said to have a global scope. It is accessible from any part of the program, including all functions.
- **Characteristics**:
  - The variable exists throughout the program's lifetime.
  - Any function can access or modify the global variable.
  - Excessive use of global variables can lead to unexpected side effects and reduced maintainability.
- **Example**:

```
int x = 20; // Global variable

void display() {
   cout << "Global x: " << x << endl;
}

int main() {
   cout << "Global x in main: " << x << endl;
   display();
   return 0;
}
```

Differences Between Local and Global Scope

Feature	Local Scope	Global Scope
Declaration	Declared inside a function or block.	Declared outside all functions.
Lifetime	Exists only during the execution of the block.	Exists for the entire program's lifetime.
Accessibility	Accessible only within the function/block.	Accessible from any part of the program.
Memory Usage	Memory is allocated on the stack.	Memory is allocated in the global data area.
Usage	Typically used for temporary or specific tasks.	Used for shared data across the program.

Example Demonstrating Local and Global Variables

```
#include <iostream>
using namespace std;

int globalVar = 100; // Global variable
```

```
void display() {
    int localVar = 50; // Local variable
    cout << "Local Variable: " << localVar << endl;
    cout << "Global Variable: " << globalVar << endl;
}

int main() {
    display();

    // Modifying global variable
    globalVar = 200;
    cout << "Modified Global Variable: " << globalVar << endl;

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

    return 0;
}
```

By understanding and managing the scope of variables, developers can avoid naming conflicts and ensure better program structure.

(Q3.) Explain recursion in C++ with an example.

Ans: Recursion **i**s a programming technique where a function calls itself directly or indirectly to solve a problem. It is commonly used for problems that can be divided into smaller, similar subproblems. A recursive function typically has two main components:

1. **Base Case**: The condition under which the recursion stops.
2. **Recursive Case**: The part of the function where it calls itself with modified parameters.

Key Features of Recursion

- **Base Case**: Prevents infinite recursion by terminating the recursive calls when a specific condition is met.
- **Recursive Case**: Breaks the problem into smaller instances and calls the function itself.
- **Stack Usage**: Each recursive call is placed on the call stack, and the function resumes execution when the call returns.

Example: Calculating Factorial Using Recursion

The factorial of a non-negative integer nnn is the product of all positive integers less than or equal to nnn. It is defined as:

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

With 0!=10! = 10!=1.

Recursive Function for Factorial

```
#include <iostream>
using namespace std;

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1; // Base case
    } else {
        return n * factorial(n - 1); // Recursive case
    }
}

int main() {
    int number;

    cout << "Enter a number to calculate its factorial: ";
    cin >> number;

    if (number < 0) {
        cout << "Factorial is not defined for negative numbers." << endl;
    } else {
        cout << "Factorial of " << number << " is " << factorial(number) << endl;
    }

    return 0;
}
```

How the Recursion Works

For n=5n = 5n=5, the execution proceeds as follows:

1. factorial(5)=5×factorial(4)factorial(5) = 5 \times factorial(4)factorial(5)=5×factorial(4)
2. factorial(4)=4×factorial(3)factorial(4) = 4 \times factorial(3)factorial(4)=4×factorial(3)
3. factorial(3)=3×factorial(2)factorial(3) = 3 \times factorial(2)factorial(3)=3×factorial(2)
4. factorial(2)=2×factorial(1)factorial(2) = 2 \times factorial(1)factorial(2)=2×factorial(1)
5. factorial(1)=1factorial(1) = 1factorial(1)=1 (Base case)

The call stack unwinds:

- factorial(2)=2×1=2factorial(2) = 2 \times 1 = 2factorial(2)=2×1=2
- factorial(3)=3×2=6factorial(3) = 3 \times 2 = 6factorial(3)=3×2=6
- factorial(4)=4×6=24factorial(4) = 4 \times 6 = 24factorial(4)=4×6=24
- factorial(5)=5×24=120factorial(5) = 5 \times 24 = 120factorial(5)=5×24=120

Output:

Enter a number to calculate its factorial: 5
Factorial of 5 is 120

Advantages of Recursion

- Simplifies code for problems that are naturally recursive (e.g., tree traversal, divide and conquer).
- Reduces the need for iterative constructs in certain cases.

Disadvantages of Recursion

- Can lead to stack overflow if the recursion depth is too large.
- May be less efficient than iterative solutions due to overhead from multiple function calls.

Iterative vs Recursive Factorial

Recursive Implementation:

```
int factorial(int n) {
   return (n == 0 || n == 1) ? 1 : n * factorial(n - 1);
}
```

Iterative Implementation:

```
int factorial(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

While recursion is elegant and concise, the iterative approach is often more efficient for problems like calculating a factorial. Use recursion judiciously!

(Q4.) What are function prototypes in C++? Why are they used?

Ans: A **function prototype** is a declaration of a function that specifies its name, return type, and parameters but does not include the function body. It provides the compiler with information about the function so that it can be used before it is defined.

Syntax of a Function Prototype

return_type function_name(parameter_list);

- **return_type**: Specifies the type of value the function will return (e.g., int, float, void).
- **function_name**: The name of the function.
- **parameter_list**: Specifies the data types and (optionally) names of the parameters.

Example:

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

Why Are Function Prototypes Used?

1. **Allows Function Calls Before Definitions**:
   - A function prototype informs the compiler about the function's existence and its signature, enabling the function to be called before it is defined.
2. **Error Checking**:
   - Ensures that the function is called with the correct number and type of arguments.
   - If there is a mismatch between the function call and its definition, the compiler raises an error.
3. **Improves Code Organization**:
   - Separates the interface (declaration) from the implementation (definition).
   - Encourages modular programming.

Example Without Function Prototype

This code would result in a compilation error:

```
#include <iostream>
using namespace std;

int main() {
   int result = add(3, 4); // Error: 'add' is not declared
```

```
    return 0;
}

int add(int a, int b) {
    return a + b;
}
```

Example With Function Prototype

cpp
Copy code

```
#include <iostream>
using namespace std;

// Function Prototype
int add(int a, int b);

int main() {
    int result = add(3, 4); // Valid: Compiler knows about 'add'
    cout << "Sum: " << result << endl;
    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;
}
```

Output:

Sum: 7

When Function Prototypes Are Necessary

Function prototypes are necessary when:

- The function is defined after its first use in the program.
- The program is large and functions are organized in separate files.

Rules for Function Prototypes

1. The prototype must match the function definition exactly in terms of:
    - Return type.
    - Number and types of parameters.
2. Parameter names are optional in the prototype but can improve readability.
   Example:

int add(int, int); // Valid

Benefits of Using Function Prototypes

- Ensures type safety by checking parameter types.
- Allows for modular and well-organized code.
- Simplifies working with multiple files in large programs.

5. Arrays and Strings

(Q1.) What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

Ans: An **array** is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow us to store and manipulate multiple values efficiently using a single variable name.

Features of Arrays

1. **Fixed Size**: The size of an array is defined at the time of declaration and cannot be changed dynamically.
2. **Indexed Access**: Elements are accessed using their index, starting from 0.
3. **Homogeneous Data**: All elements in an array must be of the same data type.

Syntax for Declaring an Array

data_type array_name[array_size];

Example:

int numbers[5]; // Declares an integer array of size 5

Types of Arrays

1. Single-Dimensional Arrays

- A single-dimensional array is a linear collection of elements, where each element can be accessed using one index.
- It is often used to store a list of related values like marks, scores, or names.

Syntax:

data_type array_name[size];

Example:

```
#include <iostream>
using namespace std;

int main() {
   int scores[5] = {90, 85, 78, 92, 88};

   // Accessing and displaying array elements
   for (int i = 0; i < 5; i++) {
      cout << "Score " << i + 1 << ": " << scores[i] << endl;
   }

   return 0;
}
```

Output:

```
Score 1: 90
Score 2: 85
Score 3: 78
Score 4: 92
Score 5: 88
```

2. Multi-Dimensional Arrays

- A multi-dimensional array is an array of arrays. It can have two or more dimensions.
- The most common type is the **two-dimensional array**, which is used to represent matrices or tables.

Syntax:

data_type array_name[size1][size2];

Example: Two-Dimensional Array:

```
#include <iostream>
using namespace std;

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

   // Accessing and displaying array elements
   for (int i = 0; i < 2; i++) {
```

```
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
1 2 3
4 5 6
```

Difference Between Single-Dimensional and Multi-Dimensional Arrays

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	A linear collection of elements.	A collection of arrays organized in rows and columns (or more dimensions).
Declaration Syntax	data_type array_name[size];	data_type array_name[size1][size2];
Access Method	Accessed using a single index (array[index]).	Accessed using multiple indices (array[row][col]).
Use Case	Used to store a list of values.	Used to represent grids, tables, or matrices.
Example	int arr[5];	int matrix[3][3];

Higher-Dimensional Arrays

- Arrays with more than two dimensions (e.g., int arr[3][3][3];) are possible but less commonly used in practice.
- They are used in advanced scenarios like 3D modeling, scientific computing, etc.

Key Points

- Arrays provide a way to store multiple elements in a structured and efficient manner.
- Single-dimensional arrays are best for linear data, while multi-dimensional arrays are suited for tabular or grid-like data.

(Q2.) Explain string handling in C++ with examples.

Ans: In C++, strings are sequences of characters used to represent text. The language provides two primary ways to handle strings:

1. **C-style strings** (using character arrays).
2. **C++ Standard Library std::string** (modern and recommended).

1. C-Style Strings

C-style strings are arrays of characters terminated by a null character ('\0'). They are a legacy approach from C and can be manipulated using functions from the <cstring> library.

Declaration and Initialization

```
#include <iostream>
using namespace std;

int main() {
   char str1[20] = "Hello";   // Direct initialization
   char str2[] = {'W', 'o', 'r', 'l', 'd', '\0'}; // Explicit null termination

   cout << "String 1: " << str1 << endl;
   cout << "String 2: " << str2 << endl;

   return 0;
}
```

Common String Functions in <cstring>

- strlen(): Returns the length of the string.
- strcpy(): Copies one string to another.
- strcat(): Concatenates two strings.
- strcmp(): Compares two strings.

Example:

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   char str1[20] = "Hello";
```

```
    char str2[20] = "World";

    // Concatenate strings
    strcat(str1, str2); // str1 = "HelloWorld"
    cout << "Concatenated String: " << str1 << endl;

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

    // Compare strings
    cout << "Comparison: " << strcmp(str1, str2) << endl; // Returns > 0

    return 0;
}
```

2. std::string Class

The std::string class, provided by the Standard Template Library (STL), is a more flexible and user-friendly way to handle strings in C++. It supports dynamic memory management, built-in operations, and overloaded operators.

Declaration and Initialization

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string str1 = "Hello";    // Direct initialization
    string str2("World");     // Constructor initialization

    cout << "String 1: " << str1 << endl;
    cout << "String 2: " << str2 << endl;

    return 0;
}
```

Common Operations

1. **Concatenation**:

   ```
   string str3 = str1 + " " + str2;
   ```

2. **Appending**:

   ```
   str1 += " Everyone!";
   ```

3. **Length**:

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

4. **Accessing Characters**:

```
cout << "First Character: " << str1[0] << endl;
```

5. **Substring**:

```
string sub = str1.substr(0, 5); // Extracts "Hello"
```

6. **Finding Substrings**:

```
size_t pos = str1.find("Everyone");
```

Example:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string str1 = "Hello";
    string str2 = "World";

    // Concatenate strings
    string str3 = str1 + " " + str2;
    cout << "Concatenated String: " << str3 << endl;

    // Append
    str1 += " Everyone!";
    cout << "Appended String: " << str1 << endl;

    // Find substring
    size_t pos = str1.find("Everyone");
    if (pos != string::npos) {
        cout << "'Everyone' found at position: " << pos << endl;
    } else {
        cout << "'Everyone' not found" << endl;
    }

    // Extract substring
    string sub = str3.substr(0, 5);
    cout << "Substring: " << sub << endl;

    return 0;
}
```

Key Differences: C-Style Strings vs. std::string

Feature	C-Style Strings	std::string Class
Memory Management	Manual (fixed size).	Dynamic (automatically managed).
Ease of Use	Requires functions from <cstring>.	Direct operations and methods.
Flexibility	Limited.	Highly flexible.
Safety	Prone to errors (e.g., buffer overflow).	Safer due to dynamic resizing.

Conclusion

While C-style strings are still supported in C++, the std::string class is the preferred way to handle strings due to its simplicity, safety, and functionality. Use std::string for most applications to take advantage of modern C++ features.

(Q3). How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Ans: In C++, arrays can be initialized at the time of declaration or assigned values later in the program. Initialization can be done for both **one-dimensional (1D)** and **two-dimensional (2D)** arrays.

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

Basic Syntax

data_type array_name[size] = {value1, value2, ..., valueN};

Examples

1. **Complete Initialization**:

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

2. **Partial Initialization**:
   - If fewer values are provided than the size, the remaining elements are automatically initialized to 0.

   int numbers[5] = {1, 2}; // {1, 2, 0, 0, 0}

3. **Default Initialization**:
    - If no values are provided, the elements are **uninitialized** (contain garbage values).

```
int numbers[5]; // Uninitialized
```

4. **Automatic Size Detection**:
    - The compiler can infer the size from the number of elements in the initializer list.

```
cpp
Copy code
int numbers[] = {1, 2, 3}; // Size is automatically set to 3
```

Example Program: 1D Array

```
#include <iostream>
using namespace std;

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

   cout << "1D Array Elements:" << endl;
   for (int i = 0; i < 5; i++) {
      cout << numbers[i] << " ";
   }

   return 0;
}
```

Output:

```
1D Array Elements:
10 20 30 40 50
```

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

Basic Syntax

```
cpp
Copy code
data_type array_name[rows][columns] = {
   {value1, value2, ..., valueN},
   {value1, value2, ..., valueN},
   ...
};
```

Examples

1. **Complete Initialization**:

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

2. **Partial Initialization**:
    - If fewer values are provided, the remaining elements are initialized to 0.

int matrix[2][3] = {{1, 2}, {4}}; // {{1, 2, 0}, {4, 0, 0}}

3. **Automatic Size Detection for Columns**:
    - The number of columns can be omitted if it can be inferred from the initializer list.

int matrix[][3] = {{1, 2, 3}, {4, 5, 6}}; // Rows are inferred

4. **Default Initialization**:
    - If no initializer is provided, the elements contain garbage values.

int matrix[2][3]; // Uninitialized

Example Program: 2D Array

```
#include <iostream>
using namespace std;

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

   cout << "2D Array Elements:" << endl;
   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 3; j++) {
         cout << matrix[i][j] << " ";
      }
      cout << endl;
   }

   return 0;
}
```

Output:

```
2D Array Elements:
1 2 3
4 5 6
```

Key Points

1. **Memory Layout**:
    - Arrays in C++ are stored in **row-major order** (for 2D arrays, all elements of a row are stored consecutively in memory).
2. **Initialization Rules**:
    - For 1D arrays, uninitialized elements default to 0 if partially initialized.
    - For 2D arrays, elements default to 0 in partially initialized rows or columns.

Comparison of 1D and 2D Array Initialization

Feature	1D Array	2D Array
Syntax	data_type array[size]	data_type array[rows][columns]
Example (Complete)	{1, 2, 3}	{{1, 2, 3}, {4, 5, 6}}
Example (Partial)	{1, 2}	{{1}, {4}}
Automatic Size Detection	Size inferred from elements.	Rows or columns can be inferred.

Arrays provide a structured way to store and manipulate data, and their initialization ensures proper usage and error prevention.

(Q4.) Explain string operations and functions in C++.

Ans: In C++, strings can be manipulated using either **C-style strings** (character arrays) or the **std::string class** from the Standard Template Library (STL). The std::string class provides a rich set of operations and functions to handle strings efficiently.

String Operations with std::string

1. **Concatenation**:
    - Combine two strings using the + operator or the += operator.

```
string str1 = "Hello";
string str2 = "World";
string result = str1 + " " + str2; // "Hello World"
```

2. **Appending**:
    - Add a string or character to the end using the append() function.

```
string str = "Hello";
str.append(" World"); // "Hello World"
```

3. **Length of a String**:
    - Get the number of characters in a string using length() or size().

```
string str = "Hello";
cout << str.length(); // 5
```

4. **Accessing Characters**:
    - Access individual characters using the subscript operator [] or the at() method.

```
string str = "Hello";
cout << str[0];       // 'H'
cout << str.at(1);    // 'e'
```

5. **Substring**:
    - Extract a portion of a string using the substr() function.

```
string str = "Hello World";
string sub = str.substr(0, 5); // "Hello"
```

6. **Finding Substrings**:
    - Search for a substring using find() or rfind().

```
string str = "Hello World";
size_t pos = str.find("World"); // Returns 6
```

7. **String Comparison**:
    - Compare two strings using compare(), ==, or <.

```
string str1 = "Hello";
string str2 = "World";
if (str1 < str2) {
   cout << "str1 is less than str2";
}
```

8. **Replace Substring**:
    - Replace part of a string with another string using replace().

```
string str = "Hello World";
str.replace(6, 5, "C++"); // "Hello C++"
```

9. **Inserting Characters**:
    - Insert characters or strings into a string using insert().

```
string str = "Hello";
str.insert(5, " World"); // "Hello World"
```

10. **Erasing Characters**:
    - Remove characters from a string using erase().

```
string str = "Hello World";
str.erase(5, 6); // "Hello"
```

11. **Clearing a String**:
    - Clear the content of a string using clear().

```
cpp
Copy code
string str = "Hello";
str.clear(); // Empty string
```

Common String Functions in C++

Function	Description
length() or size()	Returns the number of characters in the string.
empty()	Checks if the string is empty.
append()	Appends a string or character to the end of the string.
find()	Finds the first occurrence of a substring or character.
rfind()	Finds the last occurrence of a substring or character.
substr()	Extracts a substring from the string.
replace()	Replaces a portion of the string with another string.
insert()	Inserts a string or character at a specified position.
erase()	Erases a portion of the string.
clear()	Clears the content of the string.
compare()	Compares two strings lexicographically.
c_str()	Returns a C-style string (null-terminated character array).
swap()	Swaps the content of two strings.

Examples

String Manipulation

```
#include <iostream>
#include <string>
using namespace std;

int main() {
```

```
    string str1 = "Hello";
    string str2 = "World";

    // Concatenate
    string result = str1 + " " + str2;
    cout << "Concatenated: " << result << endl;

    // Find substring
    size_t pos = result.find("World");
    cout << "'World' found at: " << pos << endl;

    // Replace substring
    result.replace(pos, 5, "C++");
    cout << "After Replace: " << result << endl;

    // Extract substring
    string sub = result.substr(0, 5);
    cout << "Substring: " << sub << endl;

    // Clear string
    result.clear();
    cout << "Is string empty? " << result.empty() << endl;

    return 0;
}
```

Output:

```
Concatenated: Hello World
'World' found at: 6
After Replace: Hello C++
Substring: Hello
Is string empty? 1
```

Key Differences Between std::string and C-Style Strings

Feature	C-Style Strings	std::string
Memory Management	Manual, fixed size.	Automatic, dynamic size.
Ease of Use	Requires manual manipulation.	Rich set of built-in functions.
Safety	Prone to buffer overflows.	Safer and easier to handle.
Preferred Usage	Legacy code or performance-critical.	Modern C++ programming.

Conclusion

The std::string class provides a comprehensive and convenient way to handle strings in C++. It is safer, more flexible, and feature-rich compared to traditional C-style strings. For modern C++ programming, std::string is the recommended approach.

6. Introduction to Object-Oriented Programming

(Q1.) Explain the key concepts of Object-Oriented Programming (OOP).

Ans: Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**, which represent real-world entities. Objects encapsulate data (attributes) and behavior (methods), enabling modular, reusable, and maintainable code. OOP is widely used in languages like C++, Java, and Python.

Here are the **four key principles** of OOP:

1. Encapsulation

Encapsulation is the process of bundling data (attributes) and methods (functions) that operate on the data into a single unit, called an **object**. It also involves restricting direct access to some components of an object to maintain its integrity.

Features:

- **Access Modifiers**:
  - **private:** Members are accessible only within the same class.
  - **protected:** Members are accessible within the same class and its derived classes.
  - **public:** Members are accessible from anywhere in the program.
- Provides **data hiding** and prevents unauthorized access.

Example:

```
#include <iostream>
using namespace std;

class Person {
private:
   string name; // Private attribute
   int age;
```

```
public:
    // Setter method
    void setDetails(string n, int a) {
        name = n;
        age = a;
    }

    // Getter method
    void displayDetails() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p;
    p.setDetails("Alice", 25);
    p.displayDetails();

    return 0;
}
```

2. Inheritance

Inheritance allows a class (called a **derived class**) to inherit attributes and methods from another class (called a **base class**). It promotes code reuse and establishes a hierarchical relationship between classes.

Features:

- **Base Class**: The parent class whose members are inherited.
- **Derived Class**: The child class that inherits members.
- Types:
    - Single Inheritance
    - Multilevel Inheritance
    - Multiple Inheritance
- Access specifiers determine how base class members are inherited:
    - public: Public and protected members remain accessible.
    - protected: Public and protected members become protected.
    - private: Public and protected members become private.

Example:

```
#include <iostream>
using namespace std;

class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
```

```
    }
};

class Dog : public Animal { // Inherits from Animal
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};

int main() {
    Dog d;
    d.eat(); // Inherited from Animal
    d.bark();

    return 0;
}
```

3. Polymorphism

Polymorphism allows objects to take on many forms. It enables the same function or operator to behave differently depending on the context.

Types:

1. **Compile-Time Polymorphism (Static Binding)**:
    - Achieved using **function overloading** and **operator overloading**.
2. **Run-Time Polymorphism (Dynamic Binding)**:
    - Achieved using **virtual functions** and **function overriding**.

Example: Function Overloading (Compile-Time Polymorphism):

```
#include <iostream>
using namespace std;

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(5, 10) << endl;       // Calls int version
    cout << calc.add(3.5, 2.5) << endl;    // Calls double version
```

```
    return 0;
}
```

Example: Virtual Function (Run-Time Polymorphism):

```
#include <iostream>
using namespace std;

class Shape {
public:
    virtual void draw() { // Virtual function
        cout << "Drawing a generic shape." << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle." << endl;
    }
};

int main() {
    Shape* shape; // Pointer to base class
    Circle circle;

    shape = &circle;
    shape->draw(); // Calls Circle's draw() due to dynamic binding

    return 0;
}
```

4. Abstraction

Abstraction involves hiding complex implementation details and showing only the essential features of an object. It is achieved using abstract classes and interfaces.

Features:

- **Abstract Classes**:
    - Contain at least one pure virtual function (virtual returnType functionName() = 0;).
- Cannot instantiate abstract classes directly.

Example:

```
#include <iostream>
using namespace std;
```

```
class Shape {
public:
   virtual void draw() = 0; // Pure virtual function
};

class Rectangle : public Shape {
public:
   void draw() override {
      cout << "Drawing a rectangle." << endl;
   }
};

int main() {
   Shape* shape;
   Rectangle rect;

   shape = &rect;
   shape->draw(); // Calls Rectangle's draw()

   return 0;
}
```

Benefits of OOP

1. **Modularity**: Code is organized into objects, making it easier to manage and maintain.
2. **Reusability**: Inheritance promotes reuse of existing code.
3. **Scalability**: Polymorphism enables flexible and extensible designs.
4. **Security**: Encapsulation restricts access to sensitive data.

Conclusion

OOP principles—encapsulation, inheritance, polymorphism, and abstraction—enable efficient, organized, and scalable programming. These concepts form the foundation of modern software development.

(Q2.) What are classes and objects in C++? Provide an example.

Ans: In C++, **classes** and **objects** are fundamental concepts of **Object-Oriented Programming (OOP)**. They provide a way to model real-world entities and their behaviors in a structured and modular way.

1. Class

A **class** is a blueprint or template for creating objects. It defines a data structure and the functions (methods) that operate on the data. A class encapsulates data and functions into a single unit.

Key Components of a Class:

- **Data Members (Attributes)**: Variables that hold the state or properties of an object.
- **Member Functions (Methods)**: Functions that define the behavior of the object, allowing it to perform actions on its data.

Syntax of a Class:

```
class ClassName {
public:     // Access specifier, can be public, private, or protected
   // Data members (attributes)
   dataType memberVariable1;
   dataType memberVariable2;

   // Member functions (methods)
   void functionName() {
      // function body
   }
};
```

2. Object

An **object** is an instance of a class. It is created based on the class definition and represents a specific entity of the class with its own set of attributes and behavior.

Creating an Object:

An object is created by using the class name followed by an object name.

```
ClassName objectName;
```

Example: Class and Object in C++

```
#include <iostream>
using namespace std;

class Car {
private:
   // Data members (attributes)
   string brand;
   string model;
   int year;
```

```
public:
    // Member functions (methods)
    // Constructor to initialize attributes
    Car(string b, string m, int y) {
        brand = b;
        model = m;
        year = y;
    }

    // Function to display car details
    void displayDetails() {
        cout << "Car Brand: " << brand << endl;
        cout << "Car Model: " << model << endl;
        cout << "Car Year: " << year << endl;
    }
};

int main() {
    // Creating objects (instances of the Car class)
    Car car1("Toyota", "Corolla", 2020); // object car1
    Car car2("Ford", "Mustang", 2022);  // object car2

    // Accessing member functions using the objects
    cout << "Details of car1:" << endl;
    car1.displayDetails();

    cout << "\nDetails of car2:" << endl;
    car2.displayDetails();

    return 0;
}
```

Explanation of the Example:

1. **Class Definition**:
    - The Car class has three private data members: brand, model, and year.
    - The constructor Car(string b, string m, int y) initializes these attributes when an object is created.
    - The method displayDetails() prints the details of the car.
2. **Object Creation**:
    - In the main() function, two objects of the Car class are created: car1 and car2.
    - Each object is initialized with specific values for brand, model, and year.
3. **Accessing Methods**:
    - The displayDetails() function is called on both car1 and car2 objects to display their respective details.

Output:

Details of car1:
Car Brand: Toyota
Car Model: Corolla
Car Year: 2020

Details of car2:
Car Brand: Ford
Car Model: Mustang
Car Year: 2022

Key Points:

- **Class**: Defines the structure (attributes) and behavior (methods) of an object.
- **Object**: A specific instance of a class that contains actual values for the class's attributes.
- **Encapsulation**: The class encapsulates data and methods into a single unit.
- **Constructor**: A special method used to initialize objects when they are created.

Classes and objects are central to object-oriented design, allowing developers to create reusable and maintainable code.

(Q3.) What is inheritance in C++? Explain with an example.

Ans: **Inheritance** is one of the fundamental concepts in **Object-Oriented Programming (OOP)**. It allows a class (known as a **derived class**) to inherit attributes and methods from another class (known as a **base class**). This promotes code reuse and establishes a hierarchical relationship between classes.

With inheritance, you can create a new class that has the functionality of an existing class and add or modify certain behaviors.

Key Features of Inheritance:

1. **Base Class (Parent Class)**: The class whose properties and methods are inherited.
2. **Derived Class (Child Class)**: The class that inherits properties and methods from the base class and can also have additional or modified features.
3. **Access Specifiers**:
    - public: Members of the base class remain accessible as public in the derived class.
    - protected: Members of the base class become protected in the derived class.
    - private: Members of the base class become private in the derived class.

4. **Types of Inheritance**:
    - **Single Inheritance**: One class inherits from a single base class.
    - **Multiple Inheritance**: A class inherits from more than one base class.
    - **Multilevel Inheritance**: A class is derived from another derived class.
    - **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.
    - **Hybrid Inheritance**: A combination of multiple types of inheritance.

Syntax of Inheritance in C++:

```
class BaseClass {
   // Base class members
};

class DerivedClass : accessSpecifier BaseClass {
   // Derived class members
};
```

Example of Inheritance:

Let's consider an example of **single inheritance**, where a Vehicle class is the base class, and a Car class is the derived class.

```
#include <iostream>
using namespace std;

// Base Class (Parent Class)
class Vehicle {
public:
   string brand;
   int year;

   // Base class constructor
   Vehicle(string b, int y) {
      brand = b;
      year = y;
   }

   // Base class method
   void displayDetails() {
      cout << "Brand: " << brand << endl;
      cout << "Year: " << year << endl;
   }
};

// Derived Class (Child Class)
class Car : public Vehicle {  // Inheriting from Vehicle
```

```
public:
    string model;

    // Constructor of derived class
    Car(string b, int y, string m) : Vehicle(b, y) {
        model = m;
    }

    // Method of the derived class
    void displayCarDetails() {
        displayDetails();  // Calling base class method
        cout << "Model: " << model << endl;
    }
};

int main() {
    // Creating an object of the derived class
    Car car1("Toyota", 2020, "Corolla");

    // Calling method of derived class
    car1.displayCarDetails();

    return 0;
}
```

Explanation:

1. **Base Class (Vehicle)**:
    - The Vehicle class has two attributes (brand and year) and a method displayDetails() to print these details.
    - It also has a constructor to initialize these attributes.
2. **Derived Class (Car)**:
    - The Car class inherits from the Vehicle class using the public access specifier. This means the public members of Vehicle will be accessible in Car.
    - The Car class has its own attribute model and a method displayCarDetails() that prints the details of the car by calling the displayDetails() method from the base class.
3. **Constructor of Derived Class**:
    - The constructor of Car calls the constructor of Vehicle using the : Vehicle(b, y) syntax, passing values for brand and year.
4. **Object Creation**:
    - In main(), we create an object car1 of type Car and initialize it with brand, year, and model.
    - The displayCarDetails() method is called on car1, which in turn calls the displayDetails() method from the base class and prints all the car details.

Output:

Brand: Toyota
Year: 2020
Model: Corolla

Types of Inheritance in C++ (with Examples)

1. Single Inheritance:

A derived class inherits from one base class.

```
class Base {
   // Base class members
};

class Derived : public Base {
   // Derived class members
};
```

2. Multilevel Inheritance:

A class inherits from a derived class, forming a chain.

```
class Base {
   // Base class members
};

class Intermediate : public Base {
   // Intermediate class members
};

class Derived : public Intermediate {
   // Derived class members
};
```

3. Multiple Inheritance:

A class inherits from more than one base class.

```
class Base1 {
   // Base1 class members
};

class Base2 {
   // Base2 class members
};

class Derived : public Base1, public Base2 {
```

```
    // Derived class members
};
```

4. Hierarchical Inheritance:

Multiple classes inherit from a single base class.

```
class Base {
    // Base class members
};

class Derived1 : public Base {
    // Derived1 class members
};

class Derived2 : public Base {
    // Derived2 class members
};
```

Conclusion:

Inheritance in C++ allows for code reuse and promotes a hierarchical relationship between classes. It provides a way to extend the functionality of existing classes without modifying them. By using inheritance, you can create a new class that inherits properties and behaviors from an existing class, which helps in making the code more modular, extensible, and maintainable.

(Q4.) 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 is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit, called a **class**. Encapsulation helps in protecting the internal state of an object from unauthorized access and modification, ensuring data integrity and hiding implementation details.

In simpler terms, encapsulation allows an object to control its own data and prevents external code from directly accessing and modifying that data. It achieves **data hiding** by making certain attributes and methods private or protected, while exposing only necessary parts through public methods.

How Encapsulation is Achieved in C++:

Encapsulation is achieved in C++ using **access specifiers** and **getter and setter functions**.

Key Components:

1. **Access Specifiers**:
   - **public**: Members are accessible from outside the class.
   - **private**: Members are not accessible from outside the class (default access level for data members).
   - **protected**: Members are not accessible outside the class, except for derived classes.
2. **Getter Functions**: Functions used to access the value of private data members.
3. **Setter Functions**: Functions used to set or modify the value of private data members.

Example of Encapsulation in C++:

```
#include <iostream>
using namespace std;

class BankAccount {
private:
    // Private data members (data is hidden from outside)
    string accountHolder;
    double balance;

public:
    // Constructor to initialize the data members
    BankAccount(string name, double initialBalance) {
        accountHolder = name;
        balance = initialBalance;
    }

    // Getter function to access the balance (data retrieval)
    double getBalance() {
        return balance;
    }

    // Setter function to deposit an amount (data modification)
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            cout << "Deposited: $" << amount << endl;
        } else {
            cout << "Invalid deposit amount!" << endl;
        }
    }

    // Setter function to withdraw an amount (data modification)
    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
```

```
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
        } else {
            cout << "Insufficient balance or invalid amount!" << endl;
        }
    }

    // Function to display account details
    void displayAccountDetails() {
        cout << "Account Holder: " << accountHolder << endl;
        cout << "Balance: $" << balance << endl;
    }
};

int main() {
    // Creating an object of BankAccount
    BankAccount account("Alice", 1000.0);

    // Displaying account details
    account.displayAccountDetails();

    // Using setter methods to modify data
    account.deposit(500.0);
    account.withdraw(200.0);

    // Displaying updated account details
    account.displayAccountDetails();

    return 0;
}
```

Explanation of the Example:

1. **Private Data Members**:
    - The accountHolder and balance are private data members of the BankAccount class. This means they cannot be accessed directly from outside the class.
2. **Getter and Setter Methods**:
    - **getBalance()**: A getter function is used to retrieve the value of the private balance data member.
    - **deposit(double amount)**: A setter function is used to modify the balance by adding the deposited amount.
    - **withdraw(double amount)**: A setter function to modify the balance by subtracting the withdrawal amount. It also performs validation to ensure the withdrawal is valid.
3. **Encapsulation in Action**:
    - The private data is not directly accessed from outside the class. Instead, all interactions with the data are done through public setter and getter methods.

    - This encapsulation ensures that the balance is protected from being set to invalid or inconsistent values (e.g., negative values).
4. **Data Hiding**:
    - External code does not know or need to know how the balance is stored or managed. It only interacts with the public methods that provide a controlled interface to the class's functionality.

Output:

Account Holder: Alice
Balance: $1000
Deposited: $500
Withdrawn: $200
Account Holder: Alice
Balance: $1300

Advantages of Encapsulation:

1. **Data Protection**:
    - By restricting direct access to the data and providing controlled access through setter/getter methods, encapsulation helps protect the internal state of the object from accidental or malicious changes.
2. **Modularity**:
    - The class can be treated as a black box, with external code interacting only through well-defined interfaces (methods). This promotes a modular design and easier debugging.
3. **Code Maintainability**:
    - Changes to the internal workings of the class (e.g., how data is stored or validated) can be made without affecting the external code that interacts with the class, as long as the public interface remains consistent.
4. **Increased Flexibility**:
    - Encapsulation allows for validation and additional logic to be implemented in setter methods, providing more control over how data is manipulated. For example, ensuring that deposits and withdrawals follow business rules.

Conclusion:

Encapsulation is a powerful concept in C++ that allows you to bundle data and methods together within a class, controlling how the data is accessed and modified. By using access specifiers (public, private, protected) and providing getter and setter methods, you can ensure that an object's internal state is protected and only exposed through controlled interfaces, leading to more robust, maintainable, and secure code.