MODULE 3: Introduction to OOP Programming

1. Introduction to C++

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

Ans:

Difference between Procedural Programming vs Object-Oriented Programming is as mentioned below:-

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called *functions*.	In object-oriented programming, the program is divided into small parts called *objects*.
Procedural programming follows a *top-down approach*.	Object-oriented programming follows a *bottom-up approach*.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way of hiding data so it is *less secure*.	Object-oriented programming provides data hiding so it is *more secure*.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriented programming, the concept of data hiding and inheritance is used.

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the function is more important than the data.	In object-oriented programming, data is more important than function.
Procedural programming is based on the *unreal world*.	Object-oriented programming is based on the *real world*.
Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
Examples include C, FORTRAN, Pascal, Basic, and many more.	Examples include C++, Java, Python, C#, and many more.

Q.2] **List and explain the main advantages of OOP over POP.**

Ans:-

OOP offers several advantages over POP , including code reusability, improved modularity, and enhanced security through data hiding. OOP's ability to encapsulate data and behavior within objects makes it easier to manage and maintain complex software. Additionally, OOP's inheritance and polymorphism features further contribute to code reuse and flexibility.

Here's a more detailed look at the advantages:

Code Reusability:

OOP promotes code reusability through inheritance, allowing developers to create new classes (objects) based on existing ones, reducing redundancy and effort.

Modularity:

OOP's object-based structure promotes modularity, making code easier to organize, understand, and maintain. Complex problems can be broken down into smaller, manageable components (objects).

Encapsulation:

OOP's encapsulation feature protects data by hiding it within objects, restricting access to it from outside the object's methods. This improves data security and reduces the risk of accidental data corruption.

Inheritance:

Inheritance allows classes to inherit properties and behaviors from parent classes, further promoting code reuse and simplifying development.

Polymorphism:

Polymorphism allows objects of different classes to be treated uniformly, providing flexibility and reducing code complexity.

Easier Maintenance:

OOP's modular structure makes it easier to identify and fix errors, as changes can be localized to specific objects.

Improved Problem Solving:

OOP's object-oriented approach aligns well with real-world problem-solving, making it easier to model complex systems and their interactions.

Faster Development:

Code reusability and modularity contribute to faster development cycles, allowing developers to leverage existing code and libraries.

Q.3] **Explain the steps involved in setting up a C++ development environment.**

Ans:-

Setting up a C++ development environment involves several key steps, depending on your operating system and preferences for tools and editors.

1. Choose an IDE or Text Editor

You need a code editor or an integrated development environment (IDE) to write and manage your C++ code.

- **Popular IDEs:**
    - Visual Studio (Windows)
    - CLion (Cross-platform)
    - Code::Blocks (Cross-platform)
- **Popular Text Editors:**
    - Visual Studio Code (VS Code)
    - Sublime Text

- Atom

2. Install a C++ Compiler

C++ source code must be compiled into an executable. Choose a compiler appropriate for your OS:

- Windows: Install MinGW or MSVC (comes with Visual Studio).
- macOS: Use Xcode Command Line Tools (xcode-select --install).
- Linux: Install GCC via package manager (sudo apt install g++ for Ubuntu/Debian).

3. Set Up the Build Environment

Ensure the compiler is available in your system's PATH so you can run it from the terminal or command prompt.

- Windows (MinGW example):
  - Add C:\MinGW\bin to your environment variable PATH.
- Check Compiler:
- g++ --version

4. Configure the IDE or Editor

Link your IDE or editor to the compiler:

- **Visual Studio Code:**
  - Install C/C++ extension (by Microsoft).
  - Create or configure tasks.json and launch.json to define build and debug tasks.
- **CLion:** Automatically detects compilers and builds systems.
- **Code::Blocks:** Select the compiler in Settings > Compiler.

5. Write and Run Your First Program

Create a simple main.cpp file:

```
#include <iostream>

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

Compile and run:

- **Using terminal:**
- g++ main.cpp -o main

- ./main
- **Using IDE:** Press the build and run button.

6. Optional: Set Up a Build System

For larger projects, you may want a build system:

- Make (Makefile)
- CMake (Cross-platform, used by CLion)
- Ninja, Meson, etc.

***Q.4]* What are the main input/output operations in C++? Provide examples.**

Ans:

In C++, input and output (I/O) operations are primarily handled using **streams**, which are part of the iostream library. The main I/O operations include:

1. Output using std::cout

- **Purpose**: To display output to the console (standard output).
- **Header**: #include <iostream>

Example:

```
#include <iostream>
int main() {
   std::cout << "Hello, World!" << std::endl;
   std::cout << "The number is: " << 42 << std::endl;
   return 0;
}
```

- std::endl moves the cursor to the next line.
- << is the insertion operator.

2. Input using std::cin

- **Purpose**: To get input from the user (standard input).
- **Header**: #include <iostream>

Example:

```
#include <iostream>
int main() {
   int age;
   std::cout << "Enter your age: ";
   std::cin >> age;  // User inputs a value
   std::cout << "You entered: " << age << std::endl;
   return 0;
}
```

- >> is the extraction operator.

3. Input/Output with std::getline()

- **Purpose**: To read a full line of text, including spaces.
- **Useful when**: Reading strings with spaces (e.g., full names or sentences).

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;
}
```

4. File I/O with fstream

- **Header**: #include <fstream>
- **Classes**:
    - std::ifstream — input file stream (read)
    - std::ofstream — output file stream (write)
    - std::fstream — both input and output

Example :Writing to a file:-

```
#include <fstream>
#include <iostream>
 int main() {
    std::ofstream outFile("output.txt");
    if (outFile.is_open()) {
        outFile << "Writing to a file in C++!" << std::endl;
        outFile.close();
    }
    return 0;
}
```

Example (Reading from a file):

```
#include <fstream>
#include <iostream>
#include <string>
int main() {
    std::ifstream inFile("output.txt");
    std::string line;
    if (inFile.is_open()) {
        while (std::getline(inFile, line)) {
            std::cout << line << std::endl;
        }
        inFile.close();
    }
    return 0;
}
```

2. **Variables, Data Types, and Operators**

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

Ans:

C++ provides a wide range of data types that are used to declare variables, functions, and more. These can be broadly categorized into fundamental (built-in) and user-defined types. Here's a breakdown with examples:

1. Fundamental (Built-in) Data Types

a. Integer Types

Used to store whole numbers.

Type	Size	Example
int	4 bytes	int x = 10;
short	2 bytes	shorts = 1000;
long	4 or 8 bytes	long l = 100000;
long long	8 bytes	long long big = 1000000000;
unsigned	varies	unsigned int u = 50;

b. Floating Point Types

Used to store numbers with decimal points.

Type	Size (typically)	Example
float	4 bytes	float pi = 3.14f;
double	8 bytes	double e = 2.71828;
long double	12–16 bytes	long double precise = 1.23456789L;

c. Character Type

Stores a single character.

Type	Size	Example
char	1 byte	char ch = 'A';

d. Boolean Type

Stores true or false.

Type	Size	Example
bool	1 byte	bool isReady = true;

2. Derived Types

These are based on fundamental types.

Examples:

- **Arrays**: int arr[5];
- **Pointers**: int* ptr = &x;

- **Functions**: int sum(int a, int b);
- **References**: int& ref = x;

3. User-Defined Types

You can create your own data types using:

a. Struct

```
struct Person {
    std::string name;
    int age;
};
Person p1 = {"Alice", 30};
```

b. Class

```
class Car {
public:
    std::string brand;
    void drive() {
        std::cout << "Driving " << brand << std::endl;
    }
};
```

c. Union

Stores only one member at a time (saves memory).

```
union Data {
    int i;
    float f;  };
```

d. Enum

Used for enumerating constants.

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

Type Modifiers

These can change the size or sign of basic types:

Modifier	Description
signed	Can be positive or negative (default)
unsigned	Only positive values
short	Smaller range
long	Larger range

Example:

unsigned int score = 100;

long long bigNum = 1234567890;

Q.2] Explain the difference between implicit and explicit type conversion in C++.

Ans:-

In C++, type conversion (also known as type casting) is the process of converting a value from one data type to another. This can happen automatically (implicitly) or manually (explicitly).

1. Implicit Type Conversion (Type Promotion)

- Also called automatic type conversion, it happens when C++ automatically converts one data type to another without user's intervention, usually to avoid data loss or to match types in expressions.

Example:

```
int a = 5;
double b = a;  // 'a' is implicitly converted from int to double
```

Common cases where implicit conversion occurs:

- Assigning a smaller type to a larger type (e.g., int to float)
- In expressions involving mixed types
- Function calls with arguments of different but compatible types

Example with expression:

```
int i = 10;
float f = 2.5;
float result = i + f;  // 'i' is implicitly converted to float
```

2. Explicit Type Conversion (Type Casting):

This is when you manually convert a value from one type to another using type casting syntax. It's used when:

- You want more control over conversions
- Prevent unwanted implicit conversions
- Reduce data loss

C++-Style Cast:

double pi = 3.14;

int x = static_cast<int>(pi); // pi is explicitly cast to int (x = 3)

C-Style Cast (less preferred):

int x = (int)pi;

Example:

int a = 10;

int b = 3;

double result = static_cast<double>(a) / b; // result = 3.333.

- Prefer static_cast, const_cast, reinterpret_cast, and dynamic_cast over C-style casting.
- Avoid relying too much on implicit conversion if it might reduce code clarity or cause precision issues.

Q.3] What are the different types of operators in C++? Provide examples of each.

Ans:- Operators:

In C++, operators are special symbols that perform operations on variables and values. They are classified based on their functionality. Below are the main types of operators in C++ along with examples:

1. Arithmetic Operators

Used to perform basic mathematical operations.

Operator	Description	Example	Result
+	Addition	a + b	Adds a and b
-	Subtraction	a - b	Subtracts b from a
*	Multiplication	a * b	Multiplies a and b
/	Division	a / b	Divides a by b

Operator	Description	Example	Result
%	Modulus	a % b	Remainder of a / b

Example:

```
int a = 10, b = 3;
std::cout << a + b << ", " << a % b << std::endl;  // Output: 13, 1
```

2. Relational (Comparison) Operators

Used to compare two values and return a boolean result.

Operator	Description	Example	Result
==	Equal to	a == b	true if equal
!=	Not equal to	a != b	true if not equal
>	Greater than	a > b	true if a > b
<	Less than	a < b	true if a < b
>=	Greater than or equal	a >= b	true if a ≥ b
<=	Less than or equal	a <= b	true if a ≤ b

3. Logical Operators

Used to combine multiple conditions (return true or false).

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
!	Logical NOT	!(a == b)

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	x = 5

Operator	Description	Example
+=	Add and assign	x += 5 (x = x + 5)
-=	Subtract and assign	x -= 3
*=	Multiply and assign	x *= 2
/=	Divide and assign	x /= 4
%=	Modulus and assign	x %= 3

5. Increment/Decrement Operators

Used to increase or decrease a variable's value by 1.

Operator	Description	Example
++	Increment	++x, x++
--	Decrement	--x, x--

Pre-increment (++x) changes the value before using it, while post-increment (x++) uses it before changing.

6. Bitwise Operators

Operate on binary bits of integers.

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left shift	a << 2
>>	Right shift	a >> 2

7. Conditional (Ternary) Operator

A shorthand for if-else.

Syntax:

condition ? expression_if_true : expression_if_false;

Example:

int x = 10, y = 20;

int max = (x > y) ? x : y;

Q.4] 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 improve code readability, reliability, and safety.

1. Constants in C++

Purpose:

Constants are used to define fixed values that should not be modified after initialization. This helps prevent accidental changes and makes code easier to understand.

Types of Constants:

a. const Keyword

- Declares a variable as constant.
- Its value must be initialized when declared and cannot be changed.

const int MAX_SCORE = 100;

MAX_SCORE = 110;

b. constexpr Keyword (C++11 and later)

- Evaluated at **compile time**.
- Used for **constant expressions** and allows better optimization.

constexpr int SIZE = 10;

int arr[SIZE]; // Valid

c. #define Preprocessor Directive

- A macro that replaces text before compilation.
- Less type-safe than const.

#define PI 3.14

2. Literals in C++

Purpose:

Literals are **fixed values** used directly in the code. They represent constant values of basic data types.

Types of Literals:

a. Integer Literals

int x = 100;

long y = 100000L;

unsigned int z = 42U;

b. Floating-point Literals

float pi = 3.14f;

double e = 2.718;

long double precise = 1.23L;

c. Character Literals

char grade = 'A';

d. String Literals

const char* name = "Alice";

e. Boolean Literals

bool isOpen = true;

bool isClosed = false;

f. Escape Sequences in String/Char Literals

Used to represent special characters:

Escape	Meaning
\n	Newline
\t	Tab
\\	Backslash
\"	Double quote

Example Program Using Constants and Literals:

```
#include <iostream>
const double PI = 3.14159;       // Constant
constexpr int RADIUS = 5;        // Compile-time constant
```

```
int main() {
    double area = PI * RADIUS * RADIUS;
    std::cout << "Area of circle: " << area << std::endl;
    std::cout << "Hello\nWorld" << std::endl;  // String literal with newline escape
    return 0;   }
```

3. **Control Flow Statements:**

[Q5]. What are conditional statements in C++? Explain the if-else and switch statements.

In C++, conditional statements allow you to control the flow of your program based on certain conditions. The two primary types of conditional statements are the if-else and switch statements.

1. if-else Statement

The if-else statement evaluates a condition and executes a block of code if the condition is true; otherwise, it executes another block of code.

Syntax if-else:-

```
if (condition) {
    // Code for true condition
} else {
    // Code for false condition
}
```

Syntax elseif :-

```
if (condition1) {
    // Code for condition1
} else if (condition2) {
    // Code for condition2
} else {
    // Code if none of the above conditions are true
}
```

2. switch Statement

The switch statement is used to execute one out of multiple possible blocks of code based on the value of an expression. It is particularly useful when you have many conditions to check against a single variable.

Syntax:

```
switch (expression) {
   case value1:
      // Code to execute if expression == value1
      break;
   case value2:
      // Code to execute if expression == value2
      break;
   default:
      // Code to execute if expression doesn't match any case
}
```

[Q6]. What is the difference between for, while, and do-while loops in C++?

In C++, loops are control structures that allow you to execute a block of code repeatedly under certain conditions. The three primary types of loops are for, while, and do-while. Each has its unique syntax and use cases.

1. for Loop

The for loop is ideal when the number of iterations is known beforehand. It combines initialization, condition-checking, and iteration in a single line, making it concise and efficient.

Syntax:

```
for (initialization; condition; update) {
   // Loop body
}
```

2. while Loop

The while loop is suitable when the number of iterations is not known and depends on a condition evaluated before each iteration. If the condition is false initially, the loop body may not execute at all.

Syntax:

```
while (condition) {
   // Loop body
}
```

3. do-while Loop

The do-while loop guarantees that the loop body executes at least once, as the condition is checked after the loop's body. This is useful when the loop should run at least once regardless of the condition.

Syntax:

```
do {
   // Loop body
} while (condition);
```

Feature	for Loop	while Loop	do-while Loop
Condition Check	Before each iteration	Before each iteration	After each iteration
Guaranteed Run	No, if condition is false initially	No, if condition is false initially	Yes, executes at least once
Use Case	Known number of iterations	Unknown number of iterations	Ensure at least one execution
Initialization	Inside the loop header	Before the loop starts	Before the loop starts
Update Expression	Inside the loop header	Inside the loop body	Inside the loop body

Q7. How are break and continue statements used in loops? Provide examples.

In C++, the break and continue statements are control flow tools used within loops to alter the normal sequence of execution. Here's how each works:

1. break Statement

The break statement is used to terminate the innermost loop or switch statement immediately, regardless of the loop's condition.

Example 1: Using break in a for loop

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3) {
         break; // Exit the loop when i equals 3
      }
```

```
        cout << i << " ";
    }
    return 0;
}
```

Output:

1 2

2. continue Statement

The continue statement **skips** the current iteration of the loop and proceeds with the next iteration, if the loop's condition is still true.

Example 2: Using continue in a for loop

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue; // Skip the rest of the loop when i equals 3
        } cout << i << " ";
    }return 0;
}
```

Output:

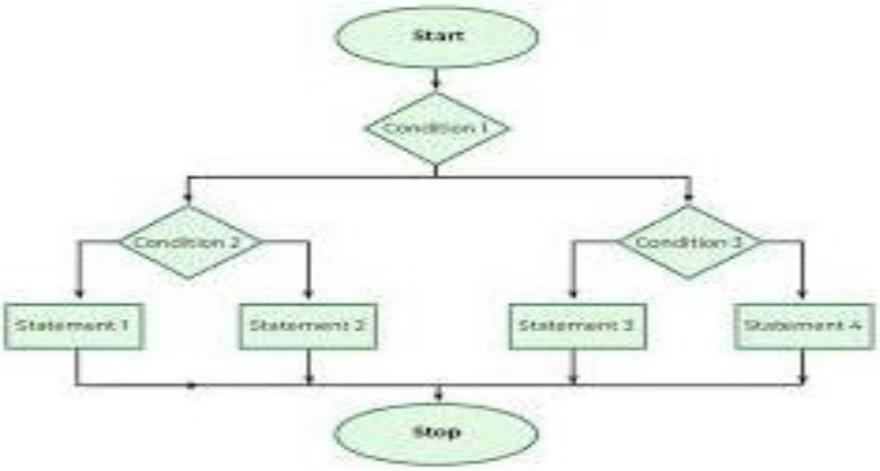
1 2 4 5

Here, when i equals 3, the continue statement causes the loop to skip printing 3 and proceed with the next iteration.

Use Cases

- **break**: Use when you need to exit a loop early, such as when a certain condition is met or an error occurs.
- **continue**: Use when you want to skip specific iterations based on a condition, without terminating the entire loop.

Q8. Explain nested control structures with an example.



In C++, nested control structures refer to placing one control statement (like if, for, while, or do-while) inside another. This allows for more complex decision-making and iteration patterns.

Understanding Nested Control Structures

Nested control structures enable you to evaluate multiple conditions or perform iterations within iterations. They are commonly used in scenarios like:

- **Multi-level decision-making**, such as classifying a student's grade based on multiple criteria.
- **Iterating over multi-dimensional data**, like processing a matrix.
- **Complex filtering**, such as checking multiple conditions before processing data.

Example 1: Nested if-else Statements

This example demonstrates how nested if-else statements can classify a number based on its divisibility:

```
#include <iostream>

using namespace std;

int main() {
   int n = 12;
   if (n % 2 == 0) {
      if (n % 3 == 0) {
         cout << "Divisible by 2 and 3";
      } else {
         cout << "Divisible by 2 but not 3";
      }
```

```
    } else {
        cout << "Not divisible by 2";
    }
    return 0;
}
```

Output:

Divisible by 2 and 3

Example 2: Nested Loops

Nested loops are used to iterate over multi-dimensional structures. Here's how you can print a 3x3 matrix

Code:-

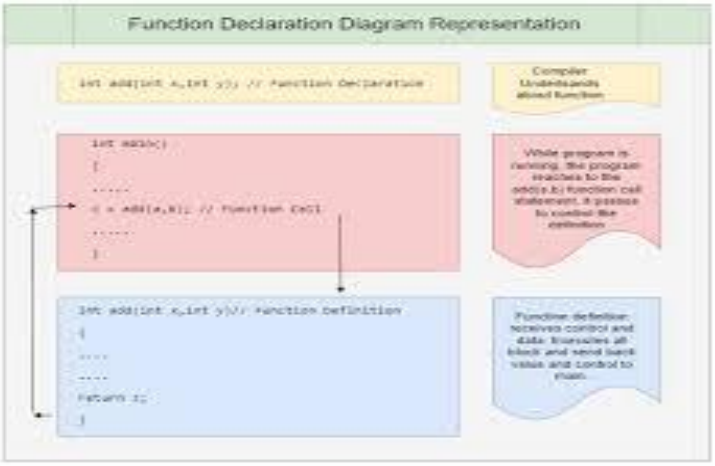
```
#include <iostream>
using namespace std;
int main() {
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

1 2 3

4 5 6

Q9. What is a function in C++? Explain the concept of function declaration, definition, and calling.



In C++, a function is a block of code designed to perform a specific task. Functions help in organizing code, promoting reusability, and improving readability. They can accept inputs (parameters), perform operations, and return outputs (values). Functions are fundamental in modular programming, allowing developers to break down complex problems into manageable pieces.

Function Declaration (Prototype)

A function declaration, also known as a prototype, informs the compiler about the function's name, return type, and parameters without providing the actual implementation. This is essential when the function is defined after its first use or is located in a different file.

Syntax:

return_type function_name(parameter1_type, parameter2_type, ...);

Example:

int add(int, int);

This declaration tells the compiler that there exists a function named add that takes two int parameters and returns an int value.

Function Definition

A function definition provides the actual implementation of the function. It includes the function header (return type, name, and parameters) followed by the function body, which contains the code to be executed.

Syntax:

return_type function_name(parameter1_type param1, parameter2_type param2, ...) {

    // Function body

    // Code to execute

    return value; // if return_type is not void

}

Example:

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

Function Calling

To execute a function, you need to call it. A function call passes control to the function, executes its code, and returns control to the point where it was called.

Syntax:

```
function_name(argument1, argument2, ...);
```

Example:

```
int result = add(5, 3);
```

Code:

```
#include <iostream>
using namespace std;
// Function declaration
int add(int, int);
int main() {
    int sum = add(10, 20); // Function call
    cout << "Sum: " << sum << endl;
    return 0;
}
// Function definition
int add(int a, int b) {
    return a + b;
}
```

Output:

```
Sum: 30
```

Q10. What is the scope of variables in C++? Differentiate between local and global scope.

In C++, the scope of a variable refers to the region of the program where the variable can be accessed. Variables can be categorized based on their scope into local and global variables.

Local Variables

Definition: Local variables are declared within a specific block of code, such as a function or a loop.

Characteristics:

- Scope: Limited to the block in which they are declared.
- Lifetime: Exist only during the execution of the block.
- Access: Cannot be accessed outside their defining block.
- Memory: Memory is allocated when the block is entered and deallocated when it exits.
- Naming: Can have the same name as variables in other blocks without conflict

Example:

```
#include <iostream>

using namespace std;

void func() {
    int age = 18;  // Local variable
    cout << "Age inside func: " << age << endl;
}

int main() {
    func();
    // cout << age;  // Error: 'age' was not declared in this scope
    return 0;
}
```

Global Variables

Definition: Global variables are declared outside of all functions, usually at the top of the program.

Characteristics:

- Scope: Accessible throughout the program after their declaration.
- Lifetime: Exist for the duration of the program's execution.
- Access: Can be accessed and modified by any function.
- Memory: Allocated when the program starts and deallocated when it ends.
- Naming: Should be used carefully to avoid unintended side effects and name conflicts.

Example:

```
#include <iostream>
using namespace std;
int globalVar = 100;  // Global variable
void display() {
    cout << "Global variable: " << globalVar << endl;
}
int main() {
    display();
    cout << "Accessing global variable in main: " << globalVar << endl;
    return 0;
}
```

- Variable Shadowing

If a local variable has the same name as a global variable, the local variable "shadows" the global one within its scope.

Example:

```
#include <iostream>
using namespace std;
int a = 5;  // Global variable
int main() {
    int a = 100;  // Local variable
    cout << "Local a: " << a << endl;  // Outputs 100
    cout << "Global a: " << ::a << endl;  // Outputs 5 using scope resolution operator
    return 0;
}
```

Q11. Explain recursion in C++ with an example.

Recursion in C++ is a programming technique where a function calls itself to solve a problem. This method is particularly useful for problems that can be broken down into smaller, similar sub-problems.

A recursive function typically consists of two main components:

1. **Base Case**: The condition under which the function stops calling itself, preventing infinite recursion.
2. **Recursive Case**: The part where the function calls itself with modified arguments, gradually moving towards the base case.

Example:

```
#include <iostream>

using namespace std;

int factorial(int n) {

   if (n <= 1) {
     return 1; // Base case
   } else {

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

   }

}

int main() {

   int num;

   cout << "Enter a number: ";

   cin >> num;

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

   return 0;

}
```

Q12.What are function prototypes in C++? Why are they used?

Function Prototype:-

A function prototype in C++ is a declaration of a function that specifies its name, return type, and parameter types, but omits the function body. It serves as a forward declaration, informing the compiler about the function's signature before its actual definition appears in the code.

Syntax:

return_type function_name(parameter1_type parameter1_name, parameter2_type parameter2_name, ...);

for eg:- int add(int a, int b);

Function Prototypes Use:-

Function prototypes are essential in C++ for several reasons:

1. **Forward Declaration**: They allow functions to be used before their definitions, enabling modular programming and better code organization.
2. **Type Checking**: Prototypes enable the compiler to verify that the correct number and types of arguments are passed to the function, helping to catch errors early.
3. **Code Clarity**: By declaring prototypes, you inform the programmer about the function's purpose and expected parameters, which aids in understanding the code structure.
4. **Modularity**: They facilitate separating the function interface from its implementation, promoting code reuse and easier maintenance.

Code:-

```
#include <iostream>

using namespace std;

// Function prototype

int add(int a, int b);

int main() {

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

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

   return 0;

}

// Function definition

int add(int a, int b) {

   return a + b; }
```

***5.* Arrays and Strings:**

[Q12] What are arrays in C++? Explain the difference between single-dimensional and multi- dimensional arrays.

Ans:

In C++, an array is a collection of fixed-size elements of the same data type stored in contiguous memory locations. Arrays allow you to store multiple values under a single variable name and access them using an index.

Syntax:

```
datatype arrayName[size];
```

Example:

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

This creates an array of 5 integers.

- **Single-Dimensional Array (1D Array)**

A 1D array is like a simple list where elements are arranged in a linear form.

Syntax:

datatype arrayName[size];

Example:

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

Access elements using:

marks[0]; // 85

marks[1]; // 90

- **Multi-Dimensional Array (e.g., 2D Array)**

A multi-dimensional array is like a table or matrix with rows and columns. The most common is the 2D array.

Syntax:

datatype arrayName[rows][columns];

Example:

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

Access elements using two indices:

matrix[0][1]; // 2

matrix[1][2]; // 6

- **Difference b/w single-dimensional array and multi-dimensional array:-**

A single-dimensional array is a linear collection of elements that can be accessed using a single index. It is often used when data needs to be stored in a sequence, such as a list of numbers, student marks, or temperatures. While single-dimensional arrays are suitable for simple lists, multi-dimensional arrays are ideal for representing structured data like matrices, tables, or grids.

For example, int marks[4] = {85, 90, 78, 92}; is a one-dimensional array where each element is accessed using one index like marks[0].

Where as multi-dimensional array (mostly a two-dimensional array) organizes data in a tabular form using rows and columns. It requires two or more indices to access elements. For example, int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; is a two-dimensional array where data is accessed using two indices such as matrix[0][1].

The key difference lies in their structure and the number of indices required to access elements: one for single-dimensional arrays and two or more for multi-dimensional arrays.

[Q13] Explain string handling in C++ with examples.

Ans:

String Handling in C++

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

1. C-style strings ie: character arrays.
2. C++ string class ie:from the Standard Template Library (STL).

1. C-style Strings

These are arrays of characters terminated by a null character ('\0').

Declaration:

```
char name[10] = "Alice";
```

Input and Output:

```
#include <iostream>

using namespace std;

int main() {
   char name[20];
   cout << "Enter your name: ";
   cin >> name;  // reads up to the first space
   cout << "Hello, " << name << "!" << endl;
   return 0;
}
```

2. C++ string Class

The string class in C++ is more powerful and user-friendly than C-style strings. It is part of the <string> header.

Declaration and Initialization:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string name = "Alice";
   cout << "Name: " << name << endl;
   return 0;
}
```

Input and Output:

```
string name;
cout << "Enter your name: ";
getline(cin, name); // reads full line including spaces
cout << "Hello, " << name << "!" << endl;
```

Common String Operations:-

Operation	Example	Description
Assign	s = "Hello";	Assign a value to a string
Concatenate	s = s1 + s2;	Join two strings
Length	s.length() or s.size()	Get the length of a string
Access character	s[0]	Access individual characters
Substring	s.substr(0, 4)	Get a substring
Comparison	s1 == s2, s1 < s2, etc.	Compare strings
Find	s.find("lo")	Find position of substring
Replace	s.replace(0, 4, "Hi")	Replace part of a string
Append	s.append(" World")	Add to the end of the string

Example: String Manipulation

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

```
int main() {
    string firstName = "John";
    string lastName = "Doe";
    string fullName = firstName + " " + lastName;
    cout << "Full Name: " << fullName << endl;
    cout << "Length: " << fullName.length() << endl;
    cout << "First character: " << fullName[0] << endl;
    fullName.replace(0, 4, "Jane"); // Replace "John" with "Jane"
    cout << "Updated Name: " << fullName << endl;
    return 0;
}
```

[Q14] How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Ans:-

- **Array Initialization in C++**

Arrays in C++ can be initialized in several ways at the time of declaration. You can initialize both 1D (one-dimensional) and 2D (two-dimensional) arrays with values.

1. One-Dimensional (1D) Array Initialization:-

a) Static Initialization

You can provide values at the time of declaration.

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

b) Partial Initialization

If fewer values are provided, the rest are automatically set to 0.

```
int nums[5] = {1, 2};  // Equivalent to {1, 2, 0, 0, 0}
```

c) Compiler-Sized Initialization

You can omit the size, and the compiler will determine it.

```
int data[] = {3, 6, 9};  // Size is 3
```

Example:-

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

```
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    for(int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

2. Two-Dimensional (2D) Array Initialization

a) Static Initialization (Row by Row)

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

b) Flat Initialization (Row-Major Order)

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

c) Partial Initialization

Unspecified elements are set to 0.

```
int matrix[2][3] = {
    {1},    // Row 0: 1, 0, 0
    {4, 5}  // Row 1: 4, 5, 0
};
```

Example:-

```
#include <iostream>
using namespace std;
int main() {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
```

```
    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

[Q15] Explain string operations and functions in C++.

Ans:-

String Operations and Functions in C++

In C++, string handling is greatly simplified using the string class provided by the Standard Template Library (STL) in the <string> header. It supports various operations and functions to manipulate and process strings efficiently.

Common String Operations

Assignment

```
string s;
s = "Hello";
```

Concatenation

```
string s1 = "Hello", s2 = "World";
string s3 = s1 + " " + s2; // "Hello World"
```

Length of a String

```
string s = "Hello";
cout << s.length(); // or s.size()
```

Accessing Characters

```
string s = "OpenAI";
cout << s[0];  // O
cout << s.at(2);  // e
```

Reading Strings

```
string name;
getline(cin, name);  // Reads input including spaces
```

Useful String Functions

Function	Description	Example
s.length() / s.size()	Returns length of the string	s.length() → 5
s.empty()	Returns true if string is empty	s.empty() → true/false
s.append(str)	Appends str to the end of string s	s.append(" GPT")
s.substr(pos, len)	Returns substring from pos of length len	"OpenAI".substr(0, 4) → "Open"
s.find(str)	Returns position of first occurrence of str	"abcabc".find("b") → 1
s.rfind(str)	Last occurrence of str	"abcabc".rfind("b") → 4
s.replace(pos, len, str)	Replaces part of string from pos with str	"Hello".replace(0, 2, "Y") → "Yllo"
s.compare(str)	Compares two strings (0 = equal, <0 or >0)	s1.compare(s2)
s.insert(pos, str)	Inserts str at position pos	"abc".insert(1, "X") → "aXbc"
s.erase(pos, len)	Erases len characters from position pos	"abcdef".erase(2, 2) → "abef"
s.clear()	Erases all characters from the string	s.clear()

Example:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "Hello";
    string str2 = "World";
    string result = str1 + " " + str2; // Concatenation
    cout << "Concatenated: " << result << endl;
    cout << "Length: " << result.length() << endl;
```

```
    result.replace(0, 5, "Hi"); // Replace "Hello" with "Hi"
    cout << "After Replace: " << result << endl;

    string sub = result.substr(3, 5); // Substring
    cout << "Substring: " << sub << endl;

    int pos = result.find("World"); // Finding
    cout << "Position of 'World': " << pos << endl;

    result.insert(2, "---"); // Insert
    cout << "After Insert: " << result << endl;

    result.erase(2, 3); // Erase inserted characters
    cout << "After Erase: " << result << endl;
    return 0;
}
```

6.Introduction to Object-Oriented Programming

[Q16]Explain the key concepts of Object-Oriented Programming (OOP).

Ans:

Key Concepts:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which contain data (fields or attributes) and functions (methods) that operate on the data. OOP is widely used in C++, Java, Python, and many other languages.

1. Encapsulation

Encapsulation is the concept of wrapping data and functions together into a single unit called a class. It helps protect the internal state of an object from outside interference and misuse.

- Access specifiers (private, public, protected) are used to control visibility.
- Only public methods are used to access and modify private data (getters/setters).

Example:

```
class Student {
private:
    int age;
public:
```

```
    void setAge(int a) {
        age = a;
    }
    int getAge() {
        return age;
    }
}
```

2. Abstraction

Abstraction means showing only essential features and hiding unnecessary implementation details from the user.

- Achieved using abstract classes and interfaces.
- Allows the programmer to focus on *what* an object does instead of *how* it does it.

Example:

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

3. Inheritance

Inheritance allows a class (child/derived class) to inherit properties and behaviors from another class (parent/base class). It promotes code reusability.

- Types: Single, Multiple, Multilevel, Hierarchical
- Keywords: public, protected, private inheritance

Example:

```
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
class Dog : public Animal {
```

```
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};
```

4. Polymorphism

Polymorphism means one function behaves differently in different contexts. It allows for dynamic behavior at runtime.

There are two types:

- Compile-time polymorphism (Function Overloading, Operator Overloading)
- Run-time polymorphism (Method Overriding using virtual functions)

Example:

```
class Animal {
public:
    virtual void speak() {
        cout << "Animal speaks" << endl;
    }
};
class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog barks" << endl;
    }
};
```

[Q17] What are classes and objects in C++? Provide an example.

Class: It is a blueprint or template for creating objects. It defines data members (variables) and member functions (methods) that operate on those data.

Object: It is an instance of a class. It represents a specific implementation of the class, with actual values assigned to the class properties.

An Example:-

```
#include <iostream>
using namespace std;
// Class definition
class Car {
public:
// Data members
    string brand;
    int year;
    // Member function
    void displayInfo() {
        cout << "Brand: " << brand << endl;
        cout << "Year: " << year << endl;
    }
};
int main() {
    // Object creation
    Car car1;
    // Assign values
    car1.brand = "Toyota";
    car1.year = 2022;
    // Call member function using object
    car1.displayInfo();
    return 0;
}
```

[Q18] What is inheritance in C++? Explain with an example.

Inheritance in C++:

Inheritance is one of the core features of Object-Oriented Programming (OOP) in C++. It allows a class (called the derived class or child class) to inherit properties and behaviors (data members and member functions) from another class (called the base class or parent class).

Types of Inheritance:

- Single inheritance: One base and one derived class (as in the example above).
- Multiple inheritance: One derived class inherits from more than one base class.
- Multilevel inheritance: Derived class becomes a base for another class.
- Hierarchical inheritance: Multiple derived classes inherit from a single base class.
- Hybrid inheritance: Combination of two or more types of inheritance.

Syntax:

```
class BaseClass {
   // members
};
class DerivedClass : access-specifier BaseClass {
   // additional members
};
```

Example:-

```
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
   void eat() {
      cout << "This animal eats food." << endl;
   }
};
// Derived class
class Dog : public Animal {
public:
   void bark() {
      cout << "The dog barks." << endl;
   }
};
```

```
int main() {
    Dog myDog;
    // Call function from base class
    myDog.eat();
    // Call function from derived class
    myDog.bark();
    return 0;
}
```

[Q19] What is encapsulation in C++? How is it achieved in classes?

Encapsulation is an object-oriented programming concept where data (variables) and the functions (methods) that operate on the data are bundled together into a single unit — a class.

It also involves restricting direct access to some components of an object, which is a way to enforce data hiding and maintain control over how data is accessed or modified.

In c++, encapsulation is achieved by using access specifiers: private, protected, and public,

Keeping data members private or protected and providing public methods (getters and setters) to access or modify the private data.

Example:-

```
#include <iostream>
using namespace std;
class Person {
private:
// Private data member (cannot be accessed directly)
   string name;
public:
// Setter method to set name
   void setName(string n) {
       name = n;
    }
    // Getter method to get name
    string getName() {
       return name;
```

```
    }
};
int main() {
    Person p;
    // Accessing private data via public methods
    p.setName("Alice");
    cout << "Name: " << p.getName() << endl;
    return 0;
}
```