-By Aakash

# Software Engineering Assignment

MODULE: 3

## Object oriented Programming(oop)

### *1. Introduction to C++*

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

➢ ***Ans. :***

Here’s a concise breakdown of the key differences between **Procedural Programming (PP)** and **Object-Oriented Programming (OOP)**:

**1. Conceptual Focus :**

- **PP**: Focuses on **functions** and procedures that perform tasks on data.
- **OOP**: Focuses on **objects** which encapsulate both **data** and **methods** (functions).

**2. Data Organization :**

- **PP**: Data is separate from functions, and often manipulated globally or passed between functions.
- **OOP**: Data is encapsulated within **objects** and can only be accessed or modified via methods.

**3. Code Reusability :**

- **PP**: Reusability is achieved by writing reusable functions, but data is not easily encapsulated, making reuse harder in larger systems.
- **OOP**: Reusability is achieved through **inheritance** (where classes can inherit properties and methods from other classes) and **polymorphism** (methods behaving differently based on the object).

**4. Modularity :**

- **PP**: Modularity is achieved through **functions**, but the code structure can become complex as programs grow.
- **OOP**: Modularity is achieved through **objects** and **classes**, providing a cleaner and more scalable structure for larger applications.

**5. State and Behavior :**

- **PP**: The state is usually stored in **global variables** or passed between functions, leading to potential side effects.
- **OOP**: The state is **encapsulated within objects**, and behavior is defined by the methods of those objects.

## 6. Abstraction and Encapsulation :

- **PP**: Limited abstraction and no direct concept of **encapsulation**.
- **OOP**: Strong emphasis on **abstraction** (hiding complexity) and **encapsulation** (hiding data from direct access).

## 7. Inheritance and Polymorphism :

- **PP**: No inheritance or polymorphism.
- **OOP**: Supports **inheritance** (classes can derive from others) and **polymorphism** (same method can behave differently based on the object).

## 8. Example Languages :

- **PP**: C, Pascal, FORTRAN.
- **OOP**: Java, C++, Python, Ruby

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

➢ *Ans.*

Object-Oriented Programming (OOP) has several advantages over Procedural Oriented Programming (POP), including:

- **Code reusability**: OOP allows developers to reuse code by creating new classes based on existing ones. This reduces code duplication and saves time.
- **Modularity**: OOP breaks down complex code into smaller, more manageable chunks, making it easier to maintain and update.
- **Security**: OOP supports access control and data hiding, making it more secure than POP.
- **Problem solving**: OOP breaks down software code into smaller components, one object at a time.
- **Productivity**: OOP practices help amplify productivity.
- **Troubleshooting**: OOP makes troubleshooting simpler.
- **Flexibility**: OOP results in flexible code.
- **Early issue resolution**: OOP addresses issues early on.
- **Organized code structure**: OOP's principles of encapsulation, inheritance, and polymorphism allow for a clear and organized code structure.

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

➢ Ans.

## 1. Install a C++ Compiler

- **Windows**: Download and install **MinGW** (Minimalist GNU for Windows) or **Microsoft Visual C++ Compiler** (via Visual Studio).
- **Mac**: Install **Xcode** via the App Store (includes `clang` compiler).
- **Linux**: Install **GCC** (GNU Compiler Collection) using a package manager (e.g., `sudo apt install g++` on Ubuntu).

## 2. Install an Integrated Development Environment (IDE) :

- **Windows**: Use **Visual Studio** or **Code::Blocks**.
- **Mac**: Use **Xcode** or **CLion**.
- **Cross-platform**: Install **Visual Studio Code**, **Eclipse**, or **CLion**, and configure them with a C++ extension.

## 3. Configure the IDE :

- Set the **compiler path** if not automatically detected (e.g., in Visual Studio Code, set up `tasks.json` for build configuration).
- Ensure the IDE is linked to the correct **C++ standard library** (e.g., GCC or Clang).

## 4. Verify Installation :

- Open a terminal or command prompt.
- Type `g++ --version` (or `clang++ --version` on Mac) to ensure the compiler is correctly installed.
- Create a simple C++ program (e.g., `Hello, World!`) and compile it using the IDE or terminal.

## 5. Run Your First Program :

- Write a basic C++ program in your IDE.
- Build and run the program to check if the setup works correctly.

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

➢ ***Ans.***

In C++, the main input/output (I/O) operations are handled through **streams** using the **iostream** library.

## 1.Output (Printing to the Console) :

- **cout:** Used to output data to the console.
- **Stream Insertion Operator (<<)**: Used to send data to cout.

**Example:**

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" ;

}
```

In C++, the main input/output (I/O) operations are handled through **streams** using the **iostream** library. The key operations are **input** using `cin` and **output** using `cout`.

## 1. Output (Printing to the Console)

- `cout`: Used to output data to the console.
- **Stream Insertion Operator (**<<**)**: Used to send data to `cout`.

Example:

```
cpp
Copy code
#include <iostream>using namespace std;
int main() {
    int age = 25;
    cout << "Hello, World!" << endl;     // Prints text
    cout << "Age: " << age << endl;      // Prints the value of age
    return 0;
}
```

---

## 2. **Input (Reading from the Console):**

- **cin:** Used to take input from the console.
- **Stream Extraction Operator (>>)**: Used to read data from cin.

**Example:**

```
#include <iostream>
using namespace std;
int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;                    // Takes input from user
    cout << "You are " << age << " years old." << endl;
    }
```

—*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*—*—

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

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

➢ *Ans.*

C++ provides several data types, which can be categorized into **basic**, **derived**, **user-defined**, and **void** data types. Here's a concise explanation:

### 1. Basic Data Types

- **`int`:** Used for integers (whole numbers).
  - **Example**: `int age = 25;`
- **`float`**: Used for single-precision floating-point numbers.
  - **Example**: `float pi = 3.14f;`
- **`double`:** Used for double-precision floating-point numbers (more precision than `float`).
  - **Example**: `double price = 19.99;`
- **`char`**: Used for single characters.
  - **Example**: `char grade = 'A';`
- **`bool`:** Used for boolean values (`true` or `false`).
  - **Example**: `bool isRaining = false;`

### 2. Derived Data Types

- **Array**: A collection of elements of the same type.
  - **Example**: `int nums[3] = {1, 2, 3};`
- **Pointer**: Stores the memory address of another variable.
  - **Example**: `int* ptr = &age;`
- **Reference**: An alias for another variable.
  - **Example**: `int& ref = age;`

### 3. User-Defined Data Types

- **`struct`:** A custom data type to group variables.

```
struct Person {

            string name;
```

```
                int age;
            };
```

- **`union`:** Stores different types of data in the same memory location.

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

- **`enum`:** Defines a set of constants.

```
enum Day { Sunday, Monday, Tuesday };
```

## 4. Void Data Type

- `void`: Represents the absence of a type, used in functions that don't return a value.

```
void printMessage() { cout << "Hello!"; }
```

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

➢ ***Ans.***

In C++, **type conversion** refers to converting one data type into another. There are two types of type conversion: **implicit** and **explicit**.

### 1. Implicit Type Conversion (Automatic Type Conversion) :

- **Definition**: This is automatically performed by the compiler when there is no risk of data loss.
- **When it occurs**: Happens when a lower data type (e.g., `int`) is assigned to a higher data type (e.g., `float`).

**Example**:

```
int a = 10;
float b = a; // Implicit conversion from int to float
```

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

- **Definition**: This is performed manually by the programmer using type casting to convert one type to another.
- **When it occurs**: Used when you need to convert data types explicitly, often when data loss might occur, or when converting between types that the compiler doesn't automatically convert.

```
double a = 9.99;
int b = (int)a; // Explicit conversion from double to int (data
loss)
```

- **Key Difference:**

- **Implicit**: Done automatically by the compiler when no data loss is expected.
- **Explicit**: Done manually by the programmer using type casting (e.g., `(type)variable`).

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

➢ ***Ans.***

C++ provides several types of operators that perform operations on variables and values. Here's a concise overview of the different types of operators:

C++ provides several types of operators that perform operations on variables and values. Here's a concise overview of the different types of operators:

### 1. Arithmetic Operators :

- Used to perform basic mathematical operations.

- **Examples**:

```
int a = 10,
b = 5;
int sum = a + b;    // Addition
int diff = a - b;   // Subtraction
int prod = a * b;   // Multiplication
int div = a / b;    // Divisionint
mod = a % b;    // Modulo (remainder)
```

### 3. Relational (Comparison) Operators :

- Used to compare two values.

**Example:**

```
 int a = 10,
 b = 5;
 bool result1 = a == b; // Equal to
 bool result2 = a != b; // Not equal to
 bool result3 = a > b; // Greater than
 bool result4 = a < b; // Less than
 bool result5 = a >= b; // Greater than or equal to
 bool result6 = a <= b; // Less than or equal to
```

**3. Logical Operators :**

- Used to perform logical operations (mainly with boolean values).

**Example :**

```
bool x = true,

 y = false;

bool andResult = x && y; // Logical AND

 bool orResult = x || y; // Logical OR

 bool notResult = !x; // Logical NOT
```

**4. Bitwise Operators :**

- Used to perform bit-level operations on integers.

```
int a = 5,

 b = 3;

 int andResult = a & b; // Bitwise AND

int orResult = a | b; // Bitwise OR

 int xorResult = a ^ b; // Bitwise XOR

int notResult = ~a; // Bitwise NOT

 int leftShift = a << 1; // Left shift

 int rightShift = a >> 1;// Right shift
```

**5. Assignment Operators :**

- Used to assign values to variables.

**Example :**

```
int a = 10;

 a += 5;   // a = a + 5

 a -= 3;   // a = a - 3

 a *= 2;   // a = a * 2

 a /= 2;  // a = a / 2

 a %= 3;  // a = a % 3
```

C++ provides several types of operators that perform operations on variables and values. Here's a concise overview of the different types of operators:

## 1. Arithmetic Operators

Used to perform basic mathematical operations.

- **Examples**:

```
int a = 10, b = 5;
int sum = a + b;    // Additionint
diff = a - b;   // Subtraction
int prod = a * b;   // Multiplication
int div = a / b;    // Division
int mod = a % b;    // Modulo (remainder)
```

## 2. Relational (Comparison) Operators

Used to compare two values.

**Examples**:

```
int a = 10, b = 5;
bool result1 = a == b;  // Equal to
bool result2 = a != b;  // Not equal to
bool result3 = a > b;   // Greater than
bool result4 = a < b;   // Less than
bool result5 = a >= b;  // Greater than or equal to
bool result6 = a <= b;  // Less than or equal to
```

## 3. Logical Operators

Used to perform logical operations (mainly with boolean values).

**Example:**

```
bool x = true, y = false;
bool andResult = x && y;  // Logical AND
bool orResult = x || y;   // Logical OR
bool notResult = !x;      // Logical NOT
```

## 4. Bitwise Operators

Used to perform bit-level operations on integers.

**Examples**:

```
int a = 5, b = 3;
int andResult = a & b;  // Bitwise AND
int orResult = a | b;   // Bitwise OR
int xorResult = a ^ b;  // Bitwise XOR
int notResult = ~a;     // Bitwise NOT
int leftShift = a << 1; // Left shift
int rightShift = a >> 1;// Right shift
```

## 5. Assignment Operators

Used to assign values to variables.

**Examples**:

```
int a = 10;
a += 5;  // a = a + 5
a -= 3;  // a = a - 3
a *= 2;  // a = a * 2
a /= 2;  // a = a / 2
a %= 3;  // a = a % 3
```

**6. Increment and Decrement Operators :**

- Used to increase or decrease the value of a variable by 1.

**Example :**

```
int a = 10;

a++; // Increment (post-increment) ++a; // Increment (pre-increment)
a--; // Decrement (post-decrement) --a; // Decrement (pre-decrement
```

C++ provides several types of operators that perform operations on variables and values. Here's a concise overview of the different types of operators:

## 1. Arithmetic Operators

Used to perform basic mathematical operations.

**Examples**:

```
      int a = 10,
       b = 5;int sum = a + b;    // Addition
      int diff = a - b;   // Subtraction
      int prod = a * b;   // Multiplication
      int div = a / b;    // Division
      int mod = a % b;    // Modulo (remainder)
```

## 2. Relational (Comparison) Operators

Used to compare two values.

**Examples**:

```
int a = 10,
 b = 5;
bool result1 = a == b;  // Equal to
bool result2 = a != b;  // Not equal to
bool result3 = a > b;   // Greater than
bool result4 = a < b;   // Less than
bool result5 = a >= b;  // Greater than or equal to
bool result6 = a <= b;  // Less than or equal to
```

## 3. Logical Operators

Used to perform logical operations (mainly with boolean values).

**Examples**:

```
bool x = true,
y = false;
bool andResult = x && y;  // Logical AND
bool orResult = x || y;   // Logical OR
bool notResult = !x;      // Logical NOT
```

## 4. Bitwise Operators

Used to perform bit-level operations on integers.

**Examples**:

```
int a = 5,
 b = 3;
int andResult = a & b;  // Bitwise AND
int orResult = a | b;   // Bitwise OR
int xorResult = a ^ b;  // Bitwise XOR
int notResult = ~a;     // Bitwise NOT
int leftShift = a << 1; // Left shift
int rightShift = a >> 1;// Right shift
```

## 5. Assignment Operators

Used to assign values to variables.

```
int a = 10;
a += 5;  // a = a + 5
a -= 3;  // a = a - 3
a *= 2;  // a = a * 2
a /= 2;  // a = a / 2
a %= 3;  // a = a % 3
```

## 6. Increment and Decrement Operators

Used to increase or decrease the value of a variable by 1.

**Examples**:

```
int a = 10;
a++;  // Increment (post-increment)
++a;  // Increment (pre-increment)
a--;  // Decrement (post-decrement)
--a;  // Decrement (pre-decrement)
```

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

- A shorthand for `if-else` statements.

**Example :**

```
int a = 10, b = 5;

int max = (a > b) ? a : b; // If a > b, max = a; else max = b
```

**8. Typecast Operator :**

-Used to convert one data type to another.

Example:

```
double a = 9.99;

 int b = (int)a; // Explicit type conversion from double to int
```

**Q.4** ***Explain the purpose and use of constants and literals in C++.***

➢ *Ans.*

In C++, **constants** and **literals** are both used to represent fixed values, but they serve different purposes and have distinct characteristics.

- **Constants in C++ :**

**Definition**: A constant is a value that cannot be changed during the execution of the program. Constants are typically defined using the `const` keyword or the `#define` preprocessor directive.

- **Purpose**:
  - To make code more readable and maintainable.
  - To avoid accidental modification of critical values.
  - To define values that should remain unchanged throughout the program, such as mathematical constants or configuration settings.

**Example :**

```
const int MAX_USERS = 100;

 const double PI = 3.14159;
```

## Use:

- Constants can be used to represent fixed values (like `MAX_USERS` or `PI`) that are referenced multiple times throughout the program.

In C++, **constants** and **literals** are both used to represent fixed values, but they serve different purposes and have distinct characteristics.

## Constants in C++

**Definition**: A constant is a value that cannot be changed during the execution of the program. Constants are typically defined using the `const` keyword or the `#define` preprocessor directive.

**Purpose**:

- To make code more readable and maintainable.
- To avoid accidental modification of critical values.
- To define values that should remain unchanged throughout the program, such as mathematical constants or configuration settings.

**Example**:

```
const int MAX_USERS = 100;const double PI = 3.14159;
```

**Use**:

- Constants can be used to represent fixed values (like `MAX_USERS` or `PI`) that are referenced multiple times throughout the program.
- They make the program easier to maintain because if a value needs to change, it only needs to be updated in one place.

---

- **Literals in C++ :**

**Definition**: A literal is a fixed value that appears directly in the source code. It can be an integer, floating-point number, character, string, or boolean value. Unlike constants, literals are not named and are written directly into expressions.

**Purpose**:

- To represent immediate values that are used directly in expressions, calculations, or function calls.

- Literals are written directly in the code, such as numbers (`10`), characters (`'A'`), or strings (`"Hello"`).

**Example :**

```
int x = 10; // 10 is an integer

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

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

## Use:

- Literals are used when specific values are needed in the program for assignments, comparisons, or calculations.
- They are typically not reused or given a name.

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

## *3. Control Flow Statements :*

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

**Conditional statements** in C++ are used to perform different actions based on whether a condition is true or false. They allow the program to make decisions and control the flow of execution.

### 1. if-else Statement:

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

**Example :**

```
int x = 10;

 if (x > 5) {

 cout << "x is greater than 5";

 } else {

 cout << "x is less than or equal to 5";

 }
```

**Conditional statements** in C++ are used to perform different actions based on whether a condition is true or false. They allow the program to make decisions and control the flow of execution.

## 1. if-else Statement:

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

**Example**:

```
cpp
Copy code
int x = 10;if (x > 5) {
    cout << "x is greater than 5";
} else {
    cout << "x is less than or equal to 5";
}
```

## 2. switch Statement:

- The `switch` statement evaluates a variable or expression and matches it with multiple possible cases. It executes the block of code corresponding to the matching case. If no case matches, the `default` block is executed (if provided).

**Example :**

```
int day = 3;

 switch (day) {

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

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

case 3: cout << "Wednesday"; break;

 default: cout << "Invalid day"; break;

 }
```

**Q.2*What is the difference between* for, while*, and* do-while *loops in C++?***

➢ ***Ans.***

In C++, **for**, **while**, and **do-while** loops are used for repeating a block of code, but they differ in their syntax and when the condition is checked:

## 1. for loop:

- **Used** when the number of iterations is known in advance.
- The condition is checked **before** each iteration.

## 2. while loop:

- **Used** when the number of iterations is unknown and you want to repeat as long as a condition is true.
- The condition is checked **before** each iteration.

## 3. do-while loop:

- **Used** when you want the code to run **at least once**, even if the condition is false initially.
- The condition is checked **after** the first iteration.

## ● Key Differences:

- **for**: Best when the number of iterations is known.
- **while**: Best when the number of iterations is not known, and you check the condition first.
- **do-while**: Best when you need to execute the loop at least once, checking the condition after the first iteration.

**Q.3** ***How are* break *and* continue *statements used in loops? Provide examples.***

➢ Ans.

In C++, `break` and `continue` are used to control the flow of execution in loops.

## 1. break Statement:

- **Purpose**: It immediately **exits** the loop, regardless of the loop condition, and control is transferred to the next statement after the loop.

**Example :**

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

if (i == 3) {

 break; // Exit the loop when i is 3

}

cout << i << " ";

} // Output: 0 1 2
```

## 2. continue Statement :

- **Purpose**: It **skips the current iteration** of the loop and moves to the next iteration, without exiting the loop.

**Example**:

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

if (i == 3) {

continue; // Skip the iteration when i is 3

}

cout << i << " ";

 } // Output: 0 1 2 4
```

- **Summary :**

  - **`break`:** Exits the loop entirely.
  - **`continue`:** Skips the current iteration and continues with the next iteration.

## Q.3 Explain nested control structures with an example.

➢ ***Ans.*** Nested control structures in C++ occur when one control structure (like a loop or conditional statement) is placed inside another.

- Example of Nested **if-else**:

```
int x = 5,     y = 10;
if (x > 0) {
 if (y > 5) {
cout << "x is positive and y is greater than 5";
} else
 {
 cout << "x is positive but y is not greater than 5";
 }
 } else
 {
cout << "x is not positive";
}
```

**Example of Nested for loop:**

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

−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−*−

## *4. Functions and Scope :*

**Q.1** ***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;
}
```

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

➢ ***Ans.***

In C++, the scope of a variable refers to the region of a program where the variable can be accessed or modified. The scope of variables in C++ can be classified into `local` scope, `global` scope, and several other specialized scopes based on where the variable is declared. The two most common types of scope are `local scope` and `global scope`.

## 1. Local Scope

A variable has **local scope** if it is declared inside a function, a block of code (e.g., inside `if`, `for`, `while` blocks), or a method. This means that the variable is only accessible within that function or block where it was defined.

- **Visibility:** The variable is only visible and accessible from the point of its declaration to the end of the block in which it is declared.

- **Lifetime:** The variable exists only during the execution of the block in which it is declared, and it is destroyed when the block finishes execution.

- **Memory:** Memory for the variable is allocated when the block starts, and deallocated when the block ends.

- **Example :**

```
 void exampleFunction() {

int x = 10; // ’x’ has local scope here

std::cout << x; // Valid: ’x’ is accessible within the function

 } // ’x’ is not accessible here, outside the function
```

## 2. Global Scope

A variable has **global scope** if it is declared outside of all functions, typically at the top of the program (before any function definitions). A global variable is accessible throughout the entire program, across all functions and blocks.

**Visibility:** The variable is visible and accessible from any function or block in the program, including after its declaration (unless shadowed by a local variable with the same name).

**Lifetime:** The variable exists for the entire duration of the program. Memory is allocated when the program starts and deallocated when the program exits.

**Memory:** Memory for the global variable is allocated once when the program starts, and it remains allocated until the program terminates.

- **Example :**

```
int globalVar = 42; // Global variable with global scope void
someFunction()

{ std::cout << globalVar; // Valid: ’globalVar’ is accessible inside
the function
```

```
 }

int main() {

 someFunction(); // Also valid: 'globalVar' can be accessed here as
well  }
```

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

- ***Ans.***

## Recursion in C++

**Recursion** in C++ (and in programming in general) is a technique where a function calls itself in order to solve a problem. A recursive function generally has two parts:

1. **Base Case**: This is the condition where the recursion stops. Without a base case, the function would call itself indefinitely, leading to a stack overflow.
2. **Recursive Case**: This is the part of the function where it calls itself with a smaller or simpler subproblem, progressively moving towards the base case.

- **C++ Code Example: Factorial Using Recursion :**

```
#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;
}
```

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

- ***Ans.***

## Function Prototypes in C++

A **function prototype** in C++ is a declaration of a function that specifies the function's name, return type, and parameters (but without the function body). Function prototypes are typically placed before the `main()` function or other functions that call the declared function, to inform the compiler about the function's signature (i.e., how the function is defined), allowing it to recognize the function's usage in the code.

- **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 With Function Prototype :**

```
#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
```

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

## *5. Array and string :*

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

➢ ***Ans.***

### Arrays in C++

An **array** in C++ is a collection of elements of the same type, stored in contiguous memory locations. Each element in the array can be accessed using an index. Arrays allow you to store multiple values in a single variable, rather than having to create individual variables for each value.

- **Syntax to Declare an Array:**

```
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**: //123          //456

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

➢ **Ans.**

## String Handling in C++ :

In C++, **strings** are used to represent sequences of characters. There are two main ways to handle strings:

1. **C-style Strings**: These are arrays of characters terminated by a null character (`'\0'`).
2. **C++ `std::string`**: This is a more modern and flexible way to handle strings, provided by the C++ Standard Library.

● **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;
}
```

**Q.3** ***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.

   ```
   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
```

**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
```

**Q.4** ***. 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().

```
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
```

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

# *5. Introduction to Object-Oriented Programming*

***Q.1 Explain the key concepts of Object-Oriented Programming (OOP).***

- ***Ans***

In C++, **classes** and **objects** follow the same fundamental concept as in other Object-Oriented Programming (OOP) languages, but the syntax and some implementation details are different. Let's break down **classes** and **objects** in C++.

## 1. Class: A Blueprint or Template for Creating Objects

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

- **Attributes**: These are variables that hold data related to the object. In C++, attributes are typically declared inside the class.
- **Methods**: These are functions that define actions the object can perform.

Syntax for Defining a Class in C++:

```
class ClassName {

private: // Attributes (private by default, can be changed to public)
int attribute1;

 float attribute2;

public: // Constructor (used to initialize attributes)

ClassName(int a, float b) {

attribute1 = a;

attribute2 = b;

 } // Method to perform an action (public method)

 void display() {

std::cout << "Attribute 1: " << attribute1 << std::endl;

std::cout << "Attribute 2: " << attribute2 << std::endl;

}

 };
```

In C++, **classes** and **objects** follow the same fundamental concept as in other Object-Oriented Programming (OOP) languages, but the syntax and some implementation details are different. Let's break down **classes** and **objects** in C++.

## 1. Class: A Blueprint or Template for Creating Objects

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

- **Attributes**: These are variables that hold data related to the object. In C++, attributes are typically declared inside the class.
- **Methods**: These are functions that define actions the object can perform.

### Syntax for Defining a Class in C++:

```
cpp
Copy code
class ClassName {private:
    // Attributes (private by default, can be changed to public)
    int attribute1;
    float attribute2;
public:
    // Constructor (used to initialize attributes)
    ClassName(int a, float b) {
        attribute1 = a;
        attribute2 = b;
    }

    // Method to perform an action (public method)
    void display() {
        std::cout << "Attribute 1: " << attribute1 << std::endl;
        std::cout << "Attribute 2: " << attribute2 << std::endl;
    }
};
```

- `private`: These attributes and methods are not accessible from outside the class. This is used to protect data integrity.
- `public`: These attributes and methods can be accessed from outside the class.

---

## 2. Object: An Instance of a Class

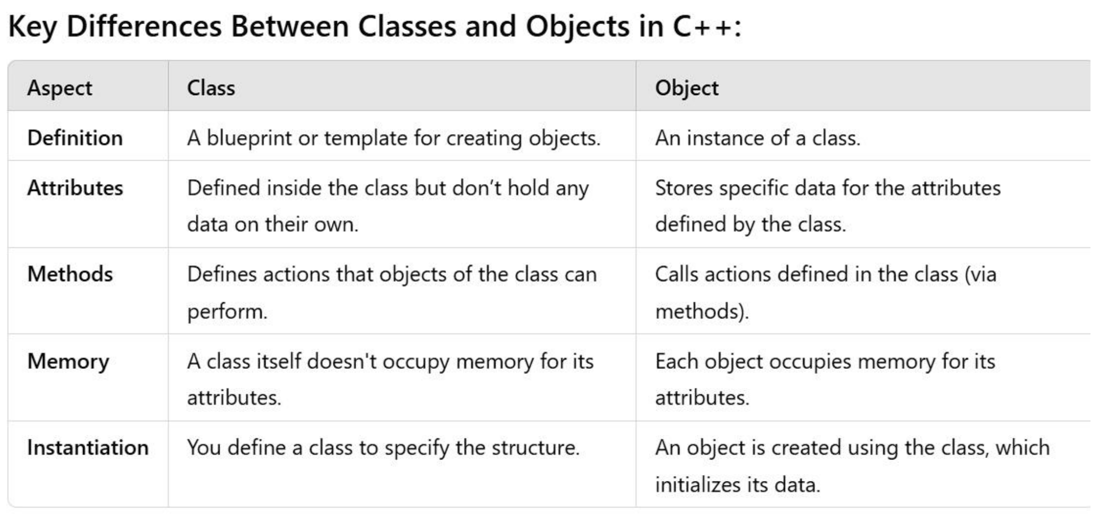
An **object** is an instance of a class. It represents a specific entity that has the properties and behaviors defined by the class.

In C++, you create an object by declaring a variable of the class type and initializing it, usually by calling the class's constructor.

## Example of Creating an Object:

```
#include <iostream>

using namespace std;

class Dog {

private:

string name; string breed;

public: // Constructor to initialize the Dog object Dog

(string n, string b) {

 name = n;

 breed = b;

} // Method to display dog details

 void bark() {

 cout << name << " is barking!" << endl;

 } // Method to display dog details

 void display() {

 cout << "Name: " << name << ", Breed: " << breed << endl;

 }

};

int main() { // Create objects (instances) of the Dog class Dog
dog1("Buddy", "Golden Retriever");

 Dog dog2("Max", "Bulldog"); // Call methods on the objects
dog1.display(); // Output: Name: Buddy, Breed: Golden Retriever
dog1.bark(); // Output: Buddy is barking! dog2.display(); // Output:
```

```
Name: Max, Breed: Bulldog dog2.bark(); // Output: Max is barking!
return 0;

 }
```



### Key Differences Between Classes and Objects in C++:

| Aspect | Class | Object |
|---|---|---|
| **Definition** | A blueprint or template for creating objects. | An instance of a class. |
| **Attributes** | Defined inside the class but don't hold any data on their own. | Stores specific data for the attributes defined by the class. |
| **Methods** | Defines actions that objects of the class can perform. | Calls actions defined in the class (via methods). |
| **Memory** | A class itself doesn't occupy memory for its attributes. | Each object occupies memory for its attributes. |
| **Instantiation** | You define a class to specify the structure. | An object is created using the class, which initializes its data. |

## Q.2 *What are classes and objects in C++? Provide an example.*

- ***Ans.***

### Classes and Objects in C++ :

**Class**: A class in C++ is a blueprint or template for creating objects. It defines the attributes (properties) and methods (functions) that the objects will have.

**Object**: An object is an instance of a class. It represents a specific entity created from the class and can hold data (attributes) and perform actions (methods) defined by the class.

Example:

```
#include <iostream>

using namespace std;

 class Dog {

public:

 string name;

string breed; // Constructor to initialize attributes

 Dog(string n, string b) {
```

```
name = n;

 breed = b;

}

// Method to display information

void display() {

 cout << "Name: " << name << ", Breed: " << breed << endl;

}

};

 int main() {

 // Creating objects (instances) of the Dog class Dog

dog1("Buddy", "Golden Retriever");

 Dog dog2("Max", "Bulldog"); // Calling methods on the objects
dog1.display(); // Output: Name: Buddy, Breed: Golden Retriever
dog2.display(); // Output: Name: Max, Breed: Bulldog

return 0;

}
```

Q.3 ***What is inheritance in C++? Explain with an example.***

- ***Ans.***

 Inheritance in C++ is a feature that allows one class (called the derived class) to inherit the properties and behaviors (data members and member functions) from another class (called the base class). This promotes code reusability and establishes a relationship between the base and derived classes.

**Key Points:**

- **Base Class**: The class whose members are inherited.
- **Derived Class**: The class that inherits the members of the base class.
- **Access Modifiers**: Inherited members can be private, protected, or public depending on the access specifier used in inheritance.

## Main Types of Inheritance:

1. **Single Inheritance**: One derived class inherits from one base class.
2. **Multiple Inheritance**: A derived class inherits from more than one base class.
3. **Multilevel Inheritance**: A class inherits from another derived class.

**Example:**
```
#include <iostream>

 using namespace std;

// Base class

 class Animal {

 public:

 void speak() {

cout << "Animal speaks!" << endl;

 }};

// Derived class

 class Dog :

 public Animal {

 public:

 void bark()

  {

cout << "Dog barks!" << endl;

 }

};

 int main() {

Dog dog; dog.speak(); // Inherited function from Animal class
dog.bark(); // Function of Dog class

}
```

**Q.4** ***What is encapsulation in C++? How is it achieved in classes?***

➢ Ans.

Encapsulation in C++ is the concept of bundling data (variables) and methods (functions) that operate on the data into a single unit, typically a class. It also involves restricting access to certain components of an object, allowing control over how data is accessed or modified.

**How it's achieved in classes:**

**Private Access Modifier**: Variables are usually declared as **private** to prevent direct access from outside the class. This ensures data is protected from unauthorized modification.

**Public Methods (Getters and Setters)**: Public member functions are provided to access or modify the private variables. These methods control the data flow, ensuring proper validation or processing if needed.

**Example:**

```
class Person {

 private:

string name; // Private variable

 int age; // Private variable

public:

 // Getter for name

string getName() {

 return name;

 }

// Setter for name

void setName(string n) {

name = n;

} // Getter for age
```

```
 int getAge() {
 return age;
}
// Setter for age
 void setAge(int a) {
 if (a > 0) age = a; // Age validation
}
};
```

-By Aakash

-By Aakash