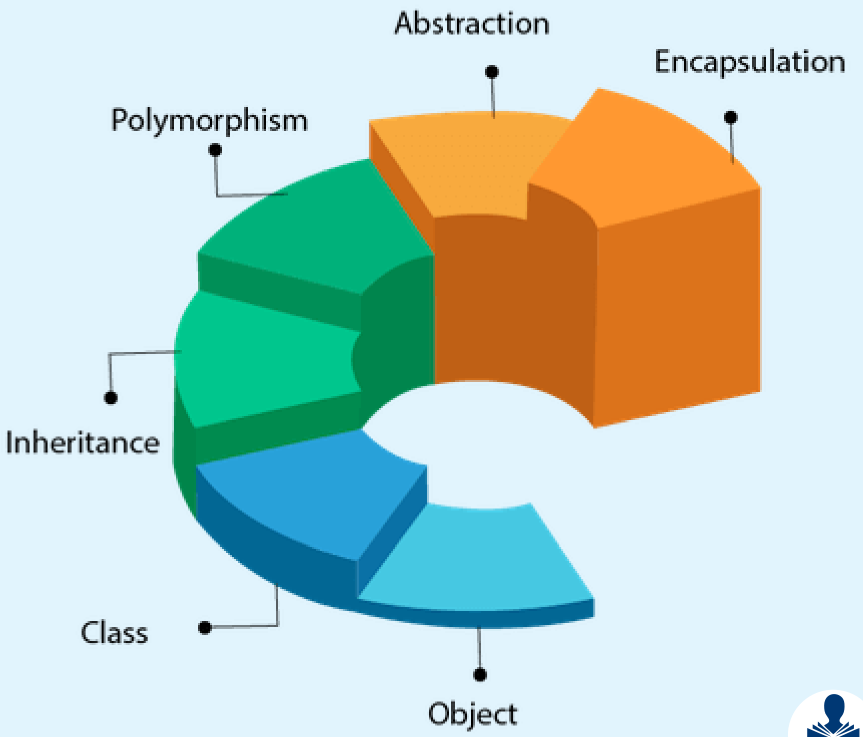
MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR
OPEN & DISTANCE EDUCATION

Object Oriented Programming Concepts

Master of Computer Applications (MCA)
Semester - 1



SELF LEARNING MATERIAL

MATS UNIVERSITY

www.matsuniversity.ac.in

NAAC GRADE A+ ACCREDITED UNIVERSITY

Master of Computer Applications
ODL MCA-101
Object Oriented Programming Concepts

Course Introduction	**1**
Module 1 **Programming Paradigms**	**7**
Unit 1: Programming Language Concepts	8
Unit 2: Types of Programming Language and Its Application Area	13
Unit 3: File Creation, Compilation and Linking	20
Module 2 **Classes, Objects, Constructors, and Destructors**	**41**
Unit 4: Object Oriented Programming Concepts, Advantage	42
Unit 5: Object and Class	46
Unit 6: Member Function	53
Module 3 **Inheritance and Polymorphism**	**102**
Unit 7: Operator Overloading: Unary and Binary	103
Unit 8: Overloading Binary Operators Using Friends	107
Unit 9: Rules of Overloading Operators, Type Conversion	110
Module 4 **Operator Overloading and Type Conversion**	**140**
Unit 10: Pointers	141
Unit 11: Virtual Function, Pure Virtual Function	154
Unit 12: Polymorphism: Compile Time, Run Time	160
Module 5 **Exception Handling and File Handling**	**173**
Unit 13: Stream Classes	174
Unit 14: File Handling in OOP’s	179
References	182

COURSE COORDINATOR

Dr. Abhishek Guru, Associate Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh.

COURSE PREPARATION

Dr. Abhishek Guru, Associate Professor and Ms. Arifa Khan, Assistant Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh.

March, 2025

ISBN: 978-93-49916-18-0

@MATS Centre for Distance and Online Education, MATS University, Village-Gullu, Aarang, Raipur-(Chhattisgarh)

Printed & published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. Meghanadhudu Katabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

This **Object-Oriented Programming (OOP)** using C++ is an essential course designed to introduce students to modern programming techniques that enhance code reusability, scalability, and efficiency. This course provides a strong foundation in object-oriented concepts such as classes, objects, inheritance, polymorphism, operator overloading, type conversion, and exception handling. By learning these concepts, students will be able to design robust and maintainable software applications. The course is structured into five Modules, each covering fundamental aspects of OOP using C++.

Module 1: Programming Paradigms

Introduces different programming approaches, including procedural, object-oriented, functional, and logical paradigms. It emphasizes the need for object-oriented programming and explains key OOP principles such as abstraction, encapsulation, inheritance, and polymorphism. Students will understand how OOP differs from procedural programming and why it is widely used in modern software development.

Module 2: Classes, Objects, Constructors, and Destructors

delves into the core building blocks of OOP in C++. Students will learn how to define and use classes and objects effectively. This Module also explores constructors, which help initialize objects, and destructors, which manage resource cleanup. Concepts such as default, parameterized, and copy constructors are covered to enhance students' understanding of object creation and memory management.

Module 3: Inheritance and Polymorphism

Focuses on one of the most powerful features of OOP—code reusability. It covers different types of inheritance, including single, multiple, multilevel, hierarchical, and hybrid inheritance. Students will learn how derived classes inherit properties from base classes, along with function overriding and virtual functions to achieve runtime polymorphism. The concept of dynamic method dispatch is introduced to enable flexible and scalable software design.

Module 4: Operator Overloading and Type Conversion

Students explore how operators can be customized to work with user-defined data types. The Module covers the rules and restrictions of operator overloading and demonstrates how unary and binary operators can be overloaded. Additionally, students will understand type conversion techniques, including implicit and explicit conversions, and how they can be applied between basic types and class types for seamless data manipulation.

Module 5: Exception Handling and File Handling

Students learn the skills to develop robust and error-free applications. This Module covers the concepts of errors and exceptions and explains how exception handling mechanisms such as try, catch, and throw can be used to manage runtime errors efficiently. Students will also learn how to handle multiple exceptions and create user-defined exceptions, ensuring that their programs remain stable even under unexpected conditions. File handling practices will also taught to students.

MODULE 1
PROGRAMMING PARADIGMS

LEARNING OUTCOMES

By the end of this module, students will be able to:

- Understand programming language concepts and their significance.
- Identify types of programming languages and their applications.
- Explain source file creation, compilation, and linking.
- Describe the features and structure of a C++ program.
- Define and differentiate data types, keywords, identifiers, variables, constants, and operators.
- Implement control statements for branching, looping, and jumping.
- Understand array declaration, initialization, and element access.
- Differentiate between types of arrays and their usage.

Unit 1: Programming Language Concepts

1.1 Programming Language Concepts

A programming language is a formal set of instructions that enables humans to communicate with computers and create software applications. It provides a structured way to define logic, process data, and control hardware operations. Over the years, programming languages have evolved to improve efficiency, readability, and modularity. This evolution has led to different programming paradigms, including procedural, object-oriented, functional, and declarative programming. Understanding the core concepts of programming languages is crucial for writing efficient, maintainable, and scalable code. These concepts form the foundation of software development and enable programmers to solve real-world problems using computational techniques.

Syntax and Semantics: Every programming language follows a set of rules that dictate how instructions should be written and interpreted. These rules are divided into two main aspects:

a) Syntax refers to the grammatical structure of a programming language. It defines how statements must be written, including keywords, symbols, and punctuation. For example, in C++, a statement must end with a semicolon (;).

b) Semantics refers to the meaning behind the written code. It ensures that a program performs the intended operations correctly. Even if a program has correct syntax, it may not produce the desired output if its semantics are flawed.

For instance, consider the following C++ statement:

int x = "Hello"; // Syntax is correct, but semantics are incorrect (type mismatch)

Here, x is declared as an integer but assigned a string value, which causes a semantic error.

High-Level vs. Low-Level Languages: Programming languages are categorized into high-level and low-level languages based on their abstraction from machine code.

a) **Low-Level Languages:** These include machine language (binary code) and assembly language, which are closely related to hardware instructions. They offer high performance but are difficult to write and maintain. Example: Assembly language.

b) **High-Level Languages:** These include languages like C++, Java, and Python, which provide human-readable syntax and abstract away hardware details. High-level languages enhance productivity and ease of development.

Example of an assembly language instruction:

```
MOV AX, 5   ; Moves the value 5 into register AX
```

In contrast, a high-level language like C++ simplifies this operation:

```
int x = 5;
```

Compilation and Interpretation: Programming languages are executed using two primary approaches: compilation and interpretation.

a) Compiled Languages: Languages like C and C++ require a compiler to convert the entire code into machine language before execution. This process improves performance but makes debugging slower.

b) Interpreted Languages: Languages like Python and JavaScript use an interpreter to execute code line by line, allowing immediate feedback but potentially reducing execution speed.

Example of a simple C++ program compiled before execution:

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

Here, the compiler converts the entire program into an executable file before running it.

Static vs. Dynamic Typing: Programming languages follow different typing systems to handle variables and data types:

a) Static Typing: In statically typed languages (e.g., C++, Java), variable types are declared explicitly and checked at compile-time.

b) Dynamic Typing: In dynamically typed languages (e.g., Python, JavaScript), variable types are determined at runtime, offering flexibility but increasing the risk of runtime errors.

Example of static typing in C++:

```
int num = 10; // The type (int) is explicitly declared
```

Example of dynamic typing in Python:

```
num = 10  # Type is inferred dynamically
```

Object-Oriented vs. Procedural Programming: Programming languages can follow different paradigms, with two of the most common being procedural programming and object-oriented programming (OOP).

a) Procedural Programming: Based on a sequence of instructions executed step-by-step. It uses functions to break down tasks but does not encapsulate data. Example: C language.

b) Object-Oriented Programming (OOP): Organizes code into objects and classes, encapsulating data and behavior. It supports features like inheritance, polymorphism, and encapsulation, making code more modular and reusable. Example: C++, Java, Python.

Example of procedural programming in C:

```
#include <stdio.h>
void greet() {
   printf("Hello, World!");
}
int main() {
   greet();
   return 0;
}
```

Example of object-oriented programming in C++:

```
#include <iostream>
using namespace std;
class Greeting {
public:
   void sayHello() {
      cout << "Hello, World!";
   }
};
int main() {
   Greeting obj;
   obj.sayHello();
   return 0;
```

}

Memory Management: Programming languages handle memory allocation and deallocation differently:

a) Manual Memory Management: In languages like C and C++, developers must allocate (new) and free (delete) memory explicitly.

b) Automatic Memory Management: In languages like Python and Java, a garbage collector automatically reclaims unused memory.

Example of manual memory allocation in C++:

```
int* ptr = new int(10);  // Dynamically allocated memory
delete ptr;              // Manually deallocated memory
```

In contrast, in Python, memory is managed automatically:

```
num = 10  # Memory is allocated and managed by Python's garbage collector
```

Standard Libraries and APIs: Modern programming languages provide standard libraries and APIs to simplify development:

a) Standard Libraries: Built-in functions for mathematical operations, file handling, and data structures. Example: C++ Standard Library (STL).

b) Application Programming Interfaces (APIs): Predefined functions that allow programs to interact with external services or hardware. Example: REST APIs in web development.

Example of using the C++ Standard Library:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
   vector<int> numbers = {1, 2, 3, 4, 5};
   for (int num : numbers) {
      cout << num << " ";
   }
   return 0;
}
```

Programming languages serve as the foundation for software development, providing structured methods to write, execute, and manage code efficiently. Understanding key concepts such as syntax, typing systems, compilation, paradigms, and memory management is essential for mastering software development. This knowledge will form the basis for learning Object-Oriented Programming (OOP) in C++, which we will explore in the upcoming sections.

Unit 2: Types of Programming Language and Its Application Area

1.2 Types of Programming Language and Its Application Area

Programming languages serve as the fundamental medium through which humans communicate with computers to develop software and applications. Over the years, these languages have evolved significantly, leading to the development of various categories based on abstraction levels, execution models, and programming paradigms. Each programming language is designed to address specific computational challenges, making it crucial for software developers to understand their classifications and application areas. Broadly, programming languages can be classified into low-level and high-level languages based on their closeness to machine hardware, and further into various paradigms such as procedural, object-oriented, functional, and scripting languages. The correct choice of a programming language depends on the nature of the task, performance requirements, and ease of development.

Low-Level and High-Level Languages: Programming languages are first categorized based on their level of abstraction from the underlying hardware. Low-level languages, which include machine language and assembly language, are closer to the hardware, making them highly efficient but difficult to program. Machine language consists of binary code (0s and 1s), which is directly executed by the computer’s CPU without any translation. However, since writing programs in binary is complex and error-prone, assembly language was introduced as an improvement. Assembly language uses symbolic representations, known as mnemonics, to make programming more readable while still being closely tied to the hardware. Assembly programs must be translated into machine code using an assembler. These low-level languages are mostly used in system programming, embedded systems, and real-time applications where direct hardware interaction is required. In contrast, high-level languages provide a greater degree of abstraction and are designed to be more human-readable. These languages are further categorized into procedural, object-oriented, functional, scripting, and logic-based languages, each serving different programming needs and application areas.

Procedural Programming Languages: Procedural languages follow a structured, step-by-step approach to program execution. These languages focus on how a task should be accomplished by dividing programs into functions, loops, and conditional statements. A key feature of procedural programming is the use of functions that enable code reusability and modularity. Examples of procedural programming languages include C, Fortran, Pascal, and COBOL. These languages are widely used in scientific computing, system programming, and business applications. For instance, C is extensively used in developing operating systems, while COBOL is utilized for business applications in the financial sector. Procedural programming is effective for developing software where a sequential flow of execution is necessary.

Object-Oriented Programming (OOP) Languages: The object-oriented programming (OOP) paradigm was introduced to overcome the limitations of procedural programming by emphasizing real-world modeling using objects and classes. Object-oriented languages support essential concepts such as encapsulation, inheritance, and polymorphism, making them highly suitable for large-scale software development. OOP provides better modularity, code reusability, and maintainability. Popular object-oriented languages include C++, Java, Python, and C#, all of which are widely used in application development, enterprise software, and game development. For example, Java is extensively used in Android app development, while C++ is preferred for high-performance game engines and real-time applications. By encapsulating data and functions within objects, OOP promotes cleaner and more manageable code structures, making it a preferred paradigm for modern software engineering.

Functional Programming Languages: Functional programming languages take a mathematical approach to problem-solving by treating functions as first-class citizens. Unlike procedural and object-oriented programming, which rely on changing states and variables, functional programming emphasizes immutability and recursion. This makes it well-suited for applications that require concurrency and parallel execution. Functional programming languages such as Haskell, Lisp, Scala, and Erlang are widely used in artificial intelligence (AI), data science, and financial modeling. A key advantage of functional programming is that it minimizes side effects, leading to more predictable and reliable code. For example, Erlang is used in building highly concurrent telecom systems, while Haskell is preferred for complex mathematical computations. Functional programming is gaining popularity due to its ability to handle large-scale distributed systems efficiently.

Scripting Languages: Scripting languages are typically interpreted rather than compiled, making them easier to learn and use. These languages are designed for automation, web development, and rapid prototyping. Unlike compiled languages, which require a separate compilation step before execution, interpreted languages execute code line by line, allowing for faster development and debugging. Popular scripting languages include Python, JavaScript, PHP, Perl, and Bash. Python is widely used in data science, artificial intelligence, and machine learning, while JavaScript is essential for web development and front-end programming. PHP is primarily used for server-side web development, powering dynamic websites and content management systems like WordPress. Scripting languages offer flexibility and ease of development, making them ideal for small-scale projects and automation tasks.

Logic Programming Languages: Logic programming is a paradigm based on formal logic, where programs are expressed as a set of rules and facts rather than step-by-step instructions. Prolog (Programming in Logic) is the most well-known logic programming language, widely used in expert systems, natural language processing, and artificial intelligence applications. In Prolog, a program consists of rules that define relationships between entities. When a query is made, the logic engine processes the rules and facts to derive a solution. This approach makes logic programming well-suited for applications requiring complex reasoning and decision-making.

Domain-Specific Languages (DSLs): While general-purpose languages can be used for a wide range of applications, some languages are designed for specific domains, known as domain-specific languages (DSLs). These languages are tailored to a particular problem area, making them highly efficient within their niche. Examples of DSLs include SQL (Structured Query Language) for database management, MATLAB for scientific computing, R for statistical analysis, and HTML/CSS for web development. SQL, for instance, is the industry standard for managing relational databases, allowing users to perform complex queries efficiently. Similarly, R and MATLAB are extensively used in academia and research for statistical modeling and data analysis. By focusing on specific problem domains, DSLs provide optimized solutions that general-purpose languages cannot easily achieve.

Compiled vs. Interpreted Languages: Programming languages can also be classified based on their execution model—whether they are compiled or interpreted. Compiled languages translate the entire source code into machine code before execution, resulting in faster performance. Examples include C, C++, and Java (via the JVM). Compiled programs run efficiently but require a compilation step before execution, making debugging more time-consuming. On the other hand, interpreted languages execute code line by line using an interpreter, making development faster but execution slower. Examples of interpreted languages include Python, JavaScript, and PHP. While interpreted languages provide greater flexibility, they are generally slower than compiled languages. Some modern languages, such as Java, use a hybrid approach, where code is first compiled into an intermediate bytecode and then interpreted by a virtual machine (JVM).

Table 1.1 Difference between two Languages

Feature	Compiled Languages	Interpreted Languages
Execution Process	Entire source code is compiled into machine code before execution.	Code is executed line-by-line by an interpreter.
Speed & Performance	Faster execution since the program is already translated into machine code.	Slower execution due to on-the-fly translation.
Error Handling	Errors are detected at compile time, requiring recompilation after fixing.	Errors are detected at runtime, making debugging easier.
Portability	Less portable since compiled code is specific to a system's architecture.	More portable as the source code can be executed on any system with an

		interpreter.
Dependency	Requires a compiler for translation.	Requires an interpreter to execute the code.
Examples	C, C++, Java (compiled to bytecode), Rust, Go	Python, JavaScript, PHP, Ruby
Use Cases	System programming, Game development, Performance-critical applications	Web development, Scripting, Rapid prototyping, Data analysis

Programming languages have evolved to meet the growing demands of software development, leading to various paradigms and classifications. Low-level languages offer efficiency and control, whereas high-level languages provide abstraction and ease of development. Procedural and object-oriented programming dominate mainstream application development, while functional and logic-based languages serve specialized computational needs. Scripting languages simplify automation and web development, while domain-specific languages optimize problem-solving in specialized fields. Understanding the strengths and application areas of different programming languages enables developers to select the best tools for their projects. In the next section, we will explore the process of source file creation, compilation, and linking, which are essential steps in executing programs efficiently.

Table 1.2 Classification of programming languages along with their specific application areas.

Programming Language Type	Description	Examples	Application Areas
Low-Level Languages	Close to machine hardware, offering high performance but difficult to program.	Assembly, Machine Code	System programming, Embedded systems, Hardware control

Procedural Languages	Follow a structured, step-by-step approach using functions and loops.	C, Fortran, Pascal, COBOL	System software, Scientific computing, Business applications
Object-Oriented Programming (OOP) Languages	Use classes and objects to structure programs with encapsulation, inheritance, and polymorphism.	C++, Java, Python, C#	Application development, Enterprise software, Game development
Functional Programming Languages	Emphasize immutability, recursion, and first-class functions.	Haskell, Lisp, Scala, Erlang	AI & Machine Learning, Data Science, Parallel computing
Scripting Languages	Typically interpreted, used for automation and web development.	Python, JavaScript, PHP, Bash, Perl	Web development, System automation, Data analysis
Logic Programming Languages	Use formal logic and rule-based programming for decision-making.	Prolog, Datalog	AI, Expert systems, Knowledge-based reasoning
Domain-Specific Languages (DSLs)	Designed for specific application areas, optimized for particular tasks.	SQL, R, MATLAB, HTML/CSS	Databases, Statistical modeling, Scientific computing, Web design
Compiled Languages	Convert source code into machine code before execution	C, C++, Java (JVM-based)	High-performance applications, Operating

	for better performance.		systems, Game engines
Interpreted Languages	Execute code line-by-line using an interpreter, making debugging easier.	Python, JavaScript, PHP	Web development, Scripting, Rapid prototyping

Unit 3: File Creation, Compilation and Linking

1.3 Source File Creation, Compilation and Linking

C++ is a powerful, general-purpose programming language that combines the efficiency of procedural programming with the flexibility of object-oriented programming (OOP). Developed by Bjarne Stroustrup in the early 1980s as an extension of C, C++ provides robust features that make it suitable for system programming, game development, large-scale applications, and performance-critical software. Understanding the features of C++ helps programmers leverage its strengths, while knowing the structure of a C++ program ensures that code is written in an organized, readable, and maintainable manner. This section explores the key features of C++ and provides a detailed breakdown of a well-structured C++ program.

Features of C++

C++ offers several advanced features that distinguish it from other programming languages. These features enable programmers to develop efficient and modular applications with enhanced performance and flexibility.

- Object-Oriented Programming (OOP): C++ is an object-oriented language, which means it follows the OOP principles of encapsulation, inheritance, polymorphism, and abstraction. These concepts allow for the creation of reusable and modular code, making software development more scalable and maintainable.
- High Performance and Efficiency: Since C++ is a compiled language, it converts source code into machine code before execution, ensuring faster performance compared to interpreted languages like Python or JavaScript. Additionally, C++ provides manual memory management, giving programmers greater control over resource allocation and optimization.
- Multi-Paradigm Programming: C++ supports multiple programming paradigms, including procedural, object-oriented, and generic programming. This flexibility allows developers to use the best approach for different types of applications.

- Strongly Typed and Statically Typed Language: C++ is strongly typed, meaning that type errors must be resolved before compilation. It is also statically typed, which means variable types are checked at compile-time rather than runtime. This helps in reducing runtime errors and improving performance.
- Memory Management with Pointers: C++ provides pointers and dynamic memory allocation using operators like new and delete. This enables efficient memory handling but also requires careful management to avoid memory leaks.
- Standard Template Library (STL): The Standard Template Library (STL) in C++ offers a collection of predefined classes and functions for common programming tasks such as data structures (vectors, lists, stacks, queues) and algorithms (sorting, searching). This enhances code efficiency and reduces development time.
- Operator Overloading: C++ allows operators like +, -, and * to be overloaded so that they can work with user-defined data types, enhancing code readability and usability.
- Platform Independence: Although C++ programs need to be compiled separately for different operating systems, the source code remains platform-independent, making it portable across different platforms.
- Low-Level and High-Level Features: C++ supports both low-level features (like direct memory manipulation) and high-level abstractions (like classes and objects), making it suitable for both system programming and application development.

Structure of a C++ Program: A well-structured C++ program consists of several components, each serving a specific purpose. Understanding the structure ensures that code is organized, readable, and efficient.

Basic Structure of a C++ Program

A C++ program generally follows this structure:

```
// 1. Header Files
#include <iostream>
// 2. Namespace Declaration
using namespace std;
// 3. Global Declarations (if any)
// 4. Function Prototypes (if required)
// 5. Main Function
int main() {
   // 6. Variable Declaration
   int num = 10;
      // 7. Function Call (if required)
   cout << "The number is: " << num << endl;

   return 0;
}

// 8. Function Definitions (if any)
```

Header Files: Header files contain predefined functions, classes, and macros that can be used in the program. They are included using the #include directive.

Example:

```
#include <iostream>  // Allows input and output operations
#include <cmath>       // Provides mathematical functions like sqrt(), pow()
```

Namespace Declaration: Namespaces prevent name conflicts by organizing code into separate scopes. The standard C++ library functions reside in the std namespace.

Example:

```
using namespace std;
```

Without using namespace std;, we would have to use std::cout and std::cin instead of cout and cin.

Global Declarations: Global variables are declared outside all functions and can be accessed from anywhere in the program.Example:

```
int globalVar = 100; // Accessible by all functions
```

Although global variables can be useful, excessive use is discouraged due to potential side effects and memory consumption.

Function Prototypes: In large programs, function prototypes are declared before main() to inform the compiler about functions used later in the program.

Example:

```
void displayMessage(); // Function prototype
```

Main Function (main()): Every C++ program must have a main() function, which serves as the program's entry point. Execution begins from main().

Example:

```
int main() {
    cout << "Hello, C++!" << endl;
    return 0;
}
```

The return 0; statement indicates successful execution to the operating system.

Variable Declaration: Variables store data that the program manipulates. C++ supports various data types such as int, float, char, double, and string.

Example:

```
int age = 25;
float temperature = 36.5;
char grade = 'A';
```

Function Calls: Functions are used to modularize the code, making it reusable and easier to manage. A function is defined separately and called in main().

Example:

```
void greet() {
    cout << "Welcome to C++ Programming!" << endl;
}

int main() {
    greet();  // Function call
    return 0;
}
```

Function Definitions: Functions implement reusable logic and are defined outside main().

Example:

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

Functions improve code maintainability and readability.

C++ is a feature-rich programming language that provides high performance, object-oriented capabilities, and extensive libraries.

Understanding its features, such as OOP, memory management, STL, and operator overloading, allows programmers to write efficient and scalable applications. Additionally, following a structured approach to writing C++ programs—by including header files, proper variable declarations, and function modularization—ensures that code remains organized, readable, and maintainable. In the next section, we will explore data types, tokens, keywords, identifiers, variables, constants, and operators, which form the fundamental building blocks of C++ programming.

Table 1.3 Common Compilation Errors and Fixes

Error Type	Description	Solution
Syntax Error	Incorrect syntax (e.g., missing semicolon).	Fix syntax and recompile.
Linker Error	Undefined reference to a function.	Ensure proper function declaration and linking.
Runtime Error	Issues that occur during execution (e.g., division by zero).	Debug and handle exceptions.
Segmentation Fault	Accessing invalid memory (e.g., dereferencing null pointers).	Check pointers and memory management.

The source file creation, compilation, and linking process are fundamental steps in C++ programming. The source file contains the program logic, which is converted into machine code through the compilation process. The linker then integrates object files and external libraries, producing an executable file that can be run on a computer. Understanding these stages helps programmers debug errors, optimize performance, and work efficiently on multi-file projects.

1.3 Features and Structure of C++ Program

C++ is a widely used, high-performance programming language that blends the features of procedural programming with object-oriented programming (OOP), making it a powerful tool for software development. It was developed by Bjarne Stroustrup in the early 1980s as an extension of the C language and has since evolved into a feature-rich language used in various domains, including system programming, game development, real-time simulations, database management, and large-scale enterprise applications. One of the key reasons for C++'s widespread adoption is its ability to provide low-level memory manipulation while also supporting high-level abstractions that enhance modularity and code reusability.

To become proficient in C++, it is essential to understand both its features and structural organization. The features of C++ highlight its unique capabilities that differentiate it from other programming languages, while its structure defines the way in which a C++ program is written, organized, and executed. This Module provides a detailed explanation of the core features of C++ and a structured breakdown of a typical C++ program, ensuring that students develop a strong foundation in the language.

1.4.1 Features of C++

C++ has a broad range of features that make it versatile, powerful, and efficient. These features allow it to be used in various domains, from low-level system programming to high-level application development. Below is a detailed discussion of the key features of C++:

1. Object-Oriented Programming (OOP)

One of the most significant advancements in C++ over its predecessor, C, is the introduction of Object-Oriented Programming (OOP). OOP is a programming paradigm that models real-world entities using objects and classes, promoting code reusability, scalability, and modularity. C++ supports four key principles of OOP:

- Encapsulation: The bundling of data (variables) and methods (functions) within a class to prevent unauthorized access.
- Inheritance: The ability of one class to acquire the properties and behaviors of another class, reducing redundancy.
- Polymorphism: The ability of a function or method to behave differently based on the context in which it is used.
- Abstraction: Hiding implementation details while exposing only the necessary functionalities to the user.

Example of OOP in C++:

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

class Car {
```

```
private:
    string brand;
public:
    Car(string b) { brand = b; } // Constructor
    void display() { cout << "Car Brand: " << brand << endl; }
};

int main() {
    Car myCar("Toyota");
    myCar.display();
    return 0;
}
```

In this example, the class Car encapsulates data (brand) and behavior (display() function), demonstrating OOP principles.

2. Multi-Paradigm Support

C++ is a multi-paradigm language, meaning it supports multiple styles of programming, including:

Table 1.4 Different Types of Paradigm

Paradigm	Description	Example Languages
Procedural	Step-by-step instructions using functions.	C, Pascal
Object-Oriented	Uses objects and classes to model real-world entities.	C++, Java
Generic	Uses templates to write type-independent functions and classes.	C++, D, Rust

This flexibility allows programmers to select the best programming paradigm based on the problem they are solving.

3. High Performance and Efficiency

Since C++ is a compiled language, it translates the entire source code into machine code before execution, leading to faster performance compared to interpreted languages like Python. Additionally, C++ provides manual memory management, allowing developers to optimize memory usage and prevent unnecessary resource consumption. This makes C++ ideal for performance-intensive applications like gaming, embedded systems, and real-time simulations.

Example of compiled C++ code execution using GCC:

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

This command first compiles the source code and then executes the generated binary file.

4. Strongly Typed Language with Static Typing

C++ is a strongly typed language, meaning that each variable must have a specific type that cannot be changed during execution. It is also statically typed, meaning that type-checking occurs at compile time rather than at runtime.

Example:

```
int num = 10;
num = "Hello"; // Error: Type mismatch
```

This prevents unexpected errors and improves code reliability.

5. Memory Management with Pointers

Unlike many high-level languages, C++ allows direct memory manipulation through pointers, providing greater control over memory allocation and deallocation. This is particularly useful in system programming and embedded systems, where efficient memory management is critical.

Example of pointer usage in C++:

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

int main() {
   int x = 10;
   int *ptr = &x; // Pointer stores the address of x

   cout << "Value of x: " << *ptr << endl; // Dereferencing pointer
   return 0;
}
```

Here, the pointer ptr stores the memory address of x and accesses its value using the * operator.

1.4.2 Structure of a C++ Program

A well-structured C++ program consists of several key components that define its execution flow. Understanding these components is crucial for writing clean, efficient, and maintainable code.

Basic Structure of a C++ Program

```
// 1. Header Files
#include <iostream>

// 2. Namespace Declaration
using namespace std;

// 3. Global Declarations (if any)

// 4. Function Prototypes (if required)

// 5. Main Function
int main() {
    // 6. Variable Declaration
    int num = 10;

    // 7. Function Call (if required)
    cout << "The number is: " << num << endl;

    return 0;
}

// 8. Function Definitions (if any)
```

Table 1.5 Explanation of Components

Component	**Description**
Header Files	Contain standard C++ libraries like <iostream>, <cmath>.
Namespace Declaration	Allows the use of functions like cout without std:: prefix.
Global Declarations	Variables that can be accessed by all functions in the program.
Function Prototypes	Declares functions before their definition for better modularity.
Main Function (main())	Entry point of the program where execution starts.
Variable Declaration	Defines variables to store data in memory.
Function Calls	Executes predefined functions to perform specific tasks.

Function Definitions	Implements the logic of user-defined functions.

C++ is a powerful, versatile, and high-performance language that supports object-oriented programming, manual memory management, operator overloading, and multiple paradigms. These features make it a preferred choice for system programming, application development, and real-time computing. A well-structured C++ program follows a logical organization, starting from header files and function declarations to variable initialization and function execution. By mastering these fundamental concepts, students can develop efficient, scalable, and maintainable C++ applications.

In the next section, we will explore data types, tokens, keywords, identifiers, variables, constants, and operators, which form the fundamental building blocks of C++ programming.

Data Types in C++

Data types define the type of data a variable can store. C++ provides several types of data types:

Table 1.6 Primary Data Types

Data Type	**Size (Bytes)**	**Description**	**Example**
int	4	Stores integers (whole numbers)	int age = 25;
float	4	Stores floating-point numbers (decimal values)	float price = 99.99;
double	8	Stores large floating-point numbers	double pi = 3.14159;
char	1	Stores single characters	char grade = 'A';
bool	1	Stores boolean values (true or false)	bool isPassed = true;

Derived Data Types

- **Array:** int arr[5] = {1, 2, 3, 4, 5};
- **Pointer:** int *ptr;
- **Reference:** int &ref = x;

User-defined Data Types

- **Structure:** struct Student { string name; int age; };
- **Class:** class Car { public: string brand; };

- **Enumeration (enum):** enum Color { RED, GREEN, BLUE };

Tokens in C++

Tokens are the smallest Modules in a C++ program. These include:

1. Keywords
2. Identifiers
3. Variables and Constants
4. Operators

Keywords in C++

Keywords are reserved words in C++ that have predefined meanings. Some commonly used keywords are:

int, float, double, char, bool, if, else, while, for, switch, case, break, continue, return, void, struct, class, public, private, protected, namespace, new, delete, this, virtual, friend, etc.

Identifiers in C++

Identifiers are the names given to variables, functions, arrays, and objects.

Rules for Identifiers:

- Must begin with a letter (A-Z or a-z) or an underscore _
- Cannot be a keyword
- Must be unique and case-sensitive

Example:

```
int studentAge;  // Valid
float _salary;   // Valid
int 2marks;      // Invalid (cannot start with a number)
```

2.3 Variables and Constants in C++

Variables:

A variable is a named storage location in memory.

```
int age = 20;
float price = 99.99;
```

Constants:

A constant is a value that does not change during program execution.

- **Using const keyword:**
  const float PI = 3.14159;
- **Using #define preprocessor directive:**
  #define MAX_SIZE 100

Operators in C++

Operators perform operations on variables and values.

Types of Operators:

1. **Arithmetic Operators:** +, -, *, /, %
2. **Relational Operators:** ==, !=, <, >, <=, >=
3. **Logical Operators:** &&, ||, !
4. **Assignment Operators:** =, +=, -=, *=, /=, %=
5. **Bitwise Operators:** &, |, ^, ~, <<, >>
6. **Increment/Decrement Operators:** ++, --
7. **Ternary Operator:** condition ? expr1 : expr2;
8. **Type Casting Operator:** (dataType)value;

Example:

```
int a = 10, b = 20;
int sum = a + b;   // Addition
bool result = (a < b);  // Relational operator
```

3. Control Statements in C++

Control statements control the flow of execution in a program. These are categorized into:

1. Branching Statements **(Decision Making)**
2. Looping Statements **(Iteration)**
3. Jumping Statements **(Control Transfer)**

Branching Statements (Decision Making)

Branching statements are used to execute different code blocks based on conditions.

1. if Statement

```
if (condition) {
   // Code to execute if condition is true
}
```

Example:

```
int num = 10;
if (num > 0) {
   cout << "Positive number";
}
```

2. if-else Statement

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

Example:

```
int num = -5;
if (num > 0) {
    cout << "Positive";
} else {
    cout << "Negative";
}
```

3. if-else-if Ladder

```
if (condition1) {
    // Code
} else if (condition2) {
    // Code
} else {
    // Code
}
```

4. switch Statement

Used for multiple conditions.

```
switch (expression) {
    case value1:
        // Code
        break;
    case value2:
        // Code
        break;
    default:
        // Code
}
```

Example:

```
int choice = 2;
switch (choice) {
    case 1: cout << "One"; break;
    case 2: cout << "Two"; break;
    default: cout << "Invalid";
}
```

Looping Statements (Iteration)

Looping statements execute a block of code multiple times.

1. for Loop

```
for (initialization; condition; increment/decrement) {
    // Code to execute
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

2. while Loop

```
while (condition) {
    // Code to execute
}
```

Example:

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

3. do-while Loop

Executes at least once before checking the condition.

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

Example:

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

Jumping Statements (Control Transfer)

Jumping statements alter the normal sequence of execution.

1. break Statement

Exits the loop or switch statement.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    cout << i << " ";
}
```

2. continue Statement

Skips the current iteration and moves to the next iteration.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    cout << i << " ";
}
```

3. goto Statement

Jumps to a labeled statement.

```
goto label;
label:
cout << "Jumped here";
```

This Module covers the **basics of C++ programming**, including **data types, tokens, operators, and control statements** with easy-to-understand explanations and code examples.

Arrays in C++

1. Array Declaration and Initialization

An **array** is a collection of elements of the same data type stored in contiguous memory locations. It allows storing multiple values using a single variable name.

Declaration of an Array

The syntax for declaring an array in C++ is:

```
data_type array_name[array_size];
```

Example:

```
int numbers[5]; // Declaring an array of 5 integers
```

Here, numbers is an integer array that can hold 5 values.

Array Initialization

Arrays can be initialized at the time of declaration:

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

If the size is omitted, the compiler automatically determines it based on the number of elements:

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

For character arrays (strings):

```
char name[] = "Hello"; // Automatically adds '\0' (null character)
```

2. Accessing Array Elements

Each element in an array is accessed using an **index** (starting from 0).

Syntax:

```
array_name[index];
```

Example:

```
#include <iostream>
```

```
using namespace std;

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

    cout << "First element: " << numbers[0] << endl;
    cout << "Third element: " << numbers[2] << endl;

    return 0;
}
```

Output:

```
First element: 10
Third element: 30
```

We can also modify array elements:

```
numbers[1] = 25; // Changing the second element to 25
```

Using Loops to Access Array Elements

To access all elements, we can use a loop:

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

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

    for(int i = 0; i < 5; i++) {
        cout << "Element at index " << i << ": " << numbers[i] << endl;
    }

    return 0;
}
```

3. Types of Arrays

C++ supports different types of arrays:

1. One-Dimensional Array

A simple linear list of elements. Example:

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

2. Two-Dimensional Array (2D Array)

Used to represent a **matrix or table** of elements.

Declaration:

```
data_type array_name[rows][columns];
```

Example:

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

Accessing 2D Array Elements:

cout << matrix[1][2]; // Accesses the element at row index 1, column index 2 (Output: 6)

Using Loops to Print a 2D Array:

```
#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;
}
```

3. Multi-Dimensional Array

An array with more than two dimensions. Example (3D Array):

```
int arr[2][2][3] = {
    {
        {1, 2, 3}, {4, 5, 6}
    },
    {
        {7, 8, 9}, {10, 11, 12}
    }
};
```

4. Dynamic Arrays (Using Pointers and new Operator)

Arrays with dynamic memory allocation:

```
int* arr = new int[5];  // Allocates memory for 5 integers
arr[0] = 10;
delete[] arr;  // Free memory
```

MCQs:

1. Which of the following programming paradigms emphasizes the use of functions and avoids changing state or mutable data?

A. Procedural programming

B. Object-oriented programming

C. Functional programming

D. Logical programming

2. In which programming paradigm are programs typically organized around objects and classes?

A. Procedural

B. Functional

C. Logical

D. Object-oriented

3. Which of the following is NOT a core principle of Object-Oriented Programming (OOP)?

A. Abstraction

B. Encapsulation

C. Compilation

D. Inheritance

4. What does encapsulation in OOP primarily help with?

A. Running code faster

B. Hiding internal details and protecting data

C. Writing functional expressions

D. Deriving new classes from existing ones

5. What is the main difference between procedural and object-oriented programming?

A. Procedural programming uses functions, OOP uses if-else statements

B. Procedural programming focuses on the “what,” OOP focuses on the “how”

C. Procedural programming structures code as procedures or routines; OOP structures code around objects and data

D. There is no difference

6. Which OOP principle allows objects to take on many forms through method overriding or overloading?

A. Inheritance

B. Polymorphism

C. Encapsulation

D. Abstraction

7. Which programming paradigm is based on formal logic and uses rules and facts to derive conclusions?

A. Object-oriented

B. Functional

C. Logical

D. Procedural

8. Why is Object-Oriented Programming widely used in modern software development?

A. It executes faster than other paradigms

B. It is only used in mobile app development

C. It promotes code reuse, scalability, and maintainability

D. It doesn't require any planning or design

9. What is abstraction in OOP?

A. Deriving new classes from existing ones

B. Representing only essential features while hiding unnecessary details

C. Storing variables in memory

D. Writing conditional logic

10. In OOP, what is inheritance used for?

A. Reducing function calls

B. Sharing code between unrelated classes

C. Allowing a class to acquire properties and methods from another class

D. Increasing program speed

Short Questions:

1. What is a programming paradigm?
2. How does procedural programming structure a program?
3. Define object-oriented programming in your own words.
4. What is the main goal of functional programming?
5. How is logical programming different from other paradigms?
6. List two key differences between procedural and object-oriented programming.

7. Why is object-oriented programming considered suitable for large and complex software systems?
8. What is abstraction in object-oriented programming? Provide an example.
9. Explain the concept of encapsulation and how it enhances data security.
10. What is inheritance in OOP, and how does it promote code reuse?
11. Describe polymorphism and give a real-world analogy.
12. Mention two advantages of using object-oriented programming over procedural programming.

Long Questions:

1. Explain the main characteristics of procedural programming. How does it handle data and functions? Provide examples.
2. Discuss the core concepts of functional programming. How does this paradigm differ from procedural and object-oriented approaches?
3. Describe the logical programming paradigm. What is its basis, and in what types of applications is it most commonly used?
4. Compare and contrast procedural programming and object-oriented programming. Highlight the strengths and limitations of each approach.
5. Why has object-oriented programming become the preferred paradigm in modern software development? Discuss its advantages with examples.
6. Define and explain the concept of abstraction in object-oriented programming. Why is it important in managing complexity in software systems?
7. What is encapsulation in OOP? How does it help in protecting the internal state of an object and ensuring data integrity?
8. Explain inheritance with the help of a real-world analogy. How does inheritance contribute to reusability and hierarchical classification in software design?
9. Define polymorphism in object-oriented programming. Differentiate between compile-time and run-time polymorphism with examples.

10. How do the principles of OOP—abstraction, encapsulation, inheritance, and polymorphism—work together to support scalable and maintainable code?
11. Discuss how different programming paradigms (procedural, object-oriented, functional, and logical) address the problem-solving process. Which paradigm do you think is most effective, and why?
12. Imagine you are designing a large software application (e.g., an online shopping platform). Explain why object-oriented programming would be a better fit than procedural programming for this task.

MODULE 2
CLASS, OBJECT, CONSTRUCTOR AND DESTRUCTOR

LEARNING OfUTCOMES

By the end of this Module, students will be able to:

- Understand Object-Oriented Programming (OOP) concepts and their advantages.
- Define and differentiate objects and classes in C++.
- Explain the role of member functions in class operations.
- Implement arrays within a class for structured data storage.
- Analyze memory allocation mechanisms for objects.
- Understand the purpose and use of friend functions in C++.
- Explore the concept of local classes and their applications.

Unit 4: Object Oriented Programming Concepts, Advantage

Paragraph 1: Core Concepts of Object-Oriented Programming

Object-Oriented Programming (OOP) is a paradigm that revolves around the concept of "objects," which are instances of "classes." A class acts as a blueprint, defining the properties (attributes) and behaviors (methods) that its objects will possess. Encapsulation is a fundamental principle of OOP, where data (attributes) and methods that operate on that data are bundled together within a single Module, the object. This bundling not only organizes code but also protects data from external interference, enhancing security and maintainability. Access modifiers, such as public, private, and protected, control the visibility and accessibility of these attributes and methods. Inheritance is another pivotal concept, enabling the creation of new classes (derived or child classes) that inherit properties and behaviors from existing classes (base or parent classes). This promotes code reusability and establishes a hierarchical structure, facilitating the modeling of real-world relationships. Polymorphism, meaning "many forms," allows objects of different classes to respond to the same method call in their own specific ways. This is achieved through method overloading (having multiple methods with the same name but different parameters within a class) and method overriding (providing a specific implementation of an inherited method in a derived class). Abstraction is the process of simplifying complex systems by modeling classes based on their essential properties and behaviors, hiding unnecessary details from the user. This allows developers to focus on the relevant aspects of an object, improving code clarity and reducing complexity. These concepts collectively form the foundation of OOP, enabling the creation of modular, maintainable, and scalable software systems that better represent real-world entities and interactions.

Paragraph 2: Advantages of Object-Oriented Programming

The advantages of Object-Oriented Programming (OOP) are numerous and have contributed significantly to its widespread adoption in software development. Firstly, OOP promotes code reusability through inheritance, allowing developers to create new classes based on existing ones, minimizing redundant code and saving development time. This reusability extends to the design phase, as well, where established class hierarchies can be adapted and extended for new applications. Encapsulation enhances data security by restricting direct access to an object's internal data, preventing unintended modifications and ensuring data integrity. This also simplifies maintenance, as changes to an object's internal implementation are less likely to affect other parts of the system. Modularity, another key advantage, is achieved by dividing a complex system into smaller, self-contained objects, each with its own responsibilities.

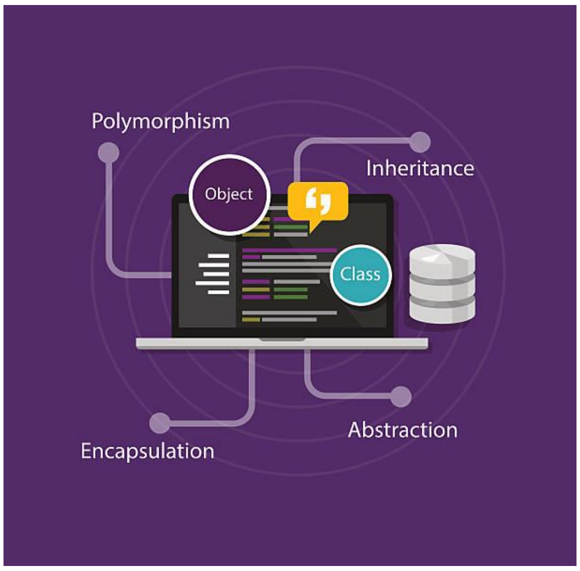


Figure 1 Core Structure of Object-Oriented Programming
[Source: https://www.istockphoto.com]

This modular structure makes it easier to understand, debug, and modify individual components without affecting the entire system. Polymorphism allows for greater flexibility and extensibility, as different objects can respond to the same method call in their own ways, enabling the creation of more adaptable and dynamic software. This adaptability is crucial in handling varying requirements and evolving systems. Furthermore, OOP facilitates better problem-solving by modeling real-world entities and relationships more accurately. The ability to abstract complex systems into simpler, manageable objects allows developers to focus on the essential aspects of a problem, leading to more efficient and effective solutions. The hierarchical structure provided by inheritance allows for intuitive organization of complex systems. Overall, OOP improves code organization, maintainability, and scalability, making it a powerful paradigm for developing large and complex software applications.

Paragraph 3: Practical Application and Real-World Impact of OOP

The practical application of Object-Oriented Programming (OOP) extends across diverse domains, demonstrating its versatility and effectiveness in solving real-world problems.

In software development, OOP is heavily used in building complex applications, from desktop software to web applications and mobile apps. Graphical User Interfaces (GUIs) are often built using OOP principles, where UI elements like buttons, windows, and menus are represented as objects with specific properties and behaviors. Game development relies heavily on OOP to model game entities, such as characters, environments, and items, allowing for complex interactions and simulations. In data management, database systems utilize OOP concepts to represent data as objects, enabling efficient data retrieval and manipulation. Enterprise applications, which often involve complex business logic and data structures, benefit significantly from OOP's modularity and reusability. In the realm of simulation and modeling, OOP is used to create realistic simulations of physical systems, biological processes, and financial models. Scientific computing leverages OOP to develop libraries and frameworks for complex calculations and data analysis. The impact of OOP is evident in the widespread adoption of languages like Java, C++, Python, and C#, which are designed to support OOP principles. These languages have empowered developers to create robust, scalable, and maintainable software systems that have transformed industries and improved daily life. The ability to model real-world entities and relationships accurately has led to more intuitive and user-friendly software experiences. Furthermore, the modularity and reusability of OOP have accelerated software development cycles and reduced maintenance costs, allowing organizations to respond more quickly to changing market demands. The principles of OOP have also influenced software design patterns and architectural styles, contributing to the development of better software engineering practices. In essence, OOP has become a cornerstone of modern software development, enabling the creation of complex and sophisticated systems that address a wide range of real-world challenges.

Unit 5: Object and Class

2.1 Objects and Classes in C++

1. Introduction to Object-Oriented Programming (OOP)

C++ is an **object-oriented programming (OOP)** language that focuses on objects and classes to structure programs efficiently. **OOP concepts** include **encapsulation, inheritance, polymorphism, and abstraction**, with **objects and classes** being the foundation.

What is a Class?

A **class** is a **user-defined data type** that acts as a blueprint for creating objects. It defines the **attributes (data members)** and **behavior (member functions)** of an object.

What is an Object?

An **object** is an **instance of a class**. When a class is defined, no memory is allocated until an object is created. Each object has its own copy of data members but shares the same functions.

2. Declaring a Class in C++

The syntax for defining a class:

```
class ClassName {
    // Access specifier
    private:
        // Data members (variables)
    public:
        // Member functions (methods)
};
```

Example: Defining a Class

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

// Class definition
class Car {
    public:
        string brand;
        int year;

        // Function to display car details
        void showDetails() {
            cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};

int main() {
  Car car1; // Object creation
  car1.brand = "Toyota";
  car1.year = 2022;

  car1.showDetails(); // Calling function

  return 0;
}
```

Output:

```
Brand: Toyota, Year: 2022
```

3. Access Specifiers in Classes

Access specifiers define the scope of class members. There are **three** main types:

1. Private (default)
   - Data members are only accessible inside the class.
   - Cannot be accessed directly by objects.

```
class Example {
  private:
    int secretNumber;
};
```

2. Public
   - Members can be accessed directly from outside the class.

```
class Example {
  public:
    int number;
};
```

3. Protected
   - Similar to private, but accessible in derived classes.

```
class Example {
  protected:
    int protectedVar;
};
```

4. Defining and Accessing Class Members

We can define member functions **inside** or **outside** the class.

Example 1: Inside Class Definition

```
class Student {
   public:
      string name;

      void display() {
         cout << "Student Name: " << name << endl;
      }
};
```

Example 2: Outside Class Definition

```
class Student {
   public:
      string name;
      void display(); // Function declaration
};

// Function definition outside the class
void Student::display() {
   cout << "Student Name: " << name << endl;
}
```

5. Constructors in C++

A **constructor** is a special function that **initializes objects automatically** when they are created. It has the same name as the class and **no return type**.

Types of Constructors

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Example: Default Constructor

```
class Car {
   public:
      string brand;
      Car() { // Constructor
         cout << "A new car object is created!" << endl;
      }
};

int main() {
    Car car1; // Constructor is called automatically
    return 0;
}
```

Output:

```
A new car object is created!
```

Example: Parameterized Constructor

```
class Car {
    public:
        string brand;
        int year;

        Car(string b, int y) {  // Constructor with parameters
            brand = b;
            year = y;
        }

        void display() {
            cout << "Brand: " << brand << ", Year: " << year << endl;
        }
};

int main() {
    Car car1("Ford", 2023); // Passing arguments
    car1.display();
    return 0;
}
```

6. Destructors in C++

A destructor is a special function that is automatically invoked when an object goes out of scope. It releases resources such as memory. It has the same name as the class but with a tilde (~) symbol**.**

Example: Destructor

```
class Car {
    public:
        Car() {
            cout << "Car object created!" << endl;
        }
        ~Car() {
```

```
        cout << "Car object destroyed!" << endl;
    }
};

int main() {
    Car car1;
    return 0;
}
```

Output:

```
Car object created!
Car object destroyed!
```

7. Objects as Function Arguments

Objects can be passed as **parameters** in functions.

Example: Passing Object to Function

```
class Student {
    public:
        string name;

        void display() {
            cout << "Student Name: " << name << endl;
        }
};

// Function to accept an object as parameter
void showStudent(Student s) {
    s.display();
}

int main() {
    Student s1;
    s1.name = "John";
    showStudent(s1); // Passing object
    return 0;
}
```

8. Array of Objects

We can create an array of objects just like an array of integers.

Example: Storing Multiple Objects in an Array

```
class Car {
   public:
      string brand;
      int year;

      void showDetails() {
         cout << "Brand: " << brand << ", Year: " << year << endl;
      }
};

int main() {
   Car cars[2] = {{"Ford", 2023}, {"BMW", 2022}};

   for (int i = 0; i < 2; i++) {
      cars[i].showDetails();
   }
   return 0;
}
```

9. Pointers to Objects

Pointers can be used to handle objects dynamically.

Example: Pointer to an Object

```
class Car {
   public:
      string brand;
      int year;

      void showDetails() {
         cout << "Brand: " << brand << ", Year: " << year << endl;
      }
};

int main() {
   Car *ptr = new Car;
   ptr->brand = "Audi";
   ptr->year = 2024;
   ptr->showDetails();
```

```
    delete ptr; // Free memory
    return 0;
}
```

- A class is a blueprint for creating objects.
- An object is an instance of a class.
- Access specifiers (public, private, protected) control visibility.
- Constructors initialize objects automatically.
- Destructors free resources when an object is destroyed.
- Objects can be passed to functions and stored in arrays.
- Pointers allow dynamic object management.

This Module provides a detailed guide to Objects and Classes in C++ with examples and syntax, making it easier to understand object-oriented programming concepts.

Unit 6: Member Function

2.2 Member Functions in C++

In C++, a **class** is a user-defined data type that can contain data members (variables) and member functions (methods). **Member functions** are functions that belong to a class and operate on its data members. They provide **encapsulation** by bundling data and behavior together.

Member functions are used to **manipulate the data members**, provide functionality, and enforce data hiding. They are declared inside the class and can be defined **either inside or outside the class**.

Syntax of Member Function

Declaring a Member Function in a Class

```
class ClassName {
public:
    void functionName() {
        // Function body
    }
};
```

Example of a Simple Member Function

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

class Car {
public:
    void display() {
        cout << "This is a car." << endl;
    }
};

int main() {
    Car myCar;
    myCar.display();
    return 0;
}
```

Output:

```
This is a car.
```

Types of Member Functions

Member functions can be classified into the following types:

1. Simple Member Function
2. Inline Member Function
3. Outside Class Definition
4. Static Member Function
5. Constant Member Function
6. Friend Function
7. Virtual Member Function

1. Simple Member Function

A normal member function is declared inside the class and defined inside the class itself.

Example:

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

class Student {
public:
    void showMessage() {
        cout << "Hello, Student!" << endl;
    }
};

int main() {
    Student obj;
    obj.showMessage();
    return 0;
}
```

Output:

```
Hello, Student!
```

2. Inline Member Function

If a function is small, it can be defined directly inside the class using the inline keyword.

Example:

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

```
class Square {
public:
    inline int calculate(int x) {
        return x * x;
    }
};

int main() {
    Square obj;
    cout << "Square of 4 is: " << obj.calculate(4);
    return 0;
}
```

Output:

```
Square of 4 is: 16
```

3. Member Function Defined Outside the Class

Member functions can also be defined **outside the class** using the **scope resolution operator ::**.

Example:

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

class Person {
public:
    void display();  // Function declaration
};

// Function definition outside the class
void Person::display() {
    cout << "Hello from outside the class!" << endl;
}

int main() {
    Person obj;
    obj.display();
    return 0;
}
```

Output:

```
Hello from outside the class!
```

4. Static Member Function

A static member function can be called without creating an object of the class. It can only access **static data members**.

Example:

cpp

CopyEdit

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

class Counter {
private:
    static int count;

public:
    static void showCount() {
        cout << "Count: " << count << endl;
    }
};

int Counter::count = 5;  // Initializing static variable

int main() {
    Counter::showCount();  // Calling static function
    return 0;
}
```

Output:

```
Count: 5
```

5. Constant Member Function

A **constant member function** ensures that the function **does not modify** any data members of the class.

Example:

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

```
class Demo {
public:
    void show() const {
        cout << "This is a constant function." << endl;
    }
};

int main() {
    Demo obj;
    obj.show();
    return 0;
}
```

Output:

```
This is a constant function.
```

6. Friend Function

A **friend function** is not a member of the class but has **access to private and protected members**.

Example:

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

class Box {
private:
    int length;

public:
    Box() { length = 10; }
    friend void showLength(Box b);
};

void showLength(Box b) {
    cout << "Length: " << b.length << endl;
}

int main() {
    Box obj;
    showLength(obj);
    return 0;
}
```

Output:

```
Length: 10
```

7. Virtual Member Function

A **virtual function** is used in **inheritance** to achieve **runtime polymorphism**.

Example:

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

class Base {
public:
    virtual void show() {
        cout << "Base class function" << endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived class function" << endl;
    }
};

int main() {
    Base* basePtr;
    Derived obj;
    basePtr = &obj;
    basePtr->show();
    return 0;
}
```

Output:

```
Derived class function
```

Member functions in C++ enhance encapsulation, data hiding, and modularity**.** They are integral to Object-Oriented Programming (OOP)**.** By understanding different types of member functions such as inline, static, friend, constant, and virtual functions**,** programmers can effectively design efficient and structured C++ programs.

2.3 Array within the Class in C++

In C++, an array within a class is used when we need to store multiple values of the same type as part of an object. Arrays within a class allow storing multiple elements inside an instance of a class, making it useful for handling structured data efficiently.

By defining an array as a data member of a class, we can manipulate the elements using member functions**.**

1. Declaring an Array Inside a Class

We can declare an array as a member variable inside a class. The syntax is similar to normal array declaration, but it is defined inside the class scope.

Syntax:

```
class ClassName {
   private:
      data_type array_name[size];  // Array as a class member

   public:
      void memberFunction();
};
```

Key Points:

- The array can be placed under private or public access specifier.
- The array size should be a **constant** or **fixed at compile time**.
- We use member functions to **initialize** and **access** array elements.

2. Example: Array within a Class

Example 1: Storing and Displaying Student Marks

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

class Student {
private:
   int marks[5];  // Array as a member of class

public:
   void inputMarks() {
        cout << "Enter 5 subject marks: ";
        for(int i = 0; i < 5; i++) {
            cin >> marks[i];  // Taking input for each element
        }
    }

    void displayMarks() {
        cout << "Student Marks: ";
        for(int i = 0; i < 5; i++) {
            cout << marks[i] << " ";  // Displaying array elements
        }
        cout << endl;
    }
};

int main() {
    Student s1;  // Creating an object
    s1.inputMarks();
    s1.displayMarks();

    return 0;
}
```

Output:

```
Enter 5 subject marks: 78 89 92 85 88
Student Marks: 78 89 92 85 88
```

Explanation:

- The class Student has an integer array marks[5] as a **private member**.
- inputMarks() function takes input for 5 subjects.
- displayMarks() function prints the stored values.
- The main() function creates an object s1, calls both member functions, and displays marks.

3. Initializing Arrays in a Class Using a Constructor

We can initialize an array inside a class using a **constructor**.

Example 2: Using Constructor for Initialization

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

```
class Numbers {
private:
    int arr[5];

public:
    Numbers() {  // Constructor to initialize array
        for(int i = 0; i < 5; i++) {
            arr[i] = i * 10;  // Assigning values 0, 10, 20, 30, 40
        }
    }

    void displayArray() {
        cout << "Array Elements: ";
        for(int i = 0; i < 5; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Numbers obj;  // Object created, constructor initializes array
    obj.displayArray();
    return 0;
}
```

Output:

```
Array Elements: 0 10 20 30 40
```

Explanation:

- The **constructor** initializes the array values.
- The displayArray() function prints the array elements.

4. Array as a Public Member in a Class

Arrays can be public members, allowing direct access from objects.

Example 3: Public Array Access

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

class Data {
```

```
public:
    int values[3];  // Public array

    void showValues() {
        cout << "Stored Values: ";
        for(int i = 0; i < 3; i++) {
            cout << values[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Data obj;

    obj.values[0] = 10;
    obj.values[1] = 20;
    obj.values[2] = 30;

    obj.showValues();
    return 0;
}
```

Output:

```
Stored Values: 10 20 30
```

Explanation:

- The array values[3] is **public**, so we can assign values directly.
- The function showValues() prints the array elements.

Note: Public arrays allow direct modification but may **violate encapsulation**.

5. Array of Objects in a Class

Instead of an array as a class member, we can have an **array of objects**.

Example 4: Array of Objects

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

class Employee {
private:
    int id;
    string name;

public:
    void setDetails(int empId, string empName) {
        id = empId;
        name = empName;
    }

    void display() {
        cout << "ID: " << id << ", Name: " << name << endl;
    }
};

int main() {
    Employee employees[3];  // Array of objects

    employees[0].setDetails(101, "Alice");
    employees[1].setDetails(102, "Bob");
    employees[2].setDetails(103, "Charlie");

    cout << "Employee Details: " << endl;
    for(int i = 0; i < 3; i++) {
        employees[i].display();
    }

    return 0;
}
```

Output:

```
Employee Details:
ID: 101, Name: Alice
ID: 102, Name: Bob
ID: 103, Name: Charlie
```

Explanation:

- Employee class has setDetails() and display() functions.
- employees[3] is an **array of objects**, storing multiple employee records.

Notes

6. Dynamic Arrays in a Class

If the array size is unknown at compile-time, we can use **dynamic memory allocation**.

Example 5: Using Dynamic Arrays

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

class DynamicArray {
private:
    int* arr;
    int size;

public:
    DynamicArray(int s) {
        size = s;
        arr = new int[size];  // Dynamically allocating memory
    }

    void inputValues() {
        cout << "Enter " << size << " values: ";
        for(int i = 0; i < size; i++) {
            cin >> arr[i];
        }
    }

    void displayValues() {
        cout << "Stored Values: ";
        for(int i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    ~DynamicArray() {
        delete[] arr;  // Free allocated memory
    }
};
```

```
int main() {
    DynamicArray obj(3);
    obj.inputValues();
    obj.displayValues();

    return 0;
}
```

Output:

```
Enter 3 values: 5 10 15
Stored Values: 5 10 15
```

Conclusion

- Arrays within a class allow storing multiple values inside an object.
- We can use constructors**,** member functions, and dynamic allocation for better management.
- Encapsulation should be maintained by keeping arrays as private members.

2.4 Memory Allocation of Objects in C++

Introduction to Memory Allocation in C++

Memory allocation refers to the process of assigning memory space for variables, objects, and data structures during the execution of a program. In C++, objects can be allocated memory in two ways:

1. **Static Memory Allocation** – Memory is allocated at compile time.
2. **Dynamic Memory Allocation** – Memory is allocated at runtime using new and delete.

Understanding memory allocation is crucial for efficient resource management and avoiding memory leaks.

1. Static Memory Allocation of Objects

In static memory allocation, memory is allocated during compile time, and the allocated memory remains fixed throughout the program execution.

Syntax:

```
class ClassName {
    // Class members
};

int main() {
    ClassName obj;  // Static allocation
}
```

Example:

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

class Student {
    public:
        string name;
        int age;

        void display() {
            cout << "Name: " << name << ", Age: " << age << endl;
        }
};

int main() {
    Student s1;  // Memory allocated statically
    s1.name = "John";
    s1.age = 20;
    s1.display();

    return 0;
}
```

Output:

```
Name: John, Age: 20
```

Key Points:

- Memory is allocated at compile time.
- Objects are created in the **stack memory**.
- Memory is automatically deallocated when the object goes out of scope.

2. Dynamic Memory Allocation of Objects

In dynamic memory allocation, memory is allocated at runtime using the new keyword, and the object is stored in heap memory. The allocated memory must be manually deallocated using delete.

Syntax:

```
ClassName* obj = new ClassName(); // Dynamic allocation
delete obj; // Deallocation
```

Example:

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

class Student {
    public:
        string name;
        int age;

        void display() {
            cout << "Name: " << name << ", Age: " << age << endl;
        }
};

int main() {
    Student* s1 = new Student();  // Memory allocated dynamically
    s1->name = "Alice";
    s1->age = 22;
    s1->display();

    delete s1;  // Deallocating memory

    return 0;
}
```

Output:

```
Name: Alice, Age: 22
```

Key Points:

- Memory is allocated at runtime.
- Objects are stored in **heap memory**.
- We must use delete to free allocated memory and prevent memory leaks.

3. Dynamic Memory Allocation for Arrays of Objects

Sometimes, we need to allocate memory dynamically for an array of objects.

Syntax:

```
ClassName* objArray = new ClassName[size]; // Allocating an array
delete[] objArray; // Deallocating the array
```

Example:

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

class Student {
  public:
    string name;
    int age;

    void display() {
      cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
  int n = 3;
  Student* students = new Student[n];  // Array of objects

  students[0].name = "John";
  students[0].age = 20;

  students[1].name = "Emma";
  students[1].age = 21;

  students[2].name = "Mike";
  students[2].age = 19;

  for (int i = 0; i < n; i++) {
    students[i].display();
  }

  delete[] students;  // Freeing allocated memory

  return 0;
}
```

Output:

```
Name: John, Age: 20
Name: Emma, Age: 21
```

```
Name: Mike, Age: 19
```

Key Points:

- We use new to allocate memory for an array of objects.
- delete[] must be used to deallocate memory for arrays.

4. Constructor and Destructor in Dynamic Memory Allocation

When objects are created dynamically, **constructors** are automatically called, but we must manually call the **destructor** by using delete.

Example:

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

class Student {
   public:
      Student() {
         cout << "Constructor called!" << endl;
      }
      ~Student() {
         cout << "Destructor called!" << endl;
      }
};

int main() {
   Student* s1 = new Student();  // Constructor is called
   delete s1;  // Destructor must be explicitly called using delete

   return 0;
}
```

Output:

```
Constructor called!
Destructor called!
```

Key Points:

- Constructor runs automatically when an object is created.
- Destructor must be invoked manually for dynamically allocated objects using delete.

5. Memory Leak and Its Prevention

What is a Memory Leak?

A **memory leak** occurs when dynamically allocated memory is not deallocated properly, leading to excessive memory usage and performance issues.

Example of Memory Leak:

```
void createObject() {
    int* ptr = new int(10);  // Memory allocated but not deleted
}
```

In this case, ptr is allocated memory but is never deleted, leading to a memory leak.

Preventing Memory Leaks:

Always use delete or delete[] after dynamic memory allocation.

```
void createObject() {
    int* ptr = new int(10);
    delete ptr;  // Properly deallocating memory
}
```

6. Smart Pointers for Automatic Memory Management

C++ provides **smart pointers** (available in the <memory> library) that automatically manage memory, preventing leaks.

Example using unique_ptr:

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

class Student {
    public:
        Student() {
            cout << "Constructor called!" << endl;
        }
        ~Student() {
            cout << "Destructor called!" << endl;
        }
};

int main() {
    unique_ptr<Student> s1 = make_unique<Student>();   // No need
for delete
```

```
    return 0;
}
```

Output:

```
Constructor called!
Destructor called!
```

Key Benefits:

- No need to use delete, as memory is automatically managed.
- Helps prevent memory leaks.

Conclusion

- Static memory allocation is handled automatically by the compiler and uses stack memory**.**
- Dynamic memory allocation uses heap memory and requires manual deallocation using delete.
- Arrays of objects can also be allocated dynamically using new[] and must be freed using delete.
- Memory leaks occur when memory is not properly deallocated, which can be prevented using delete or smart pointers.

By understanding these concepts, programmers can write efficient and optimized C++ programs while effectively managing memory.

This explanation provides a detailed yet structured approach to memory allocation in C++, covering syntax, theory, examples, and best practices.

2.5 Friend Function in C++

Introduction to Friend Function

In C++, data hiding is an important concept in object-oriented programming (OOP). The private and protected members of a class cannot be accessed directly from outside the class. However, sometimes, we need to access these members from non-member functions.

To achieve this, C++ provides Friend Functions, which allow access to private and protected members of a class without being a member of that class.

A friend function is declared inside the class but defined outside the class with the keyword friend.

Syntax of Friend Function

The general syntax of a friend function in C++ is:

```
class ClassName {
private:
    int privateData;

public:
    ClassName() : privateData(0) {}

    // Friend function declaration
    friend void friendFunction(ClassName obj);
};

// Definition of friend function
void friendFunction(ClassName obj) {
    cout << "Private data: " << obj.privateData;
}
```

Key Points in Syntax:

1. The friend function is declared inside the class using the **friend** keyword.
2. The friend function is not a member function of the class but can access private and protected data.
3. The friend function is defined outside the class like a normal function.

Example: Using Friend Function in C++

Example 1: Accessing Private Members Using a Friend Function

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

class Sample {
private:
    int secretNumber;

public:
    Sample(int num) : secretNumber(num) {}

    // Friend function declaration
    friend void showSecret(Sample obj);
};
```

```
// Friend function definition
void showSecret(Sample obj) {
    cout << "The secret number is: " << obj.secretNumber << endl;
}

int main() {
    Sample obj(42);
    showSecret(obj); // Calling friend function
    return 0;
}
```

Output:

```
The secret number is: 42
```

Explanation:

- The class Sample has a private member secretNumber.
- The function showSecret() is declared as a friend.
- Since showSecret() is a friend function, it can access the private data of the Sample class.

Friend Function with Multiple Classes

A friend function can be used to access private members of multiple classes.

Example 2: Friend Function Accessing Two Classes

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

class ClassB;  // Forward declaration

class ClassA {
private:
    int dataA;

public:
    ClassA(int value) : dataA(value) {}

    // Declaring a friend function
    friend void addValues(ClassA objA, ClassB objB);
};
```

```
class ClassB {
private:
    int dataB;

public:
    ClassB(int value) : dataB(value) {}

    // Declaring the same friend function
    friend void addValues(ClassA objA, ClassB objB);
};

// Friend function definition
void addValues(ClassA objA, ClassB objB) {
    cout << "Sum: " << objA.dataA + objB.dataB << endl;
}

int main() {
    ClassA objA(10);
    ClassB objB(20);
    addValues(objA, objB);
    return 0;
}
```

Output:

```
Sum: 30
```

Explanation:

- ClassA and ClassB each have a private variable.
- The addValues() friend function accesses private members of both classes and performs an addition.

Friend Function in Operator Overloading

A friend function is commonly used for operator overloading in C++.

Example 3: Overloading the + Operator Using Friend Function

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

class Number {
private:
    int value;
public:
    Number(int v) : value(v) {}

    // Friend function to overload the '+' operator
    friend Number operator+(Number obj1, Number obj2);

    void display() {
        cout << "Value: " << value << endl;
    }
};

// Friend function definition
Number operator+(Number obj1, Number obj2) {
    return Number(obj1.value + obj2.value);
}

int main() {
    Number n1(5), n2(10);
    Number sum = n1 + n2;
    sum.display();
    return 0;
}
```

Output:

```
Value: 15
```

Explanation:

- The + operator is overloaded using a friend function.
- The friend function accesses private members and returns a new object.

Advantages of Friend Functions

1. **Access to Private Data** – Friend functions can **access private and protected data** of a class.
2. **Useful in Operator Overloading** – Friend functions are widely used for **operator overloading**.
3. **Multiple Class Access** – A single friend function can be used to access **private members of multiple classes**.
4. **Encapsulation Is Maintained** – Even though a friend function accesses private members, it does not belong to the class.

Limitations of Friend Functions

1. **Breaks Data Hiding** – Friend functions break **the principle of encapsulation** because they can access private members.
2. **Increases Coupling** – Since a friend function is not a member of the class, it increases **dependencies** between classes.
3. **Not Inherited** – Friend functions are **not inherited** by derived classes.
4. **Security Issues** – Excessive use of friend functions may expose **sensitive data**.

The friend function in C++ allows accessing private and protected members of a class without being a member of that class. It is declared inside the class using the friend keyword and defined outside like a normal function. Friend functions are **c**ommonly used for operator overloading and accessing multiple classes but should be used carefully to avoid breaking encapsulation**.**

Key Takeaways

- Declared inside a class using friend but **defined outside** the class.
- **Not a member function** but can access **private and protected** members.
- Can be used for **multiple classes** and **operator overloading**.
- Should be used **carefully** to maintain **data security**.

By understanding **friend functions**, programmers can effectively manage **data access** while maintaining **flexibility in object-oriented design**.

2.6 Local Class in C++

In C++, a **local class** is a class that is defined within a function or a block scope. Unlike global or member classes, a local class is accessible only within the function where it is declared. Local classes are useful for **encapsulation** and **hiding implementation details** that are only relevant within a specific function.

Local classes can be used for:

- **Encapsulating helper functionality** within a function.
- **Avoiding namespace pollution**, as they are not accessible outside the function.
- **Enhancing security**, since they are not accessible from other functions.

Syntax of Local Class in C++

A local class is defined inside a function, but its methods can be declared inside or outside the function. The syntax is:

```
void function_name() {
    class LocalClass {  // Local class declaration
    public:
        void display() {
            std::cout << "Inside Local Class" << std::endl;
        }
    };

    LocalClass obj;  // Creating an object of the local class
    obj.display();   // Calling the function
}
```

Key points about local classes:

1. Defined within a function and not accessible outside.
2. Can access only static variables of the enclosing function.
3. Cannot have static data members.
4. Cannot access non-static variables or parameters of the function.
5. Objects of a local class can be created only within the function where it is defined.

Example 1: Basic Local Class Usage

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

void myFunction() {
    class LocalClass {  // Local class inside a function
    public:
        void showMessage() {
            cout << "This is a local class function!" << endl;
        }
    };

    LocalClass obj;  // Creating an object
    obj.showMessage();  // Calling the function
}
```

```
int main() {
    myFunction();  // Call function that contains local class
    return 0;
}
```

Output:

```
This is a local class function!
```

Accessing Static Variables of Enclosing Function

Since local classes cannot access non-static variables of the enclosing function, they can only use **static variables**.

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

void myFunction() {
    static int count = 0;  // Static variable

    class LocalClass {
    public:
        void increment() {
            count++;  // Accessing static variable
            cout << "Count: " << count << endl;
        }
    };

    LocalClass obj1, obj2;
    obj1.increment();
    obj2.increment();
}

int main() {
    myFunction();
    myFunction();  // Calling again to show static behavior
    return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
```

Limitations of Local Class

1. **Cannot Access Non-Static Variables**
    - Local classes **cannot directly access** the non-static variables of the enclosing function.

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

void myFunction() {
    int x = 10;  // Non-static variable

    class LocalClass {
    public:
        void display() {
            // cout << "Value of x: " << x;  // Error: Cannot access non-
static variables
        }
    };

    LocalClass obj;
    obj.display();
}

int main() {
    myFunction();
    return 0;
}
```

2. Cannot Have Static Data Members
    - Unlike normal classes, local classes cannot have static data members**.**

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

void myFunction() {
    class LocalClass {
    public:
        static int x;  // Error: Static data members not allowed
    };
}
```

```
int main() {
   myFunction();
   return 0;
}
```

Compiler Error:

Error: Static data members are not allowed in local classes

3. **Cannot Use Friend Functions or Templates**
    - Local classes cannot have friend functions**.**
    - They cannot be used as template arguments directly.

Example 2: Using Local Class with Function Parameters

A local class can work with parameters passed to a function**, but it** cannot directly access them unless they are passed to the local class as arguments.

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

void calculateSquare(int num) {
   class LocalClass {
   public:
      int square(int x) {
         return x * x;
      }
   };

   LocalClass obj;
   cout << "Square of " << num << " is: " << obj.square(num) <<
endl;
}

int main() {
   calculateSquare(5);
   calculateSquare(7);
   return 0;
}
```

Output:

```
Square of 5 is: 25
Square of 7 is: 49
```

Example 3: Using Local Class with Pointers

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

void pointerExample() {
   class LocalClass {
   public:
      void printMessage(const char* message) {
         cout << "Message: " << message << endl;
      }
   };

   LocalClass obj;
   obj.printMessage("Hello from Local Class!");
}

int main() {
   pointerExample();
   return 0;
}
```

Output:

```
Message: Hello from Local Class!
```

Advantages of Local Class

1. **Encapsulation:**
   - Hides the class implementation inside the function.
2. **Memory Efficiency:**
   - Objects of local classes exist only while the function executes, saving memory.
3. **Better Readability & Maintenance:**
   - Keeps related logic in one place, reducing global scope pollution.

Local classes in C++ provide a powerful way to encapsulate logic within a function, ensuring that certain classes remain hidden from the rest of the program. However, they come with limitations, such as the inability to have static data members or access non-static variables of the enclosing function.

2.7 Constructors in C++

A **constructor** is a special member function in C++ that initializes objects of a class. It has the same name as the class and is automatically called when an object is created.

Key Features of Constructors:

- They **do not return any value** (not even void).
- They are **invoked automatically** when an object is created.
- They **initialize** the object's data members.
- They can be **overloaded** to handle different types of initialization.

Types of Constructors in C++

1. **Parameterized Constructor**
2. **Multiple Constructors (Constructor Overloading)**
3. **Default Argument Constructor**

1. Parameterized Constructor

A **parameterized constructor** is used to initialize an object with specific values at the time of creation. It takes arguments and assigns them to object data members.

Syntax:

```
class ClassName {
public:
    ClassName(data_type param1, data_type param2) {
        // Constructor body
    }
};
```

Example:

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

class Student {
private:
    string name;
    int age;

public:
    // Parameterized Constructor
    Student(string studentName, int studentAge) {
        name = studentName;
        age = studentAge;
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Student s1("John", 20);  // Passing values at object creation
    Student s2("Alice", 22);

    s1.display();
    s2.display();

    return 0;
}
```

Output:

```
Name: John, Age: 20
Name: Alice, Age: 22
```

In this example:

- The **constructor Student(string, int)** initializes objects with values.
- When s1 and s2 are created, their data members are assigned values.

2. Multiple Constructors (Constructor Overloading)

C++ allows multiple constructors with **different parameters** in the same class. This is called **constructor overloading**.

Syntax:

```
class ClassName {
public:
    ClassName() { ... }            // Default Constructor
    ClassName(int x) { ... }       // Parameterized Constructor
    ClassName(int x, int y) { ... } // Another Parameterized Constructor
};
```

Example:

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

class Rectangle {
private:
    int length, width;

public:
    // Default Constructor
    Rectangle() {
        length = 0;
        width = 0;
    }

    // Constructor with one parameter
    Rectangle(int side) {
        length = width = side;  // Square
    }

    // Constructor with two parameters
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }

    void display() {
        cout << "Length: " << length << ", Width: " << width << endl;
    }
};

int main() {
    Rectangle r1;        // Calls Default Constructor
    Rectangle r2(5);     // Calls Constructor with one parameter
    Rectangle r3(4, 6);  // Calls Constructor with two parameters

    r1.display();
    r2.display();
    r3.display();

    return 0;
}
```

Output:

```
Length: 0, Width: 0
Length: 5, Width: 5
Length: 4, Width: 6
```

Here, the constructor is **overloaded** to accept **zero, one, or two parameters**, allowing different ways to create objects.

3. Default Argument Constructor

A default argument constructor allows assigning default values to parameters. If no arguments are provided, the default values are used.

Syntax:

```
class ClassName {
public:
    ClassName(data_type param1 = default_value1, data_type param2
= default_value2) {
        // Constructor body
    }
};
```

Example:

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

class Car {
private:
    string brand;
    int price;

public:
    // Default Argument Constructor
    Car(string carBrand = "Toyota", int carPrice = 500000) {
        brand = carBrand;
        price = carPrice;
    }

    void display() {
        cout << "Brand: " << brand << ", Price: " << price << endl;
    }
};

int main() {
    Car c1;            // Uses default values
    Car c2("Honda");   // Uses default price
    Car c3("BMW", 1200000); // Uses provided values

    c1.display();
    c2.display();
    c3.display();

    return 0;
}
```

Output:

```
Brand: Toyota, Price: 500000
Brand: Honda, Price: 500000
Brand: BMW, Price: 1200000
```

In this example:

- If no values are passed, **default values** ("Toyota", 500000) are used.
- If one argument is passed ("Honda"), the default price is used.
- If both arguments are passed ("BMW", 1200000), they override the defaults.

Table 2.1 Comparison of Constructor Types

Constructor Type	Definition	Usage Example
Parameterized Constructor	Initializes an object with specific values passed as arguments.	Student s1("John", 20);
Multiple Constructors (Constructor Overloading)	Different constructors handle different ways of initializing an object.	Rectangle r1(); or Rectangle r2(5,10);

Default Argument Constructor	Allows setting default values for parameters if no arguments are provided.	Car c1();, Car c2("Honda");

Conclusion

- Constructors help in automatic **object initialization** when an instance is created.
- **Parameterized constructors** allow passing values.
- **Multiple constructors** provide flexibility using **constructor overloading**.
- **Default argument constructors** allow setting **default values** while still allowing customization.

2.8 Dynamic Initialization of Objects, Copy Constructor, and Dynamic Constructor in C++

1. Dynamic Initialization of Objects

Dynamic initialization refers to initializing objects at runtime using values provided by the user or obtained during program execution. This is particularly useful when the values needed for initialization are not known at compile time.

C++ supports **dynamic memory allocation** using the new operator, allowing objects to be created in the **heap memory**. This is useful for efficient memory management, especially when working with **variable-sized data**.

Syntax

```
class ClassName {
   data_type variable;
public:
   ClassName(data_type value) {
       variable = value;  // Dynamic initialization
   }
};
```

Example: Dynamic Initialization of an Object

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

class Rectangle {
    int length, width;
public:
    // Constructor with dynamic initialization
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }

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

int main() {
    int l, w;

    cout << "Enter length and width: ";
    cin >> l >> w;

    Rectangle r(l, w);  // Dynamic Initialization
    cout << "Area of Rectangle: " << r.area() << endl;

    return 0;
}
```

Output:

```
Enter length and width: 10 5
Area of Rectangle: 50
```

Key Points:

- Object values are initialized at runtime using user input.
- Useful when object attributes depend on dynamic conditions.
- Helps in optimizing memory usage.

2. Copy Constructor

Theory

A **copy constructor** is a special constructor used to **initialize an object using another object of the same class**. It creates a new object by **copying the values** from an existing object.

By default, C++ provides a **default copy constructor** that performs **shallow copying**. However, in cases where dynamic memory allocation is used, we must define a **custom copy constructor** to avoid memory issues like **dangling pointers and duplicate memory deallocation**.

Syntax

```
class ClassName {
public:
    ClassName(const ClassName &obj) {
        // Copy constructor definition
    }
};
```

Example: Copy Constructor Demonstration

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

class Student {
    string name;
    int age;
public:
    // Parameterized Constructor
    Student(string n, int a) {
        name = n;
        age = a;
    }

    // Copy Constructor
    Student(const Student &s) {
        name = s.name;
        age = s.age;
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Student s1("Alice", 21); // Normal Constructor
    Student s2 = s1;         // Copy Constructor
    cout << "Original Object: ";
    s1.display();

    cout << "Copied Object: ";
    s2.display();

    return 0;
}
```

Output:

```
Original Object: Name: Alice, Age: 21
Copied Object: Name: Alice, Age: 21
```

Key Points:

- The copy constructor is called when a **new object is initialized from an existing object**.
- If not defined explicitly, the compiler provides a **default copy constructor**.
- Required when objects use **dynamic memory allocation**, preventing **shallow copying issues**.

3. Dynamic Constructor

Theory

A **dynamic constructor** is a constructor that **allocates memory dynamically** using the new operator. This is particularly useful when dealing with **variable-sized arrays, strings, or objects with memory allocated at runtime**.

Since memory is allocated dynamically, it must be **released manually** using the delete operator inside the destructor to prevent **memory leaks**.

Syntax

```
class ClassName {
    data_type* ptr;
public:
    ClassName(size_t size) {
        ptr = new data_type[size];  // Dynamic memory allocation
    }

    ~ClassName() {
        delete[] ptr;  // Releasing allocated memory
    }
};
```

Example: Dynamic Constructor in Action

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

class DynamicArray {
    int *arr;
    int size;
public:
    // Dynamic Constructor
    DynamicArray(int s) {
        size = s;
        arr = new int[size];  // Allocating memory dynamically
        for (int i = 0; i < size; i++) {
            arr[i] = i * 10;  // Assigning values dynamically
        }
    }

    void display() {
        for (int i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    // Destructor to free memory
    ~DynamicArray() {
        delete[] arr;
    }
};

int main() {
    int n;

    cout << "Enter size of array: ";
    cin >> n;
    DynamicArray dArr(n);  // Creating an object dynamically
    cout << "Array elements: ";
    dArr.display();

    return 0;
}
```

Output:

```
Enter size of array: 5
Array elements: 0 10 20 30 40
```

Key Points:

- A **dynamic constructor** allocates memory at runtime using new.
- It is useful for **dynamic data structures like linked lists, arrays, and trees**.
- The **destructor** must release memory using delete [] to **prevent memory leaks**

Table 2.2 Concepts of Constructor Types

Concept	**Description**	**Key Feature**
Dynamic Initialization	Assigns values to object attributes at runtime.	Uses parameterized constructors**.**
Copy Constructor	Initializes a new object using an existing object.	Avoids shallow copy issues when using dynamic memory allocation**.**
Dynamic Constructor	Allocates memory dynamically using new.	Must use delete in the destructor to free memory.

When to Use?

- **Dynamic Initialization:** When values for object properties are not known at compile time.
- **Copy Constructor:** When we need to create a **duplicate object** while ensuring deep copying.
- **Dynamic Constructor:** When working with **dynamic memory allocation**, such as **arrays, linked lists, or large data structures**.

By understanding and implementing these concepts, programmers can manage **object-oriented memory allocation efficiently** in C++.

2.9 Destructors in C++

In object-oriented programming, constructors and destructors play a crucial role in managing the lifecycle of an object. While a **constructor** is used to initialize an object, a **destructor** is used to clean up resources before an object is destroyed.

A **destructor** is a special member function in C++ that is automatically called when an object goes out of scope or is explicitly deleted. It is primarily used to release memory, close files, or perform cleanup operations.

1. Destructor Syntax

The destructor in C++:

- Has the **same name** as the class, but prefixed with a tilde ~.
- **Takes no parameters** and has **no return type** (not even void).
- Is **automatically invoked** when an object is destroyed.

General Syntax:

```
class ClassName {
public:
   ~ClassName() {
      // Destructor body
   }
};
```

2. Basic Example of a Destructor

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

class Demo {
public:
   // Constructor
   Demo() {
      cout << "Constructor is called!" << endl;
   }

   // Destructor
   ~Demo() {
      cout << "Destructor is called!" << endl;
   }
};
```

```
int main() {
    Demo obj; // Object created
    return 0;
}
```

Output:

```
Constructor is called!
Destructor is called!
```

Explanation:

- When obj is created, the constructor executes.
- As soon as the program reaches the end of main(), the destructor is automatically invoked, destroying obj.

3. Destructor in Dynamic Memory Allocation

Destructors are crucial when dynamically allocating memory to prevent **memory leaks**.

Example: Using Destructor to Release Heap Memory

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

class DynamicArray {
private:
    int* arr;
    int size;

public:
    // Constructor - Allocates memory
    DynamicArray(int s) {
        size = s;
        arr = new int[size];
        cout << "Memory allocated for array of size " << size << endl;
    }

    // Destructor - Deallocates memory
    ~DynamicArray() {
        delete[] arr;
        cout << "Memory deallocated" << endl;
    }
};
```

```
int main() {
    DynamicArray obj(5);
    return 0;
}
```

Output:

```
Memory allocated for array of size 5
Memory deallocated
```

Explanation:

- The constructor dynamically allocates memory using new.
- The destructor releases the allocated memory using delete[], preventing memory leaks.

4. When is a Destructor Called?

A destructor is automatically called in the following cases:

1. **When a local object goes out of scope** (at the end of a block).
2. **When a dynamically allocated object is explicitly deleted** using delete.
3. **For static objects at program termination**.
4. **For objects inside another object**, when the containing object is destroyed.

5. Destructor in Inheritance (Base & Derived Class)

In an **inheritance hierarchy**, destructors are called in **reverse order**—first the derived class destructor, then the base class destructor.

Example: Destructor in Inheritance

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

class Base {
public:
    Base() { cout << "Base Constructor\n"; }
    ~Base() { cout << "Base Destructor\n"; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor\n"; }
    ~Derived() { cout << "Derived Destructor\n"; }
};

int main() {
    Derived obj;
    return 0;
}
```

Output:

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

Explanation:

- The **Base class constructor** runs first, followed by the **Derived class constructor**.
- On destruction, the **Derived class destructor** runs first, followed by the **Base class destructor**.

6. Destructor in Polymorphism (Virtual Destructor)

If a base class has a **non-virtual destructor**, deleting a derived class object using a base class pointer causes **undefined behavior**.

Wrong Way (Without Virtual Destructor):

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

class Base {
public:
    ~Base() { cout << "Base Destructor\n"; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "Derived Destructor\n"; }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Only Base Destructor is called!
    return 0;
}
```

Output:

```
Base Destructor
```

The **Derived class destructor is never called!**, leading to a memory leak.

Correct Way (Using Virtual Destructor):

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

class Base {
public:
    virtual ~Base() { cout << "Base Destructor\n"; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "Derived Destructor\n"; }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Both destructors are called correctly
    return 0;
}
```

Output:

```
Derived Destructor
Base Destructor
```

By declaring the destructor in the base class as **virtual**, C++ ensures proper destructor chaining, avoiding memory leaks.

7. Destructor and Smart Pointers

C++11 introduced **smart pointers** to automate memory management.

Example: Using unique_ptr

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

class Demo {
public:
    Demo() { cout << "Constructor\n"; }
```

```
   ~Demo() { cout << "Destructor\n"; }
};

int main() {
   unique_ptr<Demo> ptr = make_unique<Demo>();
   return 0;
}
```

Output:

```
Constructor
Destructor
```

Since unique_ptr automatically calls the destructor, **no need for explicit delete**.

8. Key Points About Destructors

1. **Only one destructor per class** (cannot be overloaded).
2. **Cannot be declared const, volatile, or static**.
3. **Should release resources** (memory, files, database connections).
4. **Destructor execution order is reverse** of constructor execution.
5. **Use virtual destructors in base classes** when working with inheritance.
6. **Use smart pointers (unique_ptr, shared_ptr)** to avoid manual memory management.

Destructors in C++ ensure proper resource management by **automatically deallocating memory and releasing resources** when an object is destroyed. Understanding destructors is essential for writing efficient and memory-safe programs, especially when working with **dynamic memory allocation, inheritance, and polymorphism**. By following best practices such as **using virtual destructors in base classes and leveraging smart pointers**, developers can prevent memory leaks and undefined behavior, leading to more robust and maintainable C++ applications.

MCQs:

1. What is a class in C++?

A. A function that performs a specific task

B. A collection of variables

C. A blueprint for creating objects

D. A type of loop

2. Which of the following is the correct way to create an object of a class named Car?

A. Car();

B. object Car;

C. Car car1;

D. create Car;

3. What is a constructor in C++?

A. A function used to destroy an object

B. A special function used to initialize objects

C. A loop that repeats object creation

D. A static method

4. How many constructors can a class have in C++?

A. Only one

B. Only two

C. As many as needed (function overloading applies)

D. None

5. Which of the following constructor types does NOT take any parameters?

A. Parameterized constructor

B. Copy constructor

C. Default constructor

D. Virtual constructor

6. What is the purpose of a destructor in C++?

A. To create new objects

B. To copy one object to another

C. To initialize member variables

D. To release resources when an object is destroyed

7. What is the symbol used to define a destructor in C++?

A. +

B. *

C. ~

D. !

8. Which of the following statements about constructors is TRUE?

A. Constructors must have a return type

B. Constructors can be virtual

C. Constructors can be overloaded

D. Constructors cannot be defined inside the class

9. What happens if you do not define a constructor in your class?

A. The program will not compile

B. An error will be thrown

C. The compiler provides a default constructor

D. The object cannot be created

10. Which constructor is called when an object is initialized with another object of the same class?

A. Default constructor

B. Destructor

C. Copy constructor

D. Static constructor

Short Questions:

1. What is a class in C++?
2. Define an object in the context of C++ OOP.
3. How do you declare and create an object of a class in C++?
4. What is the main purpose of a constructor in C++?
5. What is a default constructor?
6. Can constructors be overloaded in C++? If yes, how?
7. What is a parameterized constructor?
8. What is a copy constructor? When is it invoked?
9. What is the syntax for defining a destructor in C++?
10. What is the role of a destructor in a class?
11. Can a class have more than one destructor in C++? Why or why not?
12. What happens if you don’t define a constructor or destructor in your class?

Long Questions:

1. Define a class and an object in C++. How do they relate to each other in the object-oriented paradigm? Provide an example.
2. Explain how to declare and define a class in C++. Then show how to create and use an object of that class.
3. What is a constructor in C++? Describe its characteristics, rules, and how it differs from a regular member function.
4. Write an algorithm that demonstrates the use of a default constructor. Explain how it is automatically invoked.
5. What is a parameterized constructor? How is it useful in initializing class members with specific values? Write a C++ example to support your explanation.
6. Describe the concept of constructor overloading in C++. Why is it important? Provide a code example with at least two different constructors.
7. What is a copy constructor in C++? When is it called? Write a program to demonstrate its use and explain its behavior.
8. Define a destructor. Explain its purpose in C++ and how it differs from a constructor. Provide an example where a destructor is useful.
9. Can constructors or destructors be overloaded or inherited in C++? Justify your answer with reasons and examples.
10. Explain how memory management is handled using constructors and destructors in C++. Why are they crucial in resource handling?
11. Write a complete C++ program that includes a class with all types of constructors (default, parameterized, and copy) and a destructor. Explain how each of them works during program execution.
12. Discuss the lifecycle of an object in C++ from creation to destruction. How do constructors and destructors play a role in this lifecycle? Illustrate with a practical example.

MODULE 3
OPERATOR OVERLOADING AND INHERITANCE

3.0 LEARNING OUTCOMES

By the end of this Module, students will be able to:

- Understand operator overloading (unary & binary) and its rules.
- Implement binary operator overloading using friend functions.
- Learn type conversion in C++.
- Explore inheritance and its role in derived classes.
- Implement single, multilevel, multiple, hierarchical, and hybrid inheritance.
- Understand virtual base classes and abstract classes.
- Explain constructors in derived classes and their execution sequence.
- Learn about member classes and their significance.

Unit 7: Operator Overloading: Unary and Binary

3.1 Operator Overloading in C++

Operator overloading is a feature in C++ that allows **redefining the behavior of operators** when applied to user-defined data types (objects). This enables objects to be manipulated in an intuitive manner, just like primitive data types.

For example, using + to add two objects of a class makes the code more readable and natural.

Syntax of Operator Overloading

The syntax for operator overloading is:

```
return_type operator symbol (parameters) {
    // Function body defining the operation
}
```

- **operator** is the keyword used for overloading.
- **symbol** is the operator being overloaded (+, -, *, etc.).
- The function can be defined inside the class or as a friend function.

Unary Operator Overloading

Unary operators operate on **a single operand**. Examples include ++, --, -, and !.

Overloading Unary Operators

- When overloading a unary operator, no arguments are passed.
- The overloaded function must be a member function.

Example: Overloading the ++ Operator (Prefix & Postfix)

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

class Counter {
    int value;
public:
    Counter() { value = 0; }

    void display() {
        cout << "Value: " << value << endl;
    }

    // Overloading Prefix ++
    void operator++() {
        ++value;
    }

    // Overloading Postfix ++
    void operator++(int) {
        value++;
    }
};

int main() {
    Counter c1;

    cout << "Initial ";
    c1.display();

    ++c1; // Calls prefix operator++
    cout << "After Prefix Increment ";
    c1.display();

    c1++; // Calls postfix operator++
    cout << "After Postfix Increment ";
    c1.display();

    return 0;
}
```

Explanation

- operator++() handles **prefix increment** (++c1).
- operator++(int) handles **postfix increment** (c1++).
- No arguments are passed for prefix overload.
- The postfix version takes an **int dummy parameter** to differentiate it from the prefix.

Output

```
Initial Value: 0
After Prefix Increment Value: 1
After Postfix Increment Value: 2
```

Binary Operator Overloading

Binary operators operate on **two operands**. Examples include +, -, *, /, ==, etc.

Overloading Binary Operators

- Binary operators require two operands, so the function typically takes one argument.
- It can be defined as a **member function** or a **friend function**.

Example: Overloading the + Operator

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

class Complex {
   int real, imag;
public:
   Complex(int r = 0, int i = 0) {
      real = r;
      imag = i;
   }

   // Overloading the + operator
   Complex operator+(Complex obj) {
      Complex temp;
      temp.real = real + obj.real;
      temp.imag = imag + obj.imag;
      return temp;
   }

   void display() {
      cout << real << " + " << imag << "i" << endl;
   }
};

int main() {
   Complex c1(3, 4), c2(1, 2);

   Complex c3 = c1 + c2;  // Calls overloaded + operator
   c3.display();
```

```
    return 0;
}
```

Explanation

- The operator+ function takes an object as a parameter.
- It **adds the real and imaginary parts** separately.
- The function returns the result as a new object.

Output

```
4 + 6i
```

Unit 8: Overloading Binary Operators Using Friends

3.2 Binary Operator Overloading Using Friend Function

A **friend function** can also be used for operator overloading when **two different objects** need to be operated on.

Example: Overloading * Using a Friend Function

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

class Multiply {
   int value;
public:
   Multiply(int v) { value = v; }

   // Friend function to overload *
   friend Multiply operator*(Multiply obj1, Multiply obj2);

   void display() {
      cout << "Result: " << value << endl;
   }
};

// Definition of the friend function
Multiply operator*(Multiply obj1, Multiply obj2) {
   return Multiply(obj1.value * obj2.value);
}

int main() {
   Multiply m1(4), m2(5);

   Multiply m3 = m1 * m2;  // Calls overloaded * operator
   m3.display();

   return 0;
}
```

Explanation

- The operator* function is a **friend function**.
- It allows access to private data of objects.

- The function **multiplies two objects** and returns the result.

Output

```
Result: 20
```

Overloading Comparison Operators (==, !=, >, <)

Comparison operators (==, !=, >, <) can also be overloaded to compare objects.

Example: Overloading == Operator

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

class Compare {
   int num;
public:
   Compare(int n) { num = n; }

   bool operator==(Compare obj) {
      return num == obj.num;
   }
};

int main() {
   Compare c1(10), c2(10), c3(20);

   if (c1 == c2)
      cout << "c1 and c2 are equal" << endl;
   else
      cout << "c1 and c2 are not equal" << endl;

   if (c1 == c3)
      cout << "c1 and c3 are equal" << endl;
   else
      cout << "c1 and c3 are not equal" << endl;

   return 0;
}
```

Output

```
c1 and c2 are equal
```

c1 and c3 are not equal

Key Points

✔ Operator overloading allows intuitive operations on objects.
✔ Unary operators (++, --) are overloaded as member functions.
✔ Binary operators (+, -, *, /) take one parameter.
✔ Friend functions are useful when working with two objects.
✔ Comparison operators (==, !=) can be overloaded for object comparison.

Using operator overloading, we can make custom classes work just like built-in types, making **code more readable, efficient, and natural**.

Unit 9: Rules of Overloading Operators, Type Conversion

3.3 Operator Overloading and Type Conversion in C++

Operator overloading is a powerful feature in C++ that allows operators to be redefined and used with user-defined data types. Similarly, type conversion enables converting one data type into another, either implicitly or explicitly. This Module covers the **rules of operator overloading and type conversion** with theory, syntax, and examples.

1. Rules of Overloading Operators

Operator overloading allows the same operator to work with user-defined types (such as objects of a class) while maintaining its original functionality with built-in types.

Syntax of Operator Overloading

The syntax for operator overloading is as follows:

```
return_type operator symbol (parameters) {
    // Function body defining the behavior of the operator
}
```

Example of Operator Overloading

cpp

CopyEdit

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

class Complex {
public:
    int real, imag;

    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }

    // Overloading + operator
    Complex operator+(Complex const& obj) {
        Complex result;
        result.real = real + obj.real;
        result.imag = imag + obj.imag;
        return result;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2; // Uses overloaded +
    c3.display();
    return 0;
}
```

Output:

```
4 + 6i
```

Rules for Operator Overloading

1. **Only Existing Operators Can Be Overloaded:**
   - C++ does not allow defining new operators.
   - Example: @ cannot be overloaded because it is not a predefined C++ operator.
2. **At Least One Operand Must Be a User-Defined Type (Class or Struct):**
   - Example: Overloading + for adding two class objects.
3. **Some Operators Cannot Be Overloaded:**
   - Operators that **cannot** be overloaded include:
     - :: (Scope resolution operator)
     - .* (Pointer-to-member operator)
     - . (Member access operator)
     - sizeof (Size operator)
4. **Overloaded Operators Follow Default Precedence and Associativity:**
   - Even if overloaded, operators follow the standard C++ precedence rules.
5. **Overloaded Operators Must Be Either Member or Friend Functions:**

- If the left operand is a built-in type, use a friend function.

6. **Unary and Binary Operators Overloading:**
   - **Unary operators** (e.g., ++, --) take no arguments.
   - **Binary operators** (e.g., +, -) take one argument if implemented as a member function and two if implemented as a friend function.

Example: Overloading Unary Operator (++)

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

class Counter {
public:
    int value;

    Counter() { value = 0; }

    // Overloading prefix ++
    void operator++() {
        ++value;
    }

    void display() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Counter c;
    ++c; // Uses overloaded ++
    c.display();
    return 0;
}
```

Output:

```
Value: 1
```

2. Type Conversion in C++

Type conversion refers to changing a value from one data type to another. It can be:

1. **Implicit Type Conversion (Type Promotion)**
2. **Explicit Type Conversion (Type Casting)**
3. **User-Defined Type Conversion**
   - Conversion from basic type to class type
   - Conversion from class type to basic type
   - Conversion from one class type to another class type

1. Implicit Type Conversion (Automatic Type Promotion)

C++ automatically converts a smaller data type to a larger data type when needed.

Example:

```
int a = 5;
float b = a; // Implicit conversion from int to float
```

2. Explicit Type Conversion (Type Casting)

The user manually converts one data type into another using type casting.

Syntax:

```
(data_type) value;
```

Example:

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

int main() {
   double num = 10.5;
   int intNum = (int)num; // Explicit conversion from double to int
   cout << "Converted value: " << intNum << endl;
   return 0;
}
```

Output:

```
Converted value: 10
```

3. User-Defined Type Conversion

A. Basic Type to Class Type

Converting primitive data types to class objects.

Example:

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

```
class Distance {
    int meters;
public:
    Distance(int m) { meters = m; } // Constructor handles conversion
    void display() { cout << "Meters: " << meters << endl; }
};

int main() {
    Distance d = 10; // Converts int to Distance object
    d.display();
    return 0;
}
```

Output:

```
Meters: 10
```

B. Class Type to Basic Type

Converting an object of a class to a primitive data type.

Example:

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

class Distance {
    int meters;
public:
    Distance(int m) { meters = m; }
    operator int() { return meters; } // Conversion function
};

int main() {
    Distance d(10);
    int meters = d; // Converts Distance object to int
    cout << "Meters: " << meters << endl;
    return 0;
}
```

Output:

```
Meters: 10
```

C. Class Type to Another Class Type

Example:

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

class Fahrenheit {
   float temp;
public:
   Fahrenheit(float t) { temp = t; }
   float getTemp() { return temp; }
};

class Celsius {
   float temp;
public:
   Celsius(float t) { temp = t; }
   // Conversion constructor
   Celsius(Fahrenheit f) {
      temp = (f.getTemp() - 32) * 5 / 9;
   }
   void display() { cout << "Temperature in Celsius: " << temp <<
endl; }
};

int main() {
   Fahrenheit f(98.6);
   Celsius c = f; // Converts Fahrenheit to Celsius
   c.display();
   return 0;
}
```

Output:

```
Temperature in Celsius: 37
```

- **Operator overloading** allows defining custom behavior for operators with user-defined types.
- **Type conversion** enables converting values between data types, either implicitly, explicitly, or via user-defined conversions.

- Following **operator overloading rules** ensures correct implementation without violating C++ constraints.
- **User-defined type conversions** help in seamless data transformations between primitive and object types.

This completes the **detailed study of operator overloading and type conversion in C++**. 🚀

3.4 Inheritance and Derived Classes in C++

Inheritance is one of the most important concepts in Object-Oriented Programming (OOP). It allows a new class (called the **derived class**) to inherit attributes and methods from an existing class (called the **base class**). This promotes **code reusability** and improves **maintainability**.

Key Advantages of Inheritance:

- Reduces code duplication.
- Promotes code reusability.
- Helps in achieving hierarchical classification.
- Enhances code readability and structure.

1. Syntax of Inheritance in C++

Basic Syntax:

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

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

Here, the **access_specifier** determines how the base class members are inherited.

Table 3.1 Types of Access Specifiers:

Access Specifier	**Private Members**	**Protected Members**	**Public Members**
private	Not inherited	Inherited as private	Inherited as private
protected	Not inherited	Inherited as protected	Inherited as protected
public	Not inherited	Inherited as protected	Inherited as public

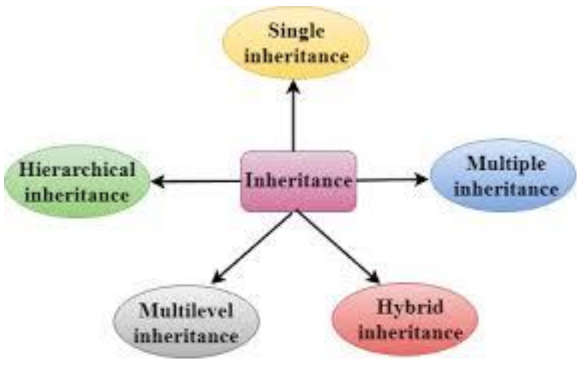


Figure 2 Types of Inheritance
[Source: https://medium.com]

3.5 Inheritance in C++

Inheritance is a fundamental concept in **Object-Oriented Programming (OOP)** that allows a class to derive properties and behaviors from another class. The class that is **inherited** is called the **base class (parent class)**, and the class that **inherits** is called the **derived class (child class)**.

Advantages of Inheritance

- **Code reusability:** Common functionalities can be reused in different classes.
- **Extensibility:** Enhances the maintainability of the code.
- **Improved readability:** Reduces code duplication.

Syntax for Inheritance in C++

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

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

Here, access_specifier can be:

- public: Public and protected members of the base class remain the same in the derived class.
- protected: Public and protected members of the base class become protected in the derived class.
- private: Public and protected members of the base class become private in the derived class.

1. Single Inheritance

In **single inheritance**, a derived class inherits from a single base class.

Syntax:

```
class Parent {
public:
   void show() {
      cout << "This is the parent class." << endl;
   }
};

class Child : public Parent {
public:
   void display() {
      cout << "This is the child class." << endl;
   }
};
```

Example:

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

class Parent {
public:
   void show() {
      cout << "This is the parent class." << endl;
   }
};

class Child : public Parent {
public:
   void display() {
      cout << "This is the child class." << endl;
   }
};

int main() {
   Child obj;
   obj.show();   // Accessing parent class function
   obj.display(); // Accessing child class function
   return 0;
}
```

Output:

```
This is the parent class.
This is the child class.
```

2. Multilevel Inheritance

In **multilevel inheritance**, a class is derived from another derived class, forming a chain.

Syntax:

```
class Grandparent {
  // Base class
};

class Parent : public Grandparent {
  // Derived class
};

class Child : public Parent {
  // Further derived class
};
```

Example:

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

class Grandparent {
public:
    void display1() {
        cout << "This is the grandparent class." << endl;
    }
};

class Parent : public Grandparent {
public:
    void display2() {
        cout << "This is the parent class." << endl;
    }
};

class Child : public Parent {
public:
    void display3() {
        cout << "This is the child class." << endl;
    }
};

int main() {
    Child obj;
    obj.display1();
    obj.display2();
    obj.display3();
    return 0;
}
```

Output:

```
This is the grandparent class.
This is the parent class.
This is the child class.
```

3. Multiple Inheritance

In **multiple inheritance**, a class inherits from two or more base classes.

Syntax:

```
class Parent1 {
   // Base class 1
};

class Parent2 {
   // Base class 2
};

class Child : public Parent1, public Parent2 {
   // Derived class
};
```

Example:

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

class Parent1 {
public:
    void show1() {
        cout << "This is the first parent class." << endl;
    }
};

class Parent2 {
public:
    void show2() {
        cout << "This is the second parent class." << endl;
    }
};

class Child : public Parent1, public Parent2 {
public:
    void display() {
        cout << "This is the child class." << endl;
    }
};

int main() {
    Child obj;
    obj.show1();
    obj.show2();
    obj.display();
    return 0;
}
```

Output:

```
This is the first parent class.
This is the second parent class.
This is the child class.
```

4. Hierarchical Inheritance

In **hierarchical inheritance**, multiple classes inherit from a single base class.

Syntax:

```
class Parent {
   // Base class
};

class Child1 : public Parent {
  // Derived class 1
};

class Child2 : public Parent {
  // Derived class 2
};
```

Example:

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

class Parent {
public:
    void display() {
        cout << "This is the parent class." << endl;
    }
};

class Child1 : public Parent {
public:
    void show1() {
        cout << "This is the first child class." << endl;
    }
};

class Child2 : public Parent {
public:
    void show2() {
        cout << "This is the second child class." << endl;
    }
};

int main() {
    Child1 obj1;
    Child2 obj2;

    obj1.display();
    obj1.show1();
    obj2.display();
    obj2.show2();

    return 0;
}
```

Output:

```
This is the parent class.
This is the first child class.
This is the parent class.
This is the second child class.
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of **two or more types of inheritance** (e.g., multiple and hierarchical).

Example:

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

class Grandparent {
public:
    void grandparentFunction() {
        cout << "This is the grandparent class." << endl;
    }
};

class Parent1 : public Grandparent {
public:
    void parent1Function() {
        cout << "This is parent 1 class." << endl;
    }
};

class Parent2 : public Grandparent {
public:
    void parent2Function() {
        cout << "This is parent 2 class." << endl;
    }
};
```

```
class Child : public Parent1, public Parent2 {
public:
    void childFunction() {
        cout << "This is the child class." << endl;
    }
};

int main() {
    Child obj;
    obj.parent1Function();
    obj.parent2Function();
    obj.childFunction();
    return 0;
}
```

Output:

```
This is parent 1 class.
This is parent 2 class.
This is the child class.
```

Inheritance is a powerful feature in C++ that promotes **code reusability** and **modularity**. The different types of inheritance allow developers to design efficient and structured programs.

This Module covered:

- **Single Inheritance** (One class inherits from another)
- **Multilevel Inheritance** (A chain of inheritance)
- **Multiple Inheritance** (A class inherits from multiple classes)
- **Hierarchical Inheritance** (Multiple classes inherit from one base class)
- **Hybrid Inheritance** (Combination of multiple inheritance types)

3.6 Virtual Base Classes and Abstract Classes in C++

1. Virtual Base Classes

When a class is derived from multiple base classes, and these base classes further inherit from a common ancestor, **the common base class can be included multiple times in the final derived class**. This leads to the **Diamond Problem**, causing ambiguity in data access and redundancy in memory usage.

To **solve this issue**, **C++ provides Virtual Base Classes**. By making a base class **virtual**, only one copy of the base class members is inherited, even if multiple paths lead to the derived class.

The Diamond Problem (Before Using Virtual Base Class)

Example Without Virtual Base Class (Problematic Case)

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

class A {
public:
   int value;
};

class B : public A { };  // Inherits from A
class C : public A { };  // Inherits from A
class D : public B, public C { }; // Multiple Inheritance

int main() {
   D obj;
   // obj.value = 10; // ERROR: Ambiguity (value exists in both B and
C)
   obj.B::value = 10;  // Resolving ambiguity by specifying class
   obj.C::value = 20;  // Still leads to duplicate copies of A's data

   cout << "Value from B: " << obj.B::value << endl;
   cout << "Value from C: " << obj.C::value << endl; // Different
copies of 'value'
   return 0;
}
```

Solution Using Virtual Base Class

By making A a **virtual base class**, C++ ensures only **one** copy of A is inherited.

Syntax of Virtual Base Class

```
class Base {
   // Members
};

class Derived1 : virtual public Base { };
class Derived2 : virtual public Base { };
class FinalClass : public Derived1, public Derived2 { };
```

Example Using Virtual Base Class (No Ambiguity)

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

class A {
public:
    int value;
};

class B : virtual public A { };  // Virtual Inheritance
class C : virtual public A { };  // Virtual Inheritance
class D : public B, public C { }; // No ambiguity

int main() {
    D obj;
    obj.value = 30;  // No ambiguity
    cout << "Value: " << obj.value << endl;  // Output: 30
    return 0;
}
```

Key Advantages of Virtual Base Class

1. **Solves the Diamond Problem** – Only one copy of the base class members exists in memory.
2. **Prevents Data Redundancy** – Saves memory by avoiding duplicate copies.
3. **Removes Ambiguity** – No need to specify B::value or C::value.

2. Abstract Class

An **Abstract Class** in C++ is a class that **cannot be instantiated** and serves as a **blueprint for derived classes**. It contains at least **one pure virtual function**, forcing derived classes to provide an implementation.

Syntax of Abstract Class

```
class AbstractClass {
public:
    virtual void pureVirtualFunction() = 0; // Pure Virtual Function
};
```

Here, = 0 indicates that this function **must be overridden** in derived classes.

Example of Abstract Class

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

class Shape {
public:
    virtual void draw() = 0;  // Pure Virtual Function (Abstract Method)
};

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

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

int main() {
    // Shape obj; // ERROR: Cannot instantiate abstract class
    Circle c;
    Rectangle r;

    c.draw();  // Output: Drawing a Circle
    r.draw();  // Output: Drawing a Rectangle

    return 0;
}
```

Key Properties of Abstract Classes

1. **Cannot create objects** of an abstract class.
2. **Must have at least one pure virtual function**.

3. **Derived classes must override** the pure virtual function; otherwise, they remain abstract.

Use Case of Abstract Classes

Abstract classes are commonly used in **polymorphism** where multiple derived classes share a common interface.

Example: Abstract Class with Polymorphism

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

class Animal {
public:
    virtual void makeSound() = 0; // Pure virtual function
};

class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Dog Barks" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Cat Meows" << endl;
    }
};

void animalSound(Animal &a) {
    a.makeSound();
}

int main() {
    Dog d;
    Cat c;

    animalSound(d); // Output: Dog Barks
    animalSound(c); // Output: Cat Meows
    return 0;
}
```

Table 3.2: Difference Between Virtual Base Class and Abstract Class

Feature	**Virtual Base Class**	**Abstract Class**
Purpose	Solves multiple inheritance issues	Defines an interface for derived classes
Instantiation	Can be instantiated	Cannot be instantiated
Inheritance	Used to avoid duplicate base class instances	Used to enforce function overriding
Contains	Normal members, virtual inheritance	At least one pure virtual function

- **Virtual Base Classes** solve **multiple inheritance ambiguity** by ensuring only **one copy** of a base class is inherited.
- **Abstract Classes** act as **blueprints** for derived classes, enforcing function overriding and enabling **polymorphism**.
- Both concepts are crucial in **object-oriented programming (OOP)** to design **efficient and scalable** C++ applications.

This **comprehensive explanation** covers **theory, syntax, examples, and key differences**, making it easier to understand **Virtual Base Classes and Abstract Classes in C++**.

3.7 Constructors in Derived Classes

In **object-oriented programming**, a derived class inherits properties and behavior from a base class. When an object of a derived class is created, both the **base class constructor** and the **derived class constructor** are executed.

The constructor of the **base class is executed first**, followed by the constructor of the **derived class**. This ensures that the base class members are properly initialized before the derived class adds its own functionalities.

Syntax of Derived Class Constructor

The constructor of a derived class must first call the constructor of the base class. This is done using an **initializer list** in the derived class constructor.

```
class Base {
```

Notes

```
public:
    Base() {
        cout << "Base class constructor called" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived class constructor called" << endl;
    }
};
```

Example 1: Constructor Execution in Inheritance

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

class Base {
public:
    Base() {
        cout << "Base class constructor called" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived class constructor called" << endl;
    }
};

int main() {
    Derived obj; // Creating an object of the Derived class
    return 0;
}
```

Output:

```
Base class constructor called
Derived class constructor called
```

Parameterized Constructor in Derived Class

If the base class has a **parameterized constructor**, the derived class must explicitly call it in its **initializer list**.

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

class Base {
public:
   Base(int x) {
      cout << "Base class constructor called with value: " << x <<
endl;
   }
};

class Derived : public Base {
public:
   Derived(int y) : Base(y) { // Calling Base class constructor
      cout << "Derived class constructor called with value: " << y <<
endl;
   }
};

int main() {
   Derived obj(10);
   return 0;
}
```

Output:

```
Base class constructor called with value: 10
Derived class constructor called with value: 10
```

Order of Constructor Execution in Multiple Inheritance

If a derived class inherits from multiple base classes, the constructors of the base classes are executed in the order of inheritance.

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

class A {
public:
    A() {
        cout << "Constructor of A" << endl;
    }
};

class B {
public:
    B() {
        cout << "Constructor of B" << endl;
    }
};

class C : public A, public B { // Multiple inheritance
public:
    C() {
        cout << "Constructor of C" << endl;
    }
};

int main() {
    C obj;
    return 0;
}
```

Output:

```
Constructor of A
Constructor of B
Constructor of C
```

2. Member Classes (Nested Classes in C++)

A **member class** (also called a **nested class**) is a class that is defined inside another class. It has access to the private and protected members of the enclosing (outer) class.

Nested classes are used when **a class logically belongs inside another class**. They help in **encapsulation** and keeping related functionalities grouped together.

Syntax of Member Class

```
class Outer {
```

```
public:
    class Inner { // Nested class
    public:
        void display() {
            cout << "Inside Inner class" << endl;
        }
    };
};
```

Example 1: Basic Member Class

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

class Outer {
public:
    class Inner { // Nested class
    public:
        void show() {
            cout << "Inside Inner class" << endl;
        }
    };
};

int main() {
    Outer::Inner obj; // Creating object of Inner class
    obj.show();
    return 0;
}
```

Output:

```
Inside Inner class
```

Example 2: Accessing Private Members of Outer Class

The nested class can access **private members** of the outer class.

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

class Outer {
private:
    int data = 100;
public:
    class Inner {
    public:
        void display(Outer &obj) { // Accessing private member
            cout << "Value of data: " << obj.data << endl;
        }
    };
};

int main() {
    Outer obj1;
    Outer::Inner obj2;
    obj2.display(obj1);
    return 0;
}
```

Output:

```
Value of data: 100
```

Example 3: Constructor in Member Class

A nested class can have its own constructor.

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

class Outer {
public:
    class Inner {
    public:
        Inner() {
            cout << "Inner class constructor called" << endl;
        }
    };
};

int main() {
    Outer::Inner obj;
    return 0;
}
```

Output:

```
Inner class constructor called
```

Example 4: Nested Class with Methods Using Outer Class Members

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

class Outer {
private:
    int data = 42;

public:
    void showData() {
        cout << "Outer class data: " << data << endl;
    }

    class Inner {
    public:
        void display(Outer &obj) {
            obj.showData(); // Accessing Outer class function
        }
    };
};

int main() {
    Outer obj1;
    Outer::Inner obj2;
    obj2.display(obj1);
    return 0;
}
```

Output:

```
Outer class data: 42
```

Table 3.3 Key Differences: Constructors in Derived Classes vs. Member Classes

Feature	**Derived Class Constructor**	**Member Class Constructor**

Definition	Constructor of a derived class in an inheritance hierarchy	Constructor inside a nested class
Execution Order	Base class constructor → Derived class constructor	Only the member class constructor is executed
Access	Can access base class members (public/protected)	Can access private/protected members of the outer class
Use Case	Used when a class inherits from another	Used to define classes within a class for logical grouping

- **Constructors in derived classes** ensure that the base class is initialized before the derived class.
- **Nested (member) classes** allow structuring complex programs by logically grouping related classes together.
- Nested classes **can access private members** of the outer class if given proper access.

These concepts are useful in modular programming, encapsulation, and data abstraction, making C++ an efficient language for object-oriented programming.

MCQs:

1. What does inheritance in C++ allow you to do?
A. Create multiple constructors
B. Reuse code by deriving a new class from an existing class
C. Declare multiple variables
D. Use templates

2. Which of the following is the correct syntax for public inheritance in C++?
A. class Derived inherits Base
B. class Derived : public Base
C. class Base -> Derived
D. class Derived extends Base

3. What is a base class in C++?
A. A class that is used only once
B. A class that contains only static members
C. A class from which other classes are derived
D. A class with no constructors

4. Which type of inheritance involves a class being derived from two or more base classes?
A. Single inheritance
B. Multilevel inheritance
C. Hybrid inheritance
D. Multiple inheritance

5. What does polymorphism mean in object-oriented programming?
A. Using only one function in a program
B. Using a single interface to represent different types
C. Writing code without any class
D. Accessing private members directly

6. Which of the following enables runtime polymorphism in C++?
A. Function overloading
B. Operator overloading
C. Virtual functions
D. Static functions

7. What is function overloading an example of?
A. Runtime polymorphism
B. Compile-time polymorphism
C. Dynamic polymorphism
D. Multilevel inheritance

8. What will happen if a derived class overrides a base class function, but the base function is not declared virtual?
A. The derived class version is always called
B. The base class version is always called when using a base pointer

C. It causes a runtime error

D. Both functions will be executed

9. Which keyword is used to allow a derived class to redefine a base class function?

A. override

B. virtual

C. friend

D. static

10. What is the benefit of polymorphism in C++?

A. Reduces the size of executable files

B. Improves performance in all cases

C. Allows for flexible and reusable code design

D. Prevents object creation

Short Questions:

1. What is inheritance in C++?
2. Define a base class and a derived class with an example.
3. What are the types of inheritance supported in C++?
4. How does public inheritance differ from private inheritance in C++?
5. What is multiple inheritance? Give a simple example.
6. What is the main advantage of using inheritance in object-oriented programming?
7. Define polymorphism in the context of C++ OOP.
8. What is the difference between compile-time polymorphism and run-time polymorphism?
9. How is function overloading used to achieve polymorphism in C++?
10. What is the role of the virtual keyword in achieving run-time polymorphism?
11. What is function overriding, and how does it relate to polymorphism?
12. What happens if a base class function is not declared virtual and is overridden in a derived class?

Long Questions:

1. Explain the concept of inheritance in C++. How does it support code reusability? Provide a code example to illustrate your answer.
2. Differentiate between single, multiple, multilevel, and hierarchical inheritance in C++. Give examples of each.
3. What is the syntax for public, protected, and private inheritance in C++? How does the access level of base class members change in each case?
4. Describe how constructors and destructors behave in inheritance. What is the order of constructor and destructor calls in an inheritance hierarchy?
5. Write a C++ program that demonstrates multiple inheritance. Explain how ambiguity is resolved when two base classes have functions with the same name.
6. What is polymorphism in C++? Explain the difference between compile-time and run-time polymorphism with appropriate code examples.
7. How does function overloading implement compile-time polymorphism in C++? Give at least two examples with different parameter lists.
8. Explain the concept of function overriding in C++. How does it support run-time polymorphism? Provide a sample program.
9. What is the significance of the virtual keyword in C++? How does it affect function binding and polymorphism?
10. Write a C++ program to demonstrate run-time polymorphism using base class pointers and virtual functions. Explain how dynamic dispatch works.
11. What are pure virtual functions and abstract classes in C++? How are they used to implement interfaces in object-oriented programming?
12. Discuss the advantages and potential pitfalls of using inheritance and polymorphism in object-oriented design. How can improper use of these features affect software maintainability?

MODULE 4
POINTER, VIRTUAL FUNCTION AND POLYMORPHISM

4.0 LEARNING OUTCOMES

By the end of this Module, students will be able to:

- Understand pointers and their use in objects and "this" pointer.
- Implement pointers to derived classes for dynamic object handling.
- Explore virtual functions and pure virtual functions in C++.
- Understand polymorphism, including compile-time and run-time polymorphism.
- Differentiate between function overloading and function overriding.
- This Module provides a deep understanding of pointers, virtual functions, and polymorphism, essential for dynamic and efficient object-oriented programming.

Unit 10: Pointers

4.1 Pointers in C++

A **pointer** is a variable that stores the **memory address** of another variable. Pointers are powerful in C++ as they enable **dynamic memory allocation, efficient data manipulation, and object-oriented programming techniques.**

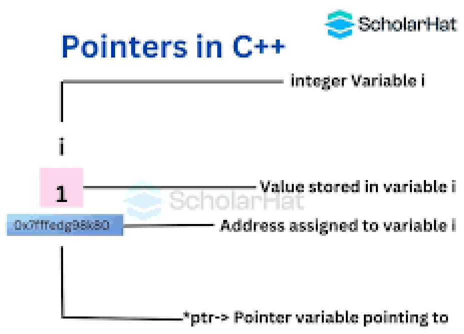

ScholarHat
Pointers in C++
integer Variable i
i
1
Value stored in variable i
0x7ffedg98k80
Address assigned to variable i
*ptr-> Pointer variable pointing to

Figure 3 Concept of Pointers in OOP'S
[Source https://www.scholarhat.com]

Syntax of a Pointer

```
data_type* pointer_name;  // Declaring a pointer
```

Example: Declaring and Using a Pointer

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

int main() {
   int num = 10;
   int* ptr = &num; // Pointer storing the address of num

   cout << "Value of num: " << num << endl;
   cout << "Address of num: " << &num << endl;
   cout << "Value stored in pointer ptr: " << ptr << endl;
   cout << "Value accessed using pointer: " << *ptr << endl; //
Dereferencing

   return 0;
}
```

Output

```
Value of num: 10
Address of num: 0x7ffee7b0b80c
Value stored in pointer ptr: 0x7ffee7b0b80c
Value accessed using pointer: 10
```

1. Pointers to Objects

In C++, pointers can also store the **addresses of objects** of a class. This allows **dynamic allocation** of objects and facilitates **polymorphism and efficient object handling**.

Syntax of Pointers to Objects

```
class ClassName {
    // Class members
};

ClassName* objPointer;  // Pointer to an object of ClassName
```

Example: Using a Pointer to an Object

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

class Student {
public:
    string name;
    int age;

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Student s1 = {"John", 20}; // Normal object
    Student* ptr = &s1; // Pointer to object

    // Accessing members using the pointer
    cout << "Using pointer: " << ptr->name << ", " << ptr->age << 
endl;
    ptr->display(); // Using -> to access function
    return 0;
}
```

Output

```
Using pointer: John, 20
Name: John, Age: 20
```

Dynamic Memory Allocation for Objects

We can use the new keyword to dynamically allocate objects at runtime.

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

class Student {
public:
    string name;
    int age;

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Student* ptr = new Student(); // Dynamically allocating an object

    // Assigning values
    ptr->name = "Alice";
    ptr->age = 22;

    ptr->display();

    delete ptr; // Free allocated memory

    return 0;
}
```

Output

```
Name: Alice, Age: 22
```

2. This Pointer

This pointer is an **implicit pointer** available in all **non-static** member functions of a class. It **stores the address of the calling object** and helps in distinguishing between local and member variables when they have the same name.

Syntax of this Pointer

```
class ClassName {
public:
    void function() {
        cout << "Address of current object: " << this << endl;
    }
};
```

Example: Using this Pointer

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

class Car {
public:
    string brand;
    int price;

    void setValues(string brand, int price) {
        this->brand = brand;  // Using this-> to refer to member variable
        this->price = price;
    }

    void display() {
        cout << "Brand: " << brand << ", Price: " << price << endl;
        cout << "Address of current object: " << this << endl;
    }
};

int main() {
    Car c1, c2;

    c1.setValues("Toyota", 20000);
    c2.setValues("Honda", 18000);
    c1.display();
    c2.display();

    return 0;
}
```

Output

```
Brand: Toyota, Price: 20000
Address of current object: 0x61ff08
Brand: Honda, Price: 18000
Address of current object: 0x61ff04
```

Advantages of this Pointer

1. **Avoids naming conflicts** between member variables and function parameters.
2. **Used for returning object reference** in function chaining.
3. **Helps in operator overloading and method chaining.**

3. **Returning Object using** this **Pointer**

The this pointer can be used to return the **current object reference**, enabling function chaining.

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

class Person {
public:
    string name;
    int age;

    Person* setName(string name) {
        this->name = name;
        return this;  // Returning object reference
    }

    Person* setAge(int age) {
        this->age = age;
        return this;  // Returning object reference
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p1;
    p1.setName("Michael")->setAge(25)->display();       //       Chained
function calls

    return 0;
}
```

Output

```
Name: Michael, Age: 25
```

In this Module, we explored **pointers in C++, pointers to objects, and the this pointer.**

- **Pointers** store memory addresses and allow efficient manipulation of variables and objects.
- **Pointers to objects** enable dynamic memory allocation and flexible object handling.
- **The this pointer** is an implicit pointer referring to the calling object, helping in method chaining and resolving naming conflicts.

Pointer to Derived Classes in C++

In C++, pointers play a crucial role in handling objects dynamically. When working with **inheritance**, we often use pointers to **base** and **derived** classes to achieve **polymorphism**. A **pointer to a derived class** allows accessing members of both the **base** and **derived** classes using a base class pointer.

4.2 Concept of Pointer to Derived Class

A **pointer to a base class** can hold the **address of a derived class object**. However, when accessed through the base class pointer, it can only use the **members of the base class** unless virtual functions are used.

Key Points:

- A **base class pointer** can point to a **derived class object**.
- It can access only the **base class members** (unless polymorphism is used).
- If **virtual functions** are present, the derived class function gets executed (dynamic binding).

1. Syntax of Pointer to Derived Class

The general syntax for creating a **pointer to a derived class** is:

```
BaseClass *ptr;  // Pointer to Base Class
DerivedClass obj;
ptr = &obj; // Base class pointer pointing to Derived class object
```

Since the pointer is of the **base class type**, it can only access base class members. To access derived class members, we either use **type casting** or **virtual functions**.

2. Example Without Virtual Functions

When a **base class pointer** points to a **derived class object**, it only accesses base class members unless virtual functions are used.

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

class Base {
public:
    void show() {
        cout << "Base class show function" << endl;
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "Derived class show function" << endl;
    }
};

int main() {
    Base *ptr; // Base class pointer
    Derived obj;
    ptr = &obj; // Base class pointer points to derived class object

    ptr->show(); // Calls Base class function

    return 0;
}
```

Output:

```
Base class show function
```

Explanation:

- The **base class pointer** (ptr) stores the address of a **derived class object (obj)**.
- However, since show() is **not virtual**, the **base class version** is called, ignoring the derived class function.

3. Example Using Virtual Functions

To achieve **runtime polymorphism**, we use the **virtual keyword** in the base class function. This enables **dynamic binding**, allowing the derived class function to be called even when accessed via a base class pointer.

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

class Base {
public:
    virtual void show() { // Virtual function
        cout << "Base class show function" << endl;
    }
};

class Derived : public Base {
public:
    void show() override { // Overrides base class function
        cout << "Derived class show function" << endl;
    }
};

int main() {
    Base *ptr; // Base class pointer
    Derived obj;
    ptr = &obj; // Base class pointer points to derived class object

    ptr->show(); // Calls Derived class function (Dynamic Binding)

    return 0;
}
```

Output:

```
Derived class show function
```

Explanation:

- The show() function in the **base class** is declared **virtual**.
- This enables **dynamic binding**, so the **derived class version** gets executed when called through the base class pointer.

4. Accessing Derived Class Members Using Base Class Pointer

Since a **base class pointer** cannot access derived class members directly, we use **typecasting**.

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

class Base {
public:
    void showBase() {
        cout << "Base class function" << endl;
    }
};

class Derived : public Base {
public:
    void showDerived() {
        cout << "Derived class function" << endl;
    }
};

int main() {
    Base *ptr; // Base class pointer
    Derived obj;

    ptr = &obj; // Base class pointer points to derived class object
    ptr->showBase(); // Allowed
    // ptr->showDerived(); // Error: Not accessible through base class
pointer

    // Accessing derived class function using typecasting
    ((Derived*)ptr)->showDerived();

    return 0;
}
```

Output:

```
Base class function
Derived class function
```

Explanation:

- The **base class pointer** (ptr) can access only showBase().
- To access showDerived(), we use **typecasting**: ((Derived*)ptr)->showDerived();.

5. Pointer to Derived Class in Multiple Inheritance

When using **multiple inheritance**, a base class pointer can still access members of the derived class.

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

class Base1 {
public:
    virtual void show() {
        cout << "Base1 class function" << endl;
    }
};

class Base2 {
public:
    void display() {
        cout << "Base2 class function" << endl;
    }
};

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

```
public:
    void show() override {
        cout << "Derived class function" << endl;
    }
};

int main() {
    Base1 *ptr;
    Derived obj;

    ptr = &obj;
    ptr->show(); // Calls Derived class function

    return 0;
}
```

Output:

```
Derived class function
```

6. Pointer to Derived Class and Virtual Destructor

If a base class has a **non-virtual destructor**, deleting a derived class object through a base class pointer causes **memory leaks**. This is solved by using a **virtual destructor**.

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

class Base {
public:
    Base() { cout << "Base Constructor" << endl; }
    virtual ~Base() { cout << "Base Destructor" << endl; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor" << endl; }
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Base *ptr = new Derived(); // Allocates memory for derived class
    delete ptr; // Calls derived class destructor properly

    return 0;
}
```

Output:

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

Explanation:

- Using a **virtual destructor** ensures the derived class destructor is called properly, preventing memory leaks.

Summary

Table 4.1 Features and Behavior of Virtual Function

Feature	**Behavior**
Base class pointer	Can store derived class object address
Without virtual function	Calls base class function
With virtual function	Calls derived class function (polymorphism)
Accessing derived class members	Requires typecasting
Virtual destructor	Ensures proper cleanup in inheritance

Pointers to derived classes are essential for achieving **polymorphism** in C++. Using **virtual functions**, we ensure that derived class functions override base class functions correctly. Proper use of **virtual destructors** avoids memory leaks when working with dynamically allocated objects.

This topic is fundamental in **object-oriented programming (OOP)** and is widely used in designing **reusable and flexible software architectures**.

Unit 11: Virtual Function, Pure Virtual Function

4.3 Virtual Function and Pure Virtual Function in C++

In C++, **polymorphism** allows objects of different classes to be treated as objects of a common base class. This is achieved using **virtual functions**.

A **virtual function** is a function in a base class that is **overridden** in a derived class and is resolved **at runtime** using **dynamic binding** (late binding). This allows C++ to call the correct function based on the actual object type rather than the pointer type.

Why Use Virtual Functions?

- To enable **runtime polymorphism**.
- To ensure that the **correct function** of the derived class is called even when accessed through a base class pointer.
- To allow **overriding** of base class methods in derived classes.

1. Syntax of Virtual Function

A virtual function is declared using the keyword **virtual** in the base class.

```
class Base {
public:
    virtual void display() {  // Virtual function
        cout << "Base class display function" << endl;
    }
};
```

When a derived class overrides the virtual function, C++ ensures that the correct function is called at runtime.

2. Example of Virtual Function

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

class Base {
public:
    virtual void show() {  // Virtual function
        cout << "Base class show() function" << endl;
    }
};

class Derived : public Base {
```

```
public:
    void show() override {  // Overriding base class function
        cout << "Derived class show() function" << endl;
    }
};

int main() {
    Base* basePtr;  // Base class pointer
    Derived derivedObj;

    basePtr = &derivedObj;  // Base class pointer points to derived class
object

    basePtr->show();  // Calls Derived class function due to late binding

    return 0;
}
```

Output:

```
Derived class show() function
```

Explanation:

- The **show()** function is declared as virtual in the Base class.
- The **Derived** class overrides the show() function.
- When calling basePtr->show(), the derived class function is called because of **dynamic binding (late binding).**

3. Virtual Function Behavior

Calling a Virtual Function without a Derived Function

If a virtual function is **not overridden** in the derived class, the **base class version** is called.

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

class Base {
public:
    virtual void show() {
        cout << "Base class function" << endl;
    }
};
```

```
class Derived : public Base {
    // No override here
};

int main() {
    Derived obj;
    obj.show();  // Calls Base class function
    return 0;
}
```

Output:

```
Base class function
```

Accessing Base Class Virtual Function

The base class function can still be accessed using **scope resolution operator (::)**.

```
basePtr->Base::show();
```

4. Pure Virtual Function (Abstract Class)

A **pure virtual function** is a **virtual function with no implementation** in the base class.

- Declared using = 0.
- It makes a class **abstract**, meaning **it cannot be instantiated**.
- Any derived class **must override** the pure virtual function.

Syntax of Pure Virtual Function

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

5. Example of Pure Virtual Function

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

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

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

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

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw();
    shape2->draw();

    delete shape1;
    delete shape2;

    return 0;
}
```

Output:

```
Drawing a Circle
Drawing a Rectangle
```

Explanation:

- Shape is an **abstract class** with a **pure virtual function draw()**.
- Circle and Rectangle **override** the draw() function.
- We **create pointers** of Shape type but assign Circle and Rectangle objects.
- The **correct function is called at runtime**.

6. Key Differences Between Virtual and Pure Virtual Functions

7. Real-Life Example: Employee Salary Calculation

Feature	Virtual Function	Pure Virtual Function
Definition	Declared using virtual keyword.	Declared using = 0 syntax.
Implementation in Base Class	Can have a definition.	No definition (abstract method).
Derived Class Requirement	Can be overridden, but not mandatory.	Must be overridden in derived class.
Instantiation	Base class can be instantiated.	Base class **cannot** be instantiated (abstract class).

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

class Employee {
public:
    virtual void calculateSalary() = 0; // Pure virtual function
};

class FullTime : public Employee {
public:
    void calculateSalary() override {
        cout << "Full-time Employee Salary Calculated" << endl;
    }
};

class PartTime : public Employee {
public:
    void calculateSalary() override {
        cout << "Part-time Employee Salary Calculated" << endl;
    }
};

int main() {
    Employee* emp1 = new FullTime();
    Employee* emp2 = new PartTime();

    emp1->calculateSalary();
    emp2->calculateSalary();

    delete emp1;
    delete emp2;

    return 0;
}
```

Output:

```
Full-time Employee Salary Calculated
Part-time Employee Salary Calculated
```

Conclusion

- **Virtual functions** allow **runtime polymorphism**, enabling C++ to call the correct function dynamically.
- **Pure virtual functions** enforce **mandatory overriding**, making a class **abstract**.
- Virtual functions make code **flexible** and **scalable** by supporting **dynamic dispatch**.

Unit 12: Polymorphism: Compile Time, Run Time

4.4 Polymorphism in C++

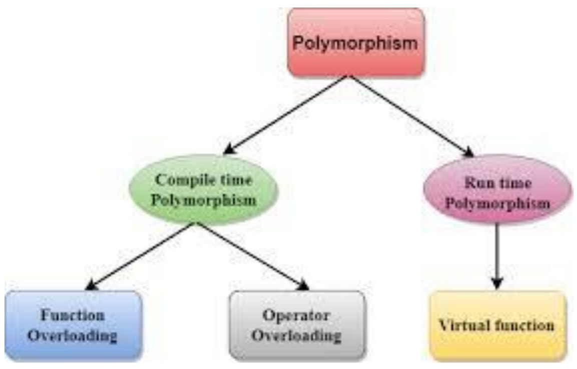


Figure 4 Types of Polymorphism
[*https://www.tpointtech.com]*

Polymorphism is one of the four fundamental principles of **Object-Oriented Programming (OOP)** in C++. The word "Polymorphism" is derived from the Greek words **"poly" (many) and "morph" (forms)**, meaning the ability to take multiple forms.

In C++, **polymorphism** allows a function or an operator to behave differently in different contexts. It provides flexibility and reusability in programs, reducing code duplication.

Polymorphism is broadly classified into **two types:**

1. **Compile-time Polymorphism (Static Binding or Early Binding)**
2. **Run-time Polymorphism (Dynamic Binding or Late Binding)**

Let's understand each type with **theory, syntax, and examples**.

1. Compile-Time Polymorphism (Static Binding)

Compile-time polymorphism is achieved through **Function Overloading and Operator Overloading**. In this type, the function call is resolved at **compile time**.

Function Overloading

Function Overloading allows multiple functions with the **same name** but **different parameter lists**. The compiler determines which function to call based on the **arguments passed**.

Syntax of Function Overloading

return_type function_name(parameter_list1);
return_type function_name(parameter_list2);

Example: Function Overloading

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

class Calculator {
public:
    // Function to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Function to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Function to add two floating-point numbers
    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;

    cout << "Addition of 2 and 3: " << calc.add(2, 3) << endl;
    cout << "Addition of 2, 3, and 4: " << calc.add(2, 3, 4) << endl;
    cout << "Addition of 2.5 and 3.5: " << calc.add(2.5, 3.5) << endl;

    return 0;
}
```

Output:

```
Addition of 2 and 3: 5
Addition of 2, 3, and 4: 9
Addition of 2.5 and 3.5: 6
```

Operator Overloading

Operator Overloading allows **operators** to be redefined for user-defined types (like classes).

Syntax of Operator Overloading

```
return_type operator symbol (parameters) {
    // Code for overloaded operator
}
```

Example: Operator Overloading

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

class Complex {
public:
    int real, imag;

    Complex(int r, int i) {
        real = r;
        imag = i;
    }

    // Overloading the '+' operator
    Complex operator+(Complex c) {
        return Complex(real + c.real, imag + c.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(5, 6);
    Complex c3 = c1 + c2; // Calls the overloaded operator
    c3.display();

    return 0;
}
```

Output:

```
8 + 10i
```

Key Points:

- In **Function Overloading**, multiple functions have the same name but different parameters.
- In **Operator Overloading**, operators like +, -, *, etc., can be redefined for user-defined data types.
- Both these techniques help in achieving **compile-time polymorphism**.

2. Run-Time Polymorphism (Dynamic Binding)

Run-time polymorphism is achieved through **Function Overriding and Virtual Functions**. In this type, the function call is resolved at **run time** using a **pointer or reference to the base class**.

Function Overriding

Function Overriding allows a **derived class** to provide a **specific implementation** of a function that is already defined in the **base class**. The function in the derived class **must have the same name and parameters** as in the base class.

Syntax of Function Overriding

```
class Base {
public:
    virtual void show() {
        cout << "Base class function" << endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived class function" << endl;
    }
};
```

Example: Function Overriding

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

class Base {
public:
    virtual void show() {
        cout << "Base class function" << endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived class function" << endl;
    }
};

int main() {
    Base* ptr;
    Derived obj;
    ptr = &obj; // Base class pointer points to Derived class object
    ptr->show(); // Calls Derived class function

    return 0;
}
```

Output:

```
Derived class function
```

Key Points:

- **Virtual functions** ensure that the correct function is called for an object, regardless of the reference type.
- **Function Overriding** occurs when a derived class provides a different implementation of a function in the base class.

Table 4.2 Comparison: Compile-Time vs. Run-Time Polymorphism

Feature	**Compile-Time Polymorphism**	**Run-Time Polymorphism**
Binding Type	Early Binding (Static)	Late Binding (Dynamic)
Achieved By	Function Overloading, Operator Overloading	Function Overriding (Using Virtual Functions)
Function Call Resolved At	Compile-Time	Run-Time
Speed	Faster	Slightly Slower
Example	Multiple add() functions	Base class pointer calling a derived class function

Polymorphism is an essential feature of **Object-Oriented Programming (OOP)** in C++.

- **Compile-Time Polymorphism** (Function Overloading, Operator Overloading) improves code reusability and efficiency.
- **Run-Time Polymorphism** (Function Overriding, Virtual Functions) allows flexibility and dynamic behavior in programs.

4.5 Overloading and Overriding in C++

In C++, **overloading and overriding** are two key concepts used in **polymorphism**, which allows the same function name or operator to have different behaviors. These concepts help in making code more readable, reusable, and efficient.

- **Function Overloading** allows multiple functions with the same name but different parameters.
- **Operator Overloading** enables the redefinition of operators for user-defined data types.
- **Method Overriding** allows a derived class to provide a specific implementation of a base class function.

1. Function Overloading

Function overloading is a feature in C++ that allows multiple functions with the same name but different parameter lists to exist. The compiler determines which function to call based on the number and type of arguments passed.

Syntax

```
return_type function_name(parameter_list1);
return_type function_name(parameter_list2);
```

Example of Function Overloading

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

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Function to add three integers
int add(int a, int b, int c) {
    return a + b + c;
}

// Function to add two floating-point numbers
float add(float a, float b) {
    return a + b;
}

int main() {
    cout << "Addition of 2 and 3: " << add(2, 3) << endl;
    cout << "Addition of 2, 3, and 5: " << add(2, 3, 5) << endl;
    cout << "Addition of 2.5 and 3.5: " << add(2.5f, 3.5f) << endl;
    return 0;
}
```

Output:

```
Addition of 2 and 3: 5
Addition of 2, 3, and 5: 10
Addition of 2.5 and 3.5: 6
```

Rules for Function Overloading

1. Functions must have the **same name**.
2. Functions must have **different parameter lists** (number or type of arguments).
3. Functions **cannot be overloaded by return type alone**.

2. Operator Overloading

Definition

Operator overloading allows defining the behavior of **operators** (+, -, *, /, ==, etc.) for user-defined data types like classes and structures.

Syntax

```
return_type operator symbol (parameters) {
    // Function body
}
```

Example of Operator Overloading

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

class Complex {
    public:
        int real, imag;
    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }

    // Overloading + operator
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2; // Calls operator overload function
    c3.display();
    return 0;
}
```

Output:

```
4 + 6i
```

Rules for Operator Overloading

1. Only **existing operators** can be overloaded.
2. Cannot overload **sizeof, ::, ., . or ?:**.
3. Overloaded operators must have **at least one user-defined data type operand**.

3. Function Overriding

Definition

Function overriding allows a **derived class** to provide a specific implementation of a function that is already defined in its **base class**.

Syntax

```
class Base {
```

```
public:
    virtual void show() {
        cout << "Base class function";
    }
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived class function";
    }
};
```

Example of Function Overriding

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

class Base {
public:
    virtual void display() {
        cout << "Base class function" << endl;
    }
};

class Derived : public Base {
public:
    void display() override { // Overriding base class method
        cout << "Derived class function" << endl;
    }
};

int main() {
    Base* basePtr;
    Derived obj;
    basePtr = &obj;
    basePtr->display(); // Calls derived class method
    return 0;
}
```

Output:

```
Derived class function
```

Key Rules for Overriding

1. The **function name and parameters** must match exactly with the base class function.
2. The base class function must be marked as **virtual** to enable runtime polymorphism.
3. If overridden incorrectly, the **base class function gets called instead of the derived class function**.

Table 4.3 Differences Between Overloading and Overriding

Feature	**Function Overloading**	**Function Overriding**
Definition	Multiple functions with the same name but different parameters.	Redefining a base class function in a derived class.
Where It Occurs	Same class.	Different classes (base and derived).
Parameters	Must be different.	Must be the same.
Return Type	Can be different.	Must be the same.
Virtual Keyword	Not required.	Requires virtual in the base class.
Purpose	Achieves **compile-time polymorphism**.	Achieves **runtime polymorphism**.

Both **overloading and overriding** are essential concepts in C++ that help achieve **polymorphism**:

- **Function Overloading** enhances code readability and flexibility by allowing multiple functions with the same name but different signatures.
- **Operator Overloading** allows defining custom behaviors for operators in user-defined classes.
- **Function Overriding** enables a derived class to modify the behavior of an inherited function, supporting runtime polymorphism.

MCQs:

1. What is operator overloading in C++?

A. Replacing built-in operators with macros

B. Assigning multiple meanings to an operator based on context

C. Changing the syntax of operators

D. Restricting operator use

2. Which keyword is used to overload an operator in C++?

A. override

B. define

C. operator

D. opload

3. Which of the following operators cannot be overloaded in C++?

A. +

B. ==

C. =

D. ::

4. How is an overloaded operator function typically defined in a class?

A. As a constructor

B. As a friend function or member function

C. As a template

D. As a macro

5. What is the return type of a type conversion operator function in C++?

A. void

B. Same as the class name

C. The target type being converted to

D. Always int

6. What is the correct syntax for defining a conversion operator in a class?

A. convert() {}

B. operator int() {}

C. int operator() {}

D. type convert operator() {}

7. Which of the following is not a rule of operator overloading?

A. You can’t change the precedence of operators

B. You can’t create new operators

C. You can overload all operators including ::

D. You can change the meaning of existing operators

8. Which type of operator overloading is used when defining operations between two different user-defined types?

A. Unary operator overloading

B. Binary operator overloading

C. Relational operator overloading

D. Ternary operator overloading

9. Can constructors be used for implicit type conversion in C++?

A. Yes

B. No

C. Only with virtual functions

D. Only in templates

10. What is the primary benefit of operator overloading?

A. Code becomes more complex

B. It allows the creation of new operators

C. It increases the size of the program

D. It allows intuitive use of custom data types

Short Questions:

1. What is operator overloading in C++?
2. Which keyword is used to overload an operator in C++?
3. Name any two operators that cannot be overloaded in C++.
4. What is the difference between a member function and a friend function when overloading operators?
5. What is the general syntax for overloading a binary operator in a class?
6. What are the rules for operator overloading in C++? Mention any two.
7. How is unary operator overloading different from binary operator overloading?
8. What is type conversion in C++?
9. Can a constructor be used for implicit type conversion? Explain briefly.
10. What is a type conversion operator? Provide an example.
11. How do you define a conversion operator from a class type to int?
12. Why is operator overloading useful in object-oriented programming?

Long Questions:

1. Explain the concept of operator overloading in C++. Why is it used in object-oriented programming? Provide an example.
2. Describe the steps and syntax for overloading a binary operator using a member function. Illustrate with a suitable program.
3. How can friend functions be used to overload operators in C++? Discuss with a detailed example.
4. Compare and contrast overloading unary and binary operators. Provide code examples for both.
5. What are the limitations and rules of operator overloading in C++? Mention at least four important rules.
6. Write a C++ program to overload the + operator for a custom Complex class to add two complex numbers. Explain the output.
7. What is a type conversion in C++? Discuss the different types of type conversions supported in C++.
8. How is a constructor used for single-argument type conversion in C++? Provide a program to demonstrate this concept.
9. What is a type conversion operator? Write a C++ program to convert a class type to a built-in type using a conversion operator.
10. Discuss the importance of type conversion operators in class design. How do they improve usability of custom data types?
11. Explain with code how to perform conversion from one user-defined type to another user-defined type in C++.
12. What are the potential pitfalls of operator overloading and type conversion in C++? How can they be avoided in large-scale software development?

MODULE 5
Exception Handling and File Handling

LEARNING OUTCOMES

By the end of this Module, students will be able to:

- Understand exception operations in C++ for user interaction.
- Learn about input and output streams.
- Implement formatted and unformatted I/O operations.
- Explore file handling concepts, including file streams, opening, reading, writing, and closing files.
- Understand file modes and their impact on data handling.
- Implement sequential and random file access techniques.

Unit 13: Stream Classes

5.1 Exception Operations and File Handling

Exception Operations and File Handling in C++

1. Introduction to exception Operations

C++ provides powerful functionalities through its standard I/O library. The most commonly used objects for console input and output operations are:

- *cin** (Standard Input) - Used for reading input from the keyboard.*
- *cout** (Standard Output) - Used for displaying output on the console.*
- *cerr** (Standard Error) - Used for error messages.*
- *clog** (Standard Log) - Used for logging information.*

2. Basic Exception operations

***2.1 Output using ****cout*

*The **cout** object, defined in the **<iostream>** header, is used for output.*

Syntax:

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

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

Example:

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

int main() {
    int a = 10;
    float b = 5.5;
    cout << "Integer: " << a << " and Float: " << b << endl;
    return 0;
}
```

***2.2 Input using ****cin*

*The **cin** object is used for taking input from the user.*

Syntax:

```
cin >> variable;
```

Example:

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

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

***2.3 Using cerr and ****clog*

*cerr** is used for error messages and does not support buffering, whereas **clog** supports buffering.*

Example:

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

int main() {
   cerr << "Error: Invalid input!" << endl;
   clog << "Log: This is a log message." << endl;
   return 0;
}
```

3. Manipulators in Console I/O

Manipulators help format the output.

3.1 Common Manipulators

- *endl** - Moves to the next line.*
- *setw(n)** - Sets the field width.*
- *setprecision(n)** - Controls floating-point precision.*
- *fixed** - Displays floating-point numbers in fixed-point notation.*
- *showpoint** - Forces decimal points in floating numbers.*
- *left** / **right** - Aligns text to the left or right.*

Example:

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;

int main() {
    double pi = 3.14159265;
    cout << fixed << setprecision(2) << pi << endl;  // Output: 3.14
    cout << setw(10) << right << "Hello" << endl;     // Right aligned
output
    return 0;
}
```

4. File Handling in C++

File handling allows us to store and retrieve data from files. The ***<fstream>*** *library provides three classes:*

- *ofstream** - Output file stream (write to a file)*
- *ifstream** - Input file stream (read from a file)*
- *fstream** - File stream (read/write to a file)*

***4.1 Writing to a File using ****ofstream*

Example:

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

int main() {
    ofstream file("example.txt");
    if (file.is_open()) {
        file << "Hello, file handling in C++!";
        file.close();
    } else {
        cout << "Unable to open file";
    }
    return 0;
}
```

***4.2 Reading from a File using ****ifstream*

Example:

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

int main() {
    ifstream file("example.txt");
    string line;
    if (file.is_open()) {
        while (getline(file, line)) {
            cout << line << endl;
        }
        file.close();
    } else {
        cout << "Unable to open file";
    }
    return 0;
}
```

***4.3 Read and Write Using ****fstream*

Example:

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

int main() {
    fstream file("example.txt", ios::in | ios::out | ios::app);
    file << "Appending new data!" << endl;
    file.close();
    return 0;
}
```

5. File Opening Modes

Table 5.1 Different modes available for file handling

Mode	***Description***
ios::in	*Opens file for reading*
ios::out	*Opens file for writing*
ios::app	*Appends to a file*
ios::ate	*Moves the write pointer to end*
ios::trunc	*Deletes contents of file if exists*

6. Checking File Status

Checking if a file exists or if an operation was successful.

Example:

```
#include <iostream>
```

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

int main() {
    ifstream file("example.txt");
    if (file) {
        cout << "File exists." << endl;
    } else {
        cout << "File does not exist." << endl;
    }
    return 0;
}
```

7. Reading and Writing Objects to Files

*Using **structures and classes** to store object data in files.*

7.1 Writing Objects to a File

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

class Student {
public:
    char name[20];
    int age;
};

int main() {
    Student s = {"John", 20};
    ofstream file("student.dat", ios::binary);
    file.write((char*)&s, sizeof(s));
    file.close();
    return 0;
}
```

7.2 Reading Objects from a File

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

class Student {
```

```
public:
    char name[20];
    int age;
};

int main() {
    Student s;
    ifstream file("student.dat", ios::binary);
    file.read((char*)&s, sizeof(s));
    cout << "Name: " << s.name << " Age: " << s.age << endl;
    file.close();
    return 0;
}
```

C++ provides robust ***console I/O operations*** using ***cin***, ***cout***, ***cerr***, and ***clog***. It also offers powerful ***file handling*** using ***fstream***, ***ifstream***, and ***ofstream***. Understanding these con*cepts helps in building real-world applications requiring persistent data storage.*

Unit 14: File Handling in OOP's

5.2. Introduction

Object-Oriented Programming (OOP) is a paradigm based on the concept of objects — which contain data (attributes) and methods (functions). In file handling, using OOP improves code organization, reusability, and scalability, especially in larger projects.

By encapsulating file operations inside classes, we can create more structured and reusable code.

5.3. Why Use OOP for File Handling?

Traditional (procedural) file handling works fine for simple tasks. However, OOP offers several advantages:

- Encapsulation of file operations.
- Easier maintenance and debugging.
- Promotes code reuse through inheritance.
- Makes it easy to build more complex systems (like file managers, parsers, etc.).

1. Creating a File Handler Class

Let’s define a class that can handle basic file operations:

```
class FileHandler:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def open_file(self):
        try:
            self.file = open(self.filename, self.mode)
            print(f"File '{self.filename}' opened successfully in
'{self.mode}' mode.")
        except Exception as e:
            print(f"Error opening file: {e}")

    def read_file(self):
        if self.file and not self.file.closed:
            return self.file.read()
        else:
            return "File not open."

    def write_file(self, data):
        if self.file and not self.file.closed:
            self.file.write(data)
        else:
            print("File not open.")

    def close_file(self):
        if self.file:
            self.file.close()
            print(f"File '{self.filename}' closed.")
```

2. Using the FileHandler Class

```
# Writing to a file
writer = FileHandler("demo.txt", "w")
writer.open_file()
writer.write_file("Hello from OOP-based file handler!\n")
writer.close_file()

# Reading from the same file
reader = FileHandler("demo.txt", "r")
reader.open_file()
content = reader.read_file()
print("File Content:\n", content)
reader.close_file()
```

3. Inheritance in File Handling

Let's extend our class to specialize in handling text files and binary files separately.

```
class TextFileHandler(FileHandler):
    def count_lines(self):
        if self.file and not self.file.closed:
            return len(self.file.readlines())
        else:
            return 0
```

Usage:

```
reader = TextFileHandler("demo.txt", "r")
reader.open_file()
lines = reader.count_lines()
print("Number of lines:", lines)
reader.close_file()
```

You can similarly create a BinaryFileHandler for binary file operations.

4. Exception Handling in OOP File Handling

Add more robust error management with try-except inside class methods:

```
def write_file(self, data):
    try:
        if self.file and not self.file.closed:
            self.file.write(data)
        else:
            print("File not open.")
    except Exception as e:
        print(f"Error writing to file: {e}")
```

5. Real-World Application: Log File Manager

```
class LogFileManager(FileHandler):
    def log(self, message):
        from datetime import datetime
        timestamp = datetime.now().strftime("%Y-%m-%d
%H:%M:%S")
        self.write_file(f"[{timestamp}] {message}\n")
```

Usage:

```
logger = LogFileManager("log.txt", "a")
logger.open_file()
logger.log("System started")
logger.log("User login successful")
logger.close_file()
```

5.4 Advantages of OOP-based File Handling

Given are the benefit of the OOP, based on file handling.

Table 5.2 Features and Benefit

Feature	***Benefit***
Encapsulation	*Keeps file logic isolated and clean.*
Inheritance	*Enables code reuse and extension for different file types*
Polymorphism	*Allows different file handlers to share method names but with different behaviors.*
Abstraction	*Hides complex file logic behind simple method calls.*

.

5.5. Best Practices

- Use context managers (with open(...)) inside methods to auto-close files.
- Always validate the file state before reading/writing.
- Use custom exceptions for better debugging.
- Avoid hardcoding file names; use parameters or configuration files.

File handling in OOP allows you to build scalable, readable, and reusable systems for interacting with files. By wrapping file operations in classes and methods, you gain the power of modular programming while keeping your code organized.

MCQs:

1. Which keyword is used to define a block of code that might throw an exception in C++?

A. throw

B. catch

C. try

D. handle

2. What is the correct keyword to catch an exception in C++?

A. try

B. throw

C. catch

D. except

3. What does the throw keyword do in C++ exception handling?

A. Declares an error

B. Ignores an error

C. Transfers control to the catch block

D. Closes a file

4. Which of the following is the base class for all standard exceptions in C++?

A. exception

B. error

C. std_error

D. base_exception

5. What happens if an exception is thrown but not caught in C++?

A. Program continues as normal

B. The exception is logged

C. The program terminates

D. The OS handles it automatically

6. Which header file is required for file handling in C++?

A. iostream

B. file.h

C. fstream

D. stdio.h

7. Which C++ stream is used for reading from a file?

A. ofstream

B. fstream

C. ifstream

D. cin

8. What does the eof() function check for in file handling?

A. End of line

B. End of file

C. File not found

D. File open failure

9. What mode is used to append data to a file in C++?

A. ios::in

B. ios::out

C. ios::trunc

D. ios::app

10. Which C++ stream allows both reading and writing to files?

A. fstream

B. ifstream

C. ofstream

D. ofstream with ios::in

Short Questions:

1. What is exception handling in C++?
2. Name the three main keywords used in exception handling in C++.
3. How do you throw an exception in C++? Give an example.
4. What is the purpose of the catch block in exception handling?
5. What happens if an exception is thrown but not caught in a C++ program?

6. What is the use of catch(...) in C++?
7. What is the try block used for in exception handling?
8. What is the purpose of the fstream header in C++?
9. Differentiate between ifstream, ofstream, and fstream.
10. How do you open a file for both reading and writing in C++?
11. What does the eof() function do in file handling?
12. How can you check if a file was opened successfully in C++?

Long Questions:

1. Explain the concept of exception handling in C++. Why is it important in object-oriented programming? Provide a simple example.
2. Describe the use and flow of try, throw, and catch blocks in C++. How do they work together to handle exceptions?
3. What are the advantages of using exception handling over traditional error handling methods in C++?
4. Write a C++ program that demonstrates exception handling using custom exception classes. Explain each part of the code.
5. What is the role of catch(...) in exception handling? When and why would you use it?
6. Discuss how multiple catch blocks can be used to handle different types of exceptions. Provide an example.
7. Explain how exception handling can be used to make programs more robust and maintainable. Give a real-world scenario.
8. What is file handling in C++? Explain how ifstream, ofstream, and fstream classes are used to perform file I/O operations.
9. Write a C++ program to open a file, read its contents, and handle any errors that may occur during file operations.
10. Explain different file modes available in C++ file handling, such as ios::in, ios::out, ios::app, ios::binary, and ios::trunc.
11. How can exception handling be integrated with file handling in C++ to create safer file operations? Illustrate with an example.
12. Discuss the common errors that may occur during file handling in C++. How can these errors be detected and handled effectively?

References

Chapter 1: Introduction to Object-Oriented Programming

1. Eckel, B. (2006). Thinking in Java (4th ed.). Prentice Hall.
2. Meyer, B. (1997). Object-oriented software construction (2nd ed.). Prentice Hall.
3. Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship. Prentice Hall.
4. Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley Professional.
5. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional.

Chapter 2: Classes and Objects

1. Horstmann, C. S. (2019). Core Java, Volume I: Fundamentals (11th ed.). Pearson.
2. Sierra, K., & Bates, B. (2005). Head First Java (2nd ed.). O'Reilly Media.
3. Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
4. Fowler, M. (2018). Refactoring: Improving the design of existing code (2nd ed.). Addison-Wesley Professional.
5. Lippman, S. B., Lajoie, J., & Moo, B. E. (2012). C++ primer (5th ed.). Addison-Wesley Professional.

Chapter 3: Inheritance and Polymorphism

1. Liskov, B., & Guttag, J. (2000). Program development in Java: Abstraction, specification, and object-oriented design. Addison-Wesley Professional.
2. Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley Professional.
3. Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. A. (2007). Object-oriented analysis and design with applications (3rd ed.). Addison-Wesley Professional.
4. Booch, G. (1994). Object-oriented analysis and design with applications (2nd ed.). Addison-Wesley Professional.

5. Budd, T. (2002). An introduction to object-oriented programming (3rd ed.). Addison-Wesley.

Chapter 4: Abstract Classes and Interfaces

1. Freeman, E., & Robson, E. (2014). Head First design patterns. O'Reilly Media.
2. Hunt, A., & Thomas, D. (2019). The pragmatic programmer: Your journey to mastery (20th anniversary ed.). Addison-Wesley Professional.
3. Lasater, C. G. (2006). Design patterns. Jones & Bartlett Learning.
4. McConnell, S. (2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.
5. McLaughlin, B. D., Pollice, G., & West, D. (2006). Head First object-oriented analysis and design. O'Reilly Media.

Chapter 5: Exception Handling and Multithreading

1. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java concurrency in practice. Addison-Wesley Professional.
2. Lea, D. (1999). Concurrent programming in Java: Design principles and patterns (2nd ed.). Addison-Wesley Professional.
3. Williams, A. (2019). C++ concurrency in action (2nd ed.). Manning Publications.
4. Oaks, S. (2014). Java performance: The definitive guide. O'Reilly Media.
5. Marlow, S. (2013). Parallel and concurrent programming in Haskell. O'Reilly Media.