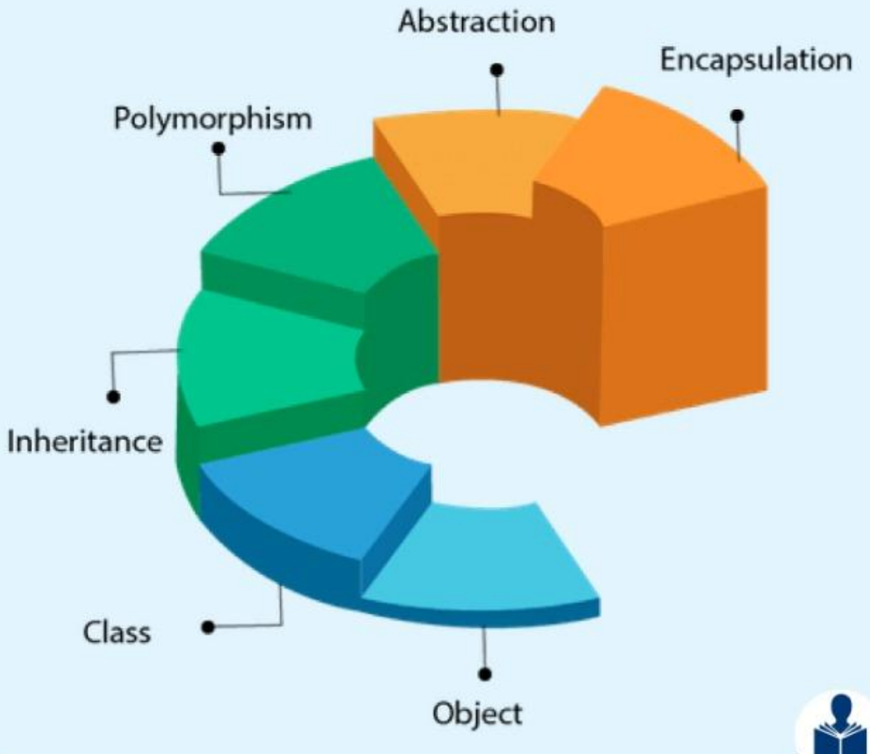
MATS UNIVERSITY
NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR DISTANCE & ONLINE 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 DSC-101-T

Object Oriented Programming Concepts

COURSE DEVELOPMENT EXPERT COMMITTEE

COURSE COORDINATOR

Prof. (Dr.) Bhavana Narain Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh.

COURSE PREPARATION

Prof. (Dr.) Bhavana Narain Professor and Ms. Arifa Khan, Assistant Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh.

March, 2025
ISBN: 978-93-49916-99-9

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 Blocks, each covering fundamental aspects of OOP using C++.

Block 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.

Block 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 Block 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.

Block 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.

Block 4: Operator Overloading and Type Conversion

Students explore how operators can be customized to work with user-defined data types. The Block 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.

Block 5: Exception Handling and File Handling

Students learn the skills to develop robust and error-free applications. This Block 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.

Block 1: Programming Paradigms

Unit 1: Programming Language Concepts

Structure

1.1 Introduction

1.2 Learning Outcomes

1.3 Syntax and semantics of programming concepts

1.4 High Level vs Low level Language

1.5 Compilation and Interpretation

1.6 Object-Oriented vs. Procedural Programming

1.7 Standard Libraries and APIs

1.8 Summary

1.9 Exercises

1.10 References and Suggested Readings

1.1: Introduction

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.

1.2: Learning Outcomes

After completing this topic, students will be able to:

- Understand the **syntax and semantics** of programming languages and how they define the structure and meaning of code.
- Differentiate between **high-level** and **low-level** programming languages based on abstraction, usability, and hardware control.
- Explain the **process of compilation and interpretation**, including how source code is converted into executable form.

- Compare and contrast **object-oriented programming (OOP)** and **procedural programming**, understanding their principles and use cases.
- Identify and apply **standard libraries** and **APIs** to enhance program functionality and reusability.
- Appreciate the role of **language translators (compiler/interpreter)** in program execution.
- Utilize **built-in libraries and APIs** effectively to simplify programming tasks.
- Build a strong conceptual foundation for **advanced programming concepts** and real-world application development.

1.3: Syntax and semantics of programming concepts

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.

1.4: High Level vs Low level Language

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;

1.5: 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
```

1.6: 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;
```

}

1.2 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

1.7: 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.

1. Standard Libraries

A library is a collection of pre-written code that we can reuse.

In C++, the Standard Library (STL) provides:

- Containers (e.g., vector, list, map)
- Algorithms (e.g., sort(), find(), reverse())
- Input/Output (iostream, fstream)
- String handling (string)
- Math functions (cmath)
- Utilities (utility, tuple, pair)

2. APIs (Application Programming Interfaces)

- An **API** is a set of **functions, classes, or methods** provided by a library or framework that lets you interact with it.

- The **C++ Standard Library itself is an API** (we call its functions, e.g., sort() or push_back() without knowing the internal code).
- External APIs also exist — like Windows API, OpenGL API, or database APIs.

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. They act as a bridge between human logic and machine instructions, allowing developers to express problem-solving ideas in a form that computers can understand.

Key Concepts in Programming Languages

1. Syntax and Semantics
    - Syntax refers to the rules that define how programs must be written (like grammar in English).
    - Semantics refers to the meaning of those statements (what the program actually does).
    - Example: In C++, int x = 10; is syntactically correct and semantically means "store 10 in integer variable x."
2. Typing Systems
    - Strongly typed vs. weakly typed → defines how strictly types are enforced.
    - Statically typed (C++, Java) → type checking at compile time.
    - Dynamically typed (Python, JavaScript) → type checking at runtime.
    - Understanding typing is important because it affects performance, safety, and flexibility.

Compilation and Interpretation

- Compiled languages (C, C++) → code is translated into machine language before execution → fast and efficient.
- Interpreted languages (Python, Ruby) → code is executed line by line by an interpreter → slower but flexible.
- C++ uses compilation which makes it highly suitable for performance-critical applications.

3. Programming Paradigms
   - A paradigm is a style of programming. Common ones include:
     - Procedural Programming (C): step-by-step instructions.
     - Object-Oriented Programming (C++, Java): code organized around objects.
     - Functional Programming (Haskell, modern C++ features like lambdas): emphasizes immutability and functions.
   - C++ supports multiple paradigms (procedural + OOP + generic programming).
4. Memory Management
   - Memory management is crucial for performance.
   - In C++, developers can use both manual memory control (new, delete) and automatic memory management (smart pointers, RAII).
   - This gives flexibility but also requires careful coding to avoid memory leaks.
5. Error Handling
   - Traditional programming languages used error codes.
   - Modern languages, including C++, support exception handling to manage runtime errors in a structured way.

All of these foundational concepts—syntax, typing systems, compilation, paradigms, memory management, and error handling—form the backbone of programming. Understanding them deeply is important before moving into advanced concepts. This knowledge sets the stage for **Object-Oriented Programming (OOP) in C++**, where these principles are combined to create structured, efficient, and scalable software solutions. OOP builds on this foundation by introducing classes, objects, encapsulation, inheritance, and polymorphism, making it easier to model real-world problems in software.

Programming languages serve as the foundation for software development, providing structured methods to write, execute, and manage code efficiently. They enable programmers to translate problem-solving strategies into a form that computers can process. By offering rules, structures, and built-in features, programming languages act as a bridge between human logic and machine instructions. Without them, it would be nearly impossible to communicate effectively with computers.

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.

CHECK YOUR PROGRESS:

1. Write one advantage and one disadvantage of low-level languages.

2.What is the difference between a library and an API?

1.8: Summary

Programming languages are formal systems that allow humans to communicate with computers to create software applications. They define rules (syntax) and meanings (semantics) to express logic, process data, and control hardware operations. Based on abstraction, languages are categorized as high-level (like C++, Python) and low-level (like Assembly). They can be compiled or interpreted, statically or dynamically typed, and follow different paradigms such as procedural, object-oriented, and functional programming. Understanding these core concepts helps programmers write efficient, readable, and maintainable code.

Memory management, standard libraries, and APIs play vital roles in modern programming. While languages like C and C++ require manual memory allocation, others like Python and Java manage it automatically through garbage collection. Standard libraries provide pre-built functions for input/output, math, and data structures, while APIs enable interaction with external systems. Together, these concepts form the foundation of software development and prepare learners to understand advanced topics such as Object-Oriented Programming (OOP) in C++, which builds upon these principles to create structured and reusable software solutions.

1.9: Exercises

Multiple Choice Questions:

1. Which of the following best defines a programming language?

A. A system for managing computer hardware

B. A set of formal instructions to communicate with a computer

C. A graphical tool for designing software

D. A hardware component

Ans: b)

2. What does *syntax* refer to in a programming language?

A. The meaning of code statements

B. The speed of program execution

C. The structure and rules of writing code

D. The memory allocation process

Ans: c)

3. Which of the following is an example of a high-level language?

A. Assembly

B. Machine language

C. C++

D. Binary

Ans: c)

4. Which type of language requires the code to be converted entirely into machine code before execution?

A. Interpreted language

B. Scripted language

C. Compiled language

D. Low-level language

Ans: c)

5. Object-Oriented Programming (OOP) primarily focuses on:

A. Sequential execution of statements

B. Data encapsulation and object interactions

C. Function-based logic only

D. Machine-level operations

Ans: b)

Descriptive Questions:

1. Define syntax and semantics in programming languages with an example.
2. Explain static typing and dynamic typing with examples.
3. Explain the main concepts of programming languages, including syntax, semantics, typing systems, and paradigms. Provide suitable examples.
4. What are compiled and interpreted languages? Compare their advantages and disadvantages with examples.
5. Differentiate between high-level and low-level programming languages.

1.10: References and Suggested Reading

- Eckel, B. (2006). Thinking in Java (4th ed.). Prentice Hall.
- Meyer, B. (1997). Object-oriented software construction (2nd ed.). Prentice Hall.
- Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship. Prentice Hall.

Block 1: Programming Paradigms

Unit 2: Types of Programming Language and its Application Area

Structure

2.1: Introduction

Programming languages serve as the primary medium of communication between humans and computers. They provide a structured way to write instructions that a computer can understand and execute to perform specific tasks. Over time, a wide variety of programming languages have been developed, each designed with unique features, syntax, and levels of abstraction to meet different computational needs.

The classification of programming languages is based on factors such as generation (machine, assembly, high-level, or fourth-generation languages), paradigm (procedural, object-oriented, functional, logic-based), and application area (web development, system programming, data analysis, artificial intelligence, etc.).

Understanding the different types of programming languages helps programmers choose the most suitable language for solving a particular problem efficiently. It also provides insight into how languages evolve to make programming easier, faster, and more efficient while supporting emerging technologies and platforms.

This unit explores the various types of programming languages, their distinct characteristics, and their real-world application areas across industries, forming the foundation for understanding how software systems are designed and developed.

2.2: Learning Outcomes

After completing this topic, students will be able to:

- Understand the concept and importance of programming languages in software development.

Notes

- Classify programming languages based on their generation (Machine, Assembly, High-level, and Fourth-generation languages).
- Differentiate between various programming paradigms such as procedural, object-oriented, functional, and logic-□ Analyze how language choice impacts software performance, scalability, and maintainability.
- Develop the ability to select an appropriate programming language based on the nature of the project and problem domain.

2.3: 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,

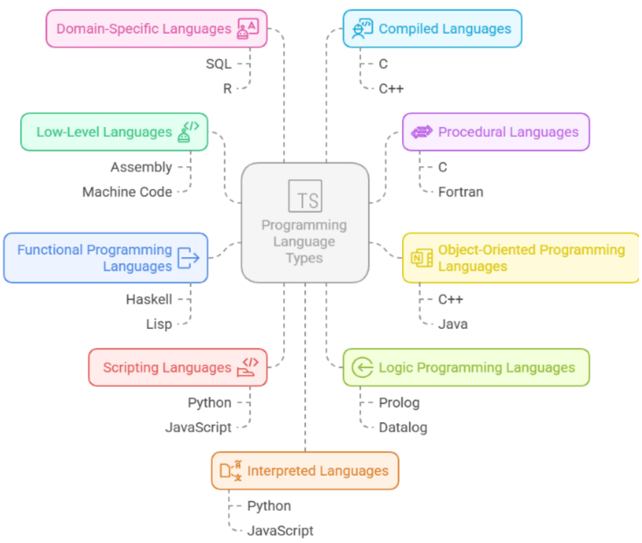


Figure 2.1: Programming language types and their application

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 2.1 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 2.2 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

Notes

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 for better performance.	C, C++, Java (JVM-based)	High-performance applications, Operating systems, Game engines
Interpreted Languages	Execute code line-by-line using an interpreter, making debugging easier.	Python, JavaScript, PHP	Web development, Scripting, Rapid prototyping

CHECK YOUR PROGRESS:

1. Define procedural programming. Give one example of a procedural language.

Notes

----

2. Describe the significance of domain-specific languages (DSLs) and their use in real-world applications.

2.4: Summary

Programming languages act as the medium through which humans communicate with computers to develop software. They can be broadly categorized into **low-level** and **high-level** languages based on their level of abstraction from hardware. Low-level languages, such as **machine language** and **assembly language**, are closer to the hardware and offer high efficiency but are difficult to code and maintain. High-level languages provide greater abstraction and human readability, making them more suitable for general-purpose programming. These are further classified into paradigms like **procedural**, **object-oriented**, **functional**, **scripting**, **logic**, and **domain-specific languages**, each designed to solve different computational problems effectively.

Procedural languages like **C** and **COBOL** follow a structured, step-by-step approach, while **object-oriented languages** such as **C++**, **Java**, and **Python** emphasize modularity through objects and classes. **Functional languages** like **Haskell** and **Erlang** focus on immutability and recursion, ideal for AI and data science. **Scripting languages** like **Python**, **PHP**, and **JavaScript** are used for automation and web development, while **logic programming languages** like **Prolog** rely on formal logic for problem-solving. **Domain-specific languages (DSLs)** such as **SQL**, **HTML**, and **R** are tailored to specialized tasks. Additionally, programming languages can be **compiled** (C, C++) or **interpreted** (Python, JavaScript), with each execution model having unique advantages in performance and flexibility. Understanding these types and their applications helps developers choose the right language for specific domains like web development, data analysis, system programming, and scientific computing.

2.5: Exercises

Multiple Choice Questions:

1. Which of the following is a low-level programming language?

A. A system for managing computer hardware

B. A set of formal instructions to communicate with a computer

C. A graphical tool for designing software

D. A hardware component

Ans: b)

2. Which programming paradigm focuses on step-by-step instructions using functions and loops?

A. Functional

B. Procedural

C. Object-Oriented

D. Logic

Ans: b)

3. In Object-Oriented Programming, which feature allows code reusability through inheritance?

A. Encapsulation

B. Abstraction

C. Polymorphism

D. Inheritance

Ans: d)

4. What does a logic programming language primarily use to define a program?

A. Objects and Classes

B. Rules and Facts

C. Loops and Conditions

D. Variables and Constants

Ans: b)

5. Which of the following is a Domain-Specific Language (DSL)?

A. Java

B. HTML

C. C++

D. Python

Ans: b)

Descriptive Questions:

1. Differentiate between low-level and high-level programming languages.
2. What are scripting languages? Give two examples.
3. Describe various types of programming languages and explain their specific application areas with examples.
4. Explain the main features of procedural, object-oriented, and functional programming languages. Provide suitable examples.
5. What are Domain-Specific Languages (DSLs)? Explain their importance with examples such as SQL, R, and HTML.

2.6: References and Suggested Reading:

- Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship. Prentice Hall.
- Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley Professional.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional.

Block 1: Programming Paradigms

Unit 3: File Creation, Compilation and Linking

Structure

3.1: Introduction

This unit introduces the fundamental building blocks of the C++ programming language, one of the most widely used and powerful languages in the software development world. It begins with understanding how a C++ source file is created, compiled, and linked to form an executable program. Learners will explore the features of C++ that make it an object-oriented and versatile language, such as data abstraction, encapsulation, inheritance, and polymorphism.

The unit also focuses on the basic structure of a C++ program, including the use of headers, functions, and main program blocks. It further covers data types, tokens, and control statements, which are essential for defining logic and managing program flow. Additionally, the concept of arrays is introduced to help learners manage and process collections of data efficiently.

By mastering these concepts, students will build a strong foundation for writing, compiling, and executing structured and object-oriented C++ programs, paving the way for advanced programming topics in later units.

3.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the process of source file creation, compilation, and linking in C++.
- Identify and explain the key features of C++, including object-oriented concepts and its advantages over C.
- Describe the basic structure of a C++ program, including headers, namespaces, main functions, and statements.
- Understand and use different data types in C++ such as int, float, char, double, and boolean.
- Recognize and apply various tokens in C++, including keywords, identifiers, constants, operators, and punctuators.
- Implement control statements (decision-making and looping constructs) such as if, switch, for, while, and do-while.
- Understand and apply arrays for storing and manipulating multiple values of the same data type.
- Develop simple to moderately complex C++ programs using appropriate data types, control structures, and arrays.
- Gain the ability to debug, compile, and link C++ programs effectively using an IDE or command-line environment.
- Build a foundation for learning object-oriented programming concepts and advanced C++ features in later units.

3.3: 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.

3.4: 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.

3.5: 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 defined outside of all functions and can be accessed from any part of 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';

Notes

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 3.1 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.

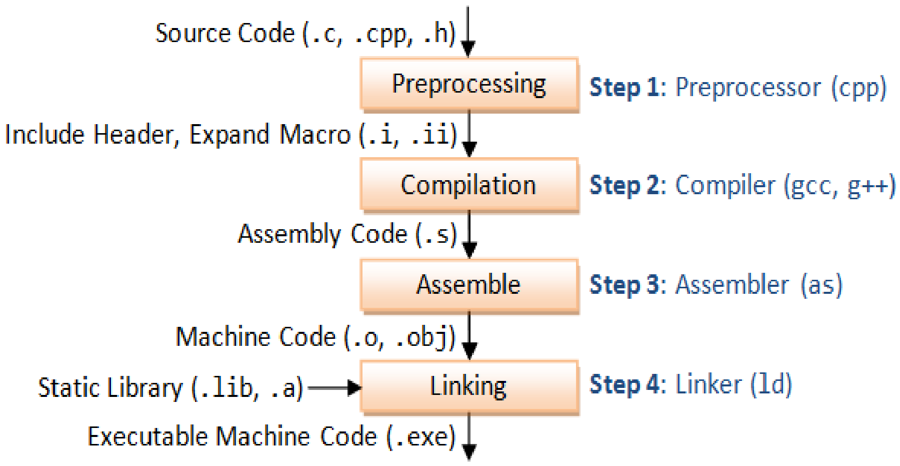


Figure 3.1 file creation, compilation to exe generation process
[Source: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html]

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.

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 Block 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.

Figure.3.2: features of C++

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. Simple

C++ is considered simple because it offers a structured approach to programming and provides a rich set of functions and libraries. Its syntax is largely influenced by the C language, making it easier for those with a C background to learn and use effectively.

2. 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.

3. Multi-Paradigm Support

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

Table 3.2 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.

4. 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.

5. 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.

6. 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.

7. Portability

Programs written in C++ can be compiled and run on different platforms without significant modification. This makes C++ a portable language, allowing developers to write cross-platform applications efficiently.

8. Powerful

C++ is a powerful language due to its ability to handle low-level programming, memory management, and its close association with system-level operations. It supports both procedural and object-oriented programming, giving developers extensive control over system resources.

9. Fast and Efficient

C++ is compiled directly into machine code, which makes execution fast and efficient. It is suitable for applications where performance and speed are critical, such as game development, real-time simulations, and operating systems.

10. Modularity

C++ encourages a modular approach to programming, where code can be organized into separate Blocks or functions. This improves readability, maintainability, and reusability of code.

11. Compiler Based

C++ is a compiler-based language, meaning that the source code must be compiled before it can be executed. This process helps catch syntax and semantic errors at compile time, increasing program stability and performance.

12. Huge Function Library

C++ offers an extensive standard library that includes a wide range of functions and classes for handling data structures, algorithms, input/output operations, and other utilities. These pre-defined functions help speed up development and reduce the need to write code from scratch.

13. Uses of Pointer

One of the unique features of C++ is its support for pointers, which allows direct memory access and manipulation. This makes it easier to work with dynamic memory allocation, arrays, and data structures like linked lists and trees.

These features make C++ a versatile and efficient language, widely used in software development fields such as system software, game engines, embedded systems, and high-performance applications.

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 3.3 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.

3.6: Data Types in C++

Data types specify the type of data that a variable can store. Whenever a variable is defined in C++, the compiler reserves a specific amount of memory for the variable according to its data type. with which it is declared as every data type requires a different amount of memory.

C++ supports a wide variety of data types, and the programmer can select the data type appropriate to the needs of the application.

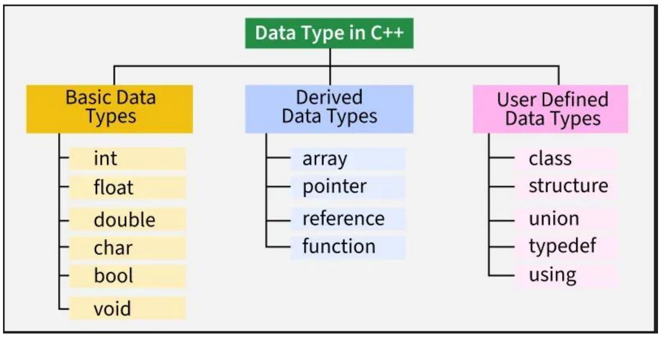


Figure 3.3: Datatypes in c++

Notes

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

Table 3.4 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

Built from fundamental types.

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

User-defined Data Types

These are the basic data types provided by the language.

- **Structure:** struct Student { string name; int age; };
- **Class:** class Car { public: string brand; };
- **Enumeration (enum):** enum Color { RED, GREEN, BLUE };

3.7: Tokens in C++

Tokens are the smallest Blocks 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 form the basic foundation of any programming language. Without operators, we cannot modify or manipulate the entities of programming languages and thereby cannot produce the desired results. C++ is very rich in built-in operators which we will discuss in detail in this tutorial.In C++ most of the operators are binary operators i.e. these operators require two operands to perform an operation. Few operators like ++ (increment) operator are the unary operator which means they operate on one operand only.

Notes

There is also a ternary operator in C++ called Conditional Operator which takes three operands. We will learn about this in detail in the later part of the tutorial.

Types of Operators: ***Operators in C++ are classified as shown below:***

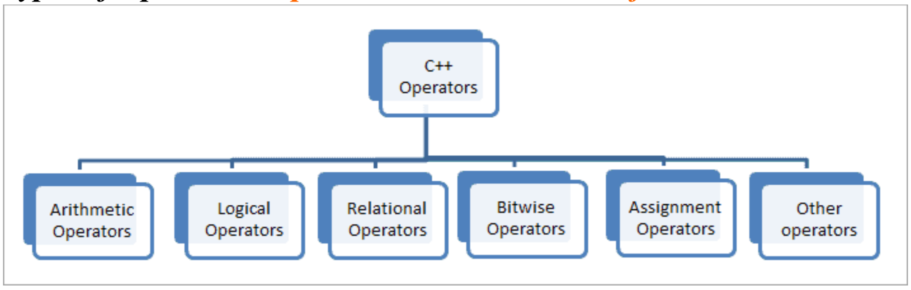


Figure 3.4: Types of operators in c++

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
```

3.8: 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 in C++ to **make decisions** and execute different code blocks based on certain conditions. They allow a program to follow different paths during execution depending on the logical outcome (true/false) of expressions.

1. *if Statement*

The if statement is the simplest form of a decision-making structure. It executes a block of code only when a specified condition is true. If the condition is false, the code inside the if block is skipped.

Syntax:

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

Example:

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

2. *if-else Statement*

The if-else statement is used when there are two possible outcomes. If the condition is true, one block of code is executed; otherwise, a different block is executed.

Syntax:

```
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
```

Notes

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

3. *if-else-if Ladder*

The if-else-if ladder is used when you need to check multiple conditions sequentially. The conditions are checked from top to bottom, and the first one that evaluates to true gets executed. If none are true, the else block is executed.

Syntax:

```
if (condition1) {
   // Code if condition1 is true
} else if (condition2) {
   // Code if condition2 is true
} else if (condition3) {
   // Code if condition3 is true
} else {
   // Code if none of the above conditions are true
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
   int marks = 75;

   if (marks >= 90) {
      cout << "Grade A";
   } else if (marks >= 75) {
      cout << "Grade B";
   } else if (marks >= 60) {
      cout << "Grade C";
   } else {
      cout << "Grade D";
   }
   return 0;
}
```

4. *switch Statement*

The switch statement is used when you want to select one block of code to execute from multiple options, based on the value of a single expression (usually an integer or character). Each option is labeled with a case, and the break statement is used to exit the switch block.

Syntax:

```
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, also known as iteration, is a core concept in programming that allows a set of instructions to be executed repeatedly as long as a specified condition holds true. In C++, there are three primary types of loops: for, while, and do-while. These loops help reduce code redundancy and make programs more efficient when repetitive tasks are involved.

1. *for Loop*

The for loop is used when the number of iterations is known in advance. It includes all loop control elements—initialization, condition check, and increment/decrement—in a single line, which makes it compact and easy to use.

Syntax:

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

- Initialization: Sets a loop control variable.
- Condition: Loop continues as long as this condition is true.
- Increment/Decrement: Updates the loop control variable after each iteration.

Example:

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

2. ***while Loop***

The while loop is used when the number of iterations is not known beforehand. It checks the condition before executing the loop body. If the condition is false initially, the loop body will not execute at all.

Syntax:

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

Example:

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

3. ***do-while Loop***

The do-while loop is similar to the while loop, but with one key difference: it executes the loop body at least once, regardless of whether the condition is true or false initially. The condition is checked after the loop body.

Syntax:

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

Example:

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

Jumping Statements (Control Transfer)

Jumping statements in C++ are used to alter the normal sequential flow of control in a program. These statements allow a program to exit loops, skip iterations, or jump to a specific label. They play a crucial role in implementing non-linear control flow, especially within loops and switch-case constructs.

1. ***break Statement***

The break statement is used to terminate the execution of a loop or switch statement prematurely. When the break statement is encountered, the control immediately exits the loop or switch and resumes with the next statement following the loop/switch block.

- It is commonly used:
- Inside for, while, or do-while loops to stop execution when a specific condition is met.
- Within switch statements to prevent fall-through behavior.

Example:

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

2. ***continue Statement***

The continue statement is used to skip the current iteration of a loop and jump to the beginning of the next iteration. It does not exit the loop but bypasses the remaining code in the current iteration when a specified condition is true.It is used when you want to ignore certain values or conditions temporarily, without stopping the entire loop.

Example:

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

3. ***goto Statement***

The goto statement is used to transfer control unconditionally to another part of the program marked with a label. It can be used to jump forward or backward in the code.

Notes

While goto can be useful in certain cases such as error handling in legacy systems, its use is generally discouraged in modern programming because it makes the control flow hard to understand and debug (often referred to as "spaghetti code").

Example:

goto label;

label:

cout << "Jumped here";

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

3.9: Arrays in C++

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.

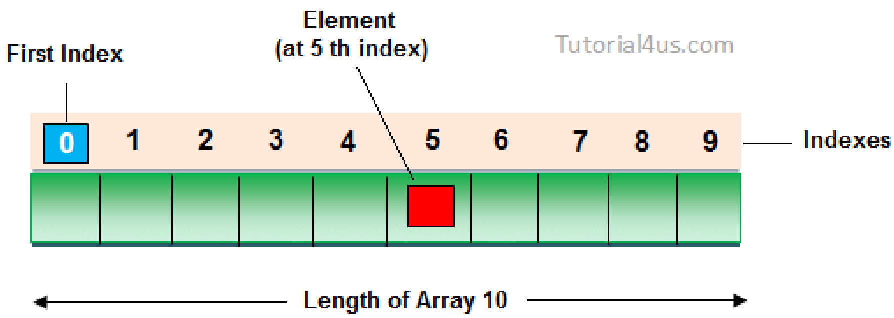


Figure 3.5: Arrays in c++

Advantage of array

Code Optimization: Less code is required, one variable can store numbers of value.

Easy to traverse data: By using array easily retrieve the data of array.

Easy to sort data: Easily short the data using swapping technique.

Random Access: With the help of array index you can randomly access any elements from array.

Dis-Advantage of array

Fixed Size: Whatever size, we define at the time of declaration of array, we can not change their size, if you need more memory in that time you can not increase memory size, and if you need less memory in that case also wastage of memory.

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:

Notes

```
#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 one-dimensional array is a linear structure that holds a fixed number of elements, all of the same data type, arranged in a single row. You can think of it as a list or a row of boxes, each containing a value and accessible by an index.

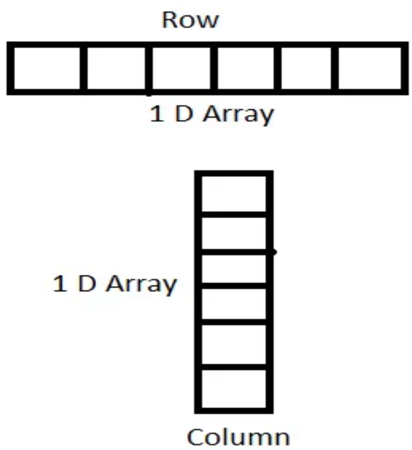


Figure 3.6: 1-Dimensional Array Representation

Example:

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

2. **Two-Dimensional Array (2D Array)**

A two-dimensional array is essentially an array of arrays. It is used to organize and store data in a matrix or table format, consisting of rows and columns. Each element is accessed using two indices: one for the row and one for the column.

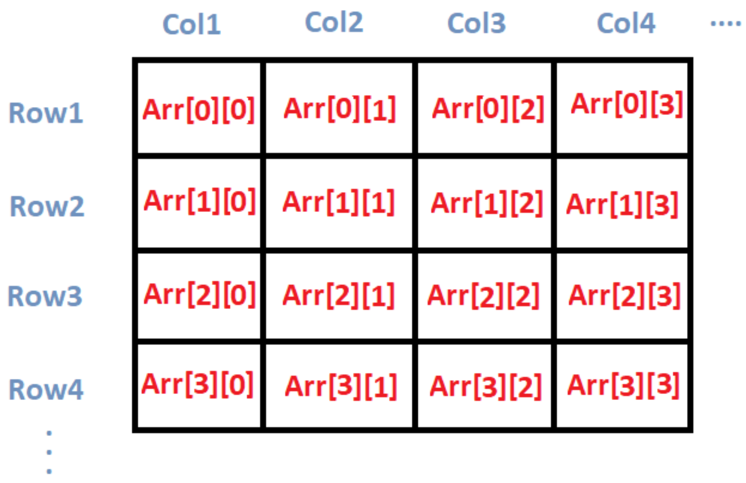


Figure 3.7: 2-Dimensional array

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;
    }
```

Notes

```
    return 0;
}
```

3. **Multi-Dimensional Array**

A multi-dimensional array is an extension of the concept of one-dimensional and two-dimensional arrays. It is a structure consisting of arrays within arrays (and so on), used to represent data in more than two dimensions. These are useful when dealing with complex data structures like 3D grids or higher-dimensional mathematical data.

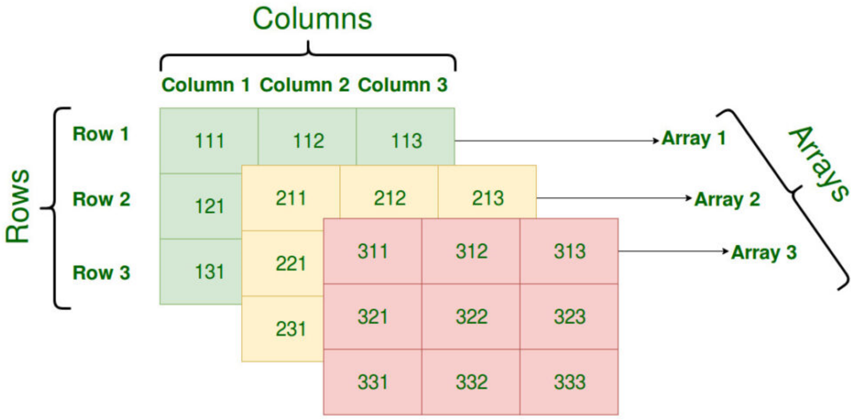


Figure 3.8: Multidimensional Array

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
```

CHECK YOUR PROGRESS:

1. Describe the importance of header files and namespaces in C++.

----

2. Explain any five major features of C++ in detail.

3.10: Summary

This Unit focuses on understanding how a C++ program is created, compiled, and executed. The process begins with writing source code in a file with a .cpp extension, which is then compiled to convert human-readable code into object code. The linker combines this object code with required libraries to generate an executable file that can be run on the system. C++ is a powerful, object-oriented, and multi-paradigm language that supports both low-level and high-level programming. Its key features include encapsulation, inheritance, polymorphism, operator overloading, memory management, and the use of the Standard Template Library (STL).

A C++ program generally consists of header files, namespaces, global declarations, the main function, and statements. Data in C++ is categorized into primary (int, float, char, double), derived (arrays, pointers), and user-defined types (struct, class, enum). The basic building blocks of C++ programs are tokens, which include keywords, identifiers, constants, operators, and punctuators. Operators perform various functions such as arithmetic, relational, logical, assignment, and bitwise operations. Control statements manage the program's flow using branching (if, switch), looping (for, while, do-while), and jumping (break, continue, goto) constructs. Arrays, which are collections of elements of the same data type, are used for efficient data storage and manipulation, though they have limitations like fixed size. Overall, this unit builds a foundation for writing, compiling, and

Notes

executing efficient C++ programs while understanding their structural and logical components.

3.11: Exercises

Multiple Choice Questions:

1. What is the extension of a C++ source file?

A. .exe

B. .cpp

C. .obj

D. .txt

Ans: b)

2. Which stage converts source code into object code?

A. Linking

B. Execution

C. Compilation

D. Debugging

Ans: c)

3. Which of the following is a feature of C++?

A. Machine language

B. Object-Oriented Programming

C. Interpreter based

D. Database query

Ans: b)

4. Which operator is used for conditional checking in C++?

A. &&

B. ?:

C. ++

D. ->

Ans: b)

5. Which loop is guaranteed to execute at least once?

A. for

B. while

C. do-while

D. switch

Ans: c)

Descriptive Questions:

1. What is compilation in C++?

2. What are control statements?
3. Describe the structure of a C++ program with an example.
4. Explain different types of operators in C++.
5. What are arrays? Explain types with examples.

3.12: References and Suggested Reading:

- Eckel, B. (2006). Thinking in Java (4th ed.). Prentice Hall.
- Meyer, B. (1997). Object-oriented software construction (2nd ed.). Prentice Hall.
- Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship. Prentice Hall.

Block 2: Class, Object, Constructor and Destructor

Unit 4: Object Oriented Programming Concepts and its Advantages

Structure

4.1: Introduction

This unit focuses on the core concepts of Object-Oriented Programming (OOP) — a modern programming paradigm that organizes software design around data, or objects, rather than functions and logic. OOP enhances code reusability, scalability, and maintainability, making it a fundamental approach in today's software development.

Learners will explore the main principles of OOP, including encapsulation, inheritance, polymorphism, and abstraction, which help in building modular and efficient programs. The unit also discusses the advantages of OOP over traditional procedural programming, emphasizing how OOP promotes better data security, code flexibility, and teamwork in large projects.

Furthermore, the unit highlights the practical applications and real-world impact of OOP, demonstrating how object-oriented principles are applied in developing modern technologies — from mobile applications and web systems to artificial intelligence and gaming software. By the end of this unit, students will understand how OOP principles shape the design and functionality of most modern programming languages like C++, Java, and Python.

4.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the core principles of Object-Oriented Programming (OOP).

- Explain the concepts of classes, objects, methods, and attributes in OOP.
- Describe and apply the four major pillars of OOP:
  - Encapsulation – Binding data and functions together.
  - Inheritance – Reusing and extending existing code.
  - Polymorphism – Performing a single action in different forms.
  - Abstraction – Hiding complex implementation details.
- Differentiate between OOP and procedural programming approaches.
- Identify the advantages of OOP, such as reusability, modularity, flexibility, and ease of maintenance.
- Understand how OOP helps in team-based software development by encouraging structured and reusable code.
- Explore the practical applications of OOP in various domains like software engineering, web development, mobile apps, AI, and game design.
- Analyze the real-world impact of OOP on modern programming practices and system design.

4.3: 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 Block, 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 accomplished through **method overloading**, where multiple methods share the same name but differ in parameters

Notes

within a class, and **method overriding**, where a derived class provides a specific implementation for a method inherited from its base class. **Abstraction** involves simplifying complex systems by designing classes that focus only on essential attributes and behaviors, while hiding irrelevant 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.

4.4: 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.

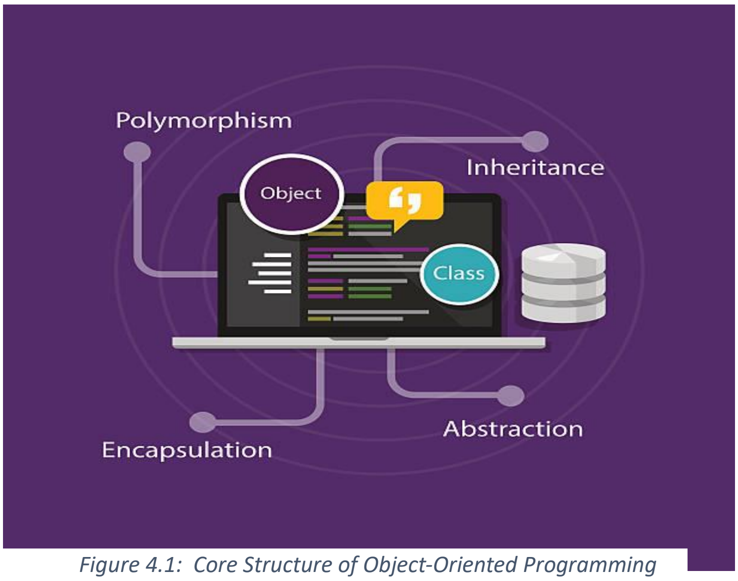


Figure 4.1: Core Structure of Object-Oriented Programming
[Source: https://www.istockphoto.com]

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.

4.5: 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

Notes

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.

CHECK YOUR PROGRESS:

1. Define the term "object" in OOP.

----

2. What is data abstraction in OOP?

4.6: Summary

This Unit focuses on the fundamental concepts, advantages, and practical applications of Object-Oriented Programming (OOP). OOP is a programming paradigm centered around the concept of **objects** and **classes**. A **class** serves as a blueprint defining the attributes (data) and methods (functions) that an **object** will possess. The key concepts of OOP include **encapsulation**, **inheritance**, **polymorphism**, and **abstraction**. Encapsulation binds data and functions together into a single unit, ensuring data protection through access specifiers like *public*, *private*, and *protected*. **Inheritance** allows a new class to derive properties and behaviors from an existing class, enabling code reusability and hierarchical organization. **Polymorphism** provides flexibility by allowing the same operation to behave differently on different objects, achieved through *method overloading* and *method overriding*. **Abstraction** simplifies complexity by focusing only on essential features, hiding unnecessary implementation details.

The advantages of OOP include **reusability**, **modularity**, **security**, **scalability**, and **flexibility**. It reduces code duplication, enhances problem-solving, and supports better maintenance by organizing complex systems into manageable components. In real-world applications, OOP is used in GUI development, game design, simulation, enterprise software, and scientific computation. Popular languages like **C++, Java, Python, and C#** are based on OOP principles, enabling developers to model real-world entities effectively. OOP has revolutionized modern programming by providing a structured, maintainable, and extensible approach to building complex and scalable software systems.

4.7: Exercises

Multiple Choice Questions:

1. **Which of the following best defines Object-Oriented Programming?**

a. A procedural approach to coding
b. A paradigm based on functions
c. A paradigm based on objects and classes
d. A database programming model

Ans: c)

2. Which concept of OOP binds data and functions together?

a. Inheritance
b. Polymorphism
c. Encapsulation
d. Abstraction

Ans: c)

3. Which of the following achieves runtime polymorphism?

a. Function overloading
b. Operator overloading
c. Method overriding
d. Inline functions

Ans: c)

4. **Which access specifier hides data from outside the class?**

a. Public

Notes

b. Private

c. Protected

d. Global

Ans: b)

5. Which OOP concept allows code reuse from existing classes?

a. Abstraction

b. Encapsulation

c. Inheritance

d. Polymorphism

Ans: c)

Descriptive Questions:

1. Define Object-Oriented Programming.
2. What is inheritance in OOP?
3. Explain how abstraction and polymorphism improve code flexibility.
4. Explain the core concepts of Object-Oriented Programming.
5. How does OOP differ from procedural programming?

4.8: References and Suggested Reading

- Horstmann, C. S. (2019). Core Java, Volume I: Fundamentals (11th ed.). Pearson.
- Sierra, K., & Bates, B. (2005). Head First Java (2nd ed.). O'Reilly Media.
- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.

Block 2: Class, Object, Constructor and Destructor

Unit 5: Object and Class

Structure

5.1: Introduction

This unit provides a detailed understanding of **Object-Oriented Programming (OOP) concepts in C++**, focusing on how real-world entities can be represented as objects within programs. It introduces learners to the **core OOP concepts**, including **objects, classes, constructors, and destructors**, which form the backbone of object-oriented design.

The unit explains how **classes** serve as blueprints for creating **objects**, and how **constructors** and **destructors** are used to initialize and clean up objects automatically. It also discusses how **objects can be passed as function arguments** to support modular and reusable program design. Additionally, learners will explore the concept of **arrays of objects**, which allows handling multiple instances of a class efficiently.

By mastering these concepts, students will gain the practical skills required to design, implement, and manage object-based programs in C++. This unit lays the foundation for advanced topics such as inheritance, polymorphism, and operator overloading in later chapters.

5.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the core concepts of Object-Oriented Programming (OOP) as implemented in C++.

- Define and use classes and objects to represent real-world entities in programs.
- Explain the concept and working of constructors and their types (default, parameterized, and copy constructors).
- Understand the role of destructors in resource management and automatic object cleanup.
- Pass objects as function arguments (by value, by reference, and by pointer) and understand how data is shared between functions and objects.
- Create and manipulate arrays of objects to store and process multiple instances of a class.
- Develop the ability to design modular and reusable code using OOP principles.
- Demonstrate understanding of object lifecycle — creation, usage, and destruction — in C++.
- Strengthen programming logic by applying OOP features to solve real-world problems.
- Build a foundation for advanced OOP concepts like inheritance, polymorphism, and operator overloading.

5.3: Objects and Classes in C++

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)
};
```

Notes

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;
}
```

5.4: 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

In C++, a constructor is a special member function of a class that is automatically called when an object is created. Its primary role is to initialize the object's data members.

General Rules:

- Constructor name is same as the class name.
- It has no return type (not even void).
- Can be overloaded to define multiple constructors for a class.

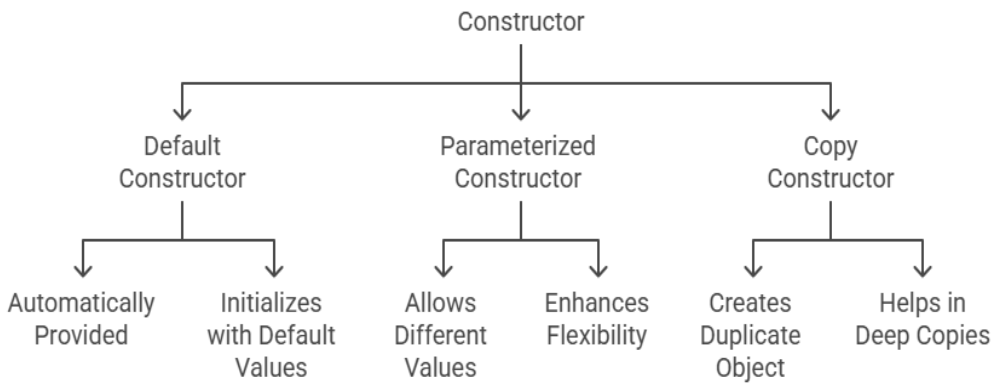


Figure 5.1: Types of Constructors

1. ***Default Constructor***

A default constructor is a constructor that takes no parameters. It is either provided by the compiler implicitly or defined explicitly by the user.

Features:

- Automatically provided by the compiler if no constructor is defined.
- Initializes the object with default values (0, null, false, etc.).
- Can also be defined manually.

Example:

```
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!

2. ***Parameterized Constructor***

A parameterized constructor is a constructor that takes arguments to initialize the object with specific values at the time of creation.

Features:

- Allows different objects to be initialized with different values.
- Enhances flexibility and control over object initialization.

Example:

```
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;
}
```

Example: Student Class with Parameterized Constructor

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

class Student {
    string name;
```

```
    int age;

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

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

int main() {
    // Creating objects using parameterized constructor
    Student s1("Shrishti", 20);
    Student s2("Sanjay", 22);

    // Display details
    s1.display();
    s2.display();

    return 0;
}
```

OUTPUT:

Name: Shrishti, Age: 20

Name: Sanjay, Age: 22

3. Copy Constructor

A copy constructor creates a new object by copying the values of another object of the same class. Java doesn't provide a built-in copy constructor like C++, but you can define it manually.

Features:

- Used to create a duplicate object.
- Helps in creating deep copies (manually) when needed.

Example:

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

Notes

```
class Student {
    int roll;
    string name;

public:
    // Parameterized Constructor
    Student(int r, string n) {
        roll = r;
        name = n;
    }
    // Copy Constructor
    Student(const Student &s) {
        roll = s.roll;
        name = s.name;
        cout << "Copy constructor called!" << endl;
    }
    void display() {
        cout << "Roll: " << roll << ", Name: " << name << endl;
    }
};
int main() {
    Student s1(101, "Ravi");    // Calls parameterized constructor
    Student s2 = s1;            // Calls copy constructor
    s1.display();
    s2.display();
    return 0;
}
```

Example: Copy Constructor

```
#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() {
    // Create object with parameterized constructor
    Student s1("Siddhi", 20);

    // Create object using copy constructor
    Student s2(s1);

    // Display both
    cout << "Original Object (s1): ";
    s1.display();

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

    return 0;
}
```

OUTPUT:

Original Object (s1): Name: Siddhi, Age: 20
Copied Object (s2): Name: Siddhi, Age: 20

5.5: 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 {
```

Notes

```
    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!

Example: Destructor in Action

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

class Student {
    string name;

public:
    // Constructor
    Student(string n) {
        name = n;
        cout << "Constructor called for " << name << endl;
    }

    // Destructor
    ~Student() {
        cout << "Destructor called for " << name << endl;
    }
};

int main() {
    // Creating objects
    Student s1("Siddhi");
    Student s2("Sanjay");
```

```
    cout << "Inside main function." << endl;

    return 0;
}
```

OUTPUT:

Constructor called for Siddhi

Constructor called for Sanjay

Inside main function.

Destructor called for Sanjay

Destructor called for Siddhi

5.6: 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;
}
```

5.7: 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 {
```

Notes

```
    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;
}
```

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

CHECK YOUR PROGRESS:

1. Explain the concept of an array of objects.

---

2. List the ways to pass an object to a function.

5.8: Summary

This Unit focuses on one of the most important aspects of C++ — **Objects and Classes**, the foundation of Object-Oriented Programming (OOP). A **class** is a user-defined data type that serves as a **blueprint** for creating objects. It contains **data members** (attributes) and **member functions** (methods) that define an object's behavior. An **object** is an instance of a class that uses these attributes and methods. Classes in C++ use **access specifiers** — *public*, *private*, and *protected* — to control the visibility and accessibility of data members.

The unit also explains **constructors** and **destructors**. A **constructor** is a special member function that initializes objects automatically when they are created, and it has the same name as the class. Constructors can be **default**, **parameterized**, or **copy constructors**. A **destructor**, on the other hand, is called automatically when an object is destroyed and is used to release memory or other resources.

Objects can be passed as **function arguments**, stored in **arrays**, and accessed using **pointers** for dynamic memory management. This enables efficient handling of large data sets and supports modular program design. Overall, this unit helps students understand how real-

Notes

world entities can be represented as objects in C++, improving program structure, readability, and reusability.

5.9: Exercises

Multiple Choice Questions:

1. What is a class in C++?

a. A pre-defined function

b. A built-in data type

c. A user-defined blueprint for creating objects

d. A library file

Ans: c)

2. Which of the following is automatically called when an object is created?

a. Destructor

b. Constructor

c. Copy function

d. Allocator

Ans: b)

3. Which symbol is used to define a destructor?

a. @

b. #

c. ~

d. $

Ans: c)

4. Which access specifier allows members to be accessed from anywhere?

a. Private

b. Protected

c. Public

d. Global

Ans: c)

5. Which of the following creates an object dynamically in C++?

a. Car car1;

b. Car* ptr = new Car;

c. Car() function;

d. Car.variable();

Ans: b)

Notes

Descriptive Questions:

1. What are constructors?
2. What are access specifiers?
3. Explain the concept of classes and objects in C++ with an example.
4. Describe different types of constructors in C++ with examples.
5. Explain how objects can be passed as function arguments.

5.10: References and Suggested Reading

- Horstmann, C. S. (2019). Core Java, Volume I: Fundamentals (11th ed.). Pearson.
- Sierra, K., & Bates, B. (2005). Head First Java (2nd ed.). O'Reilly Media.
- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.

Block 2: Class, Object, Constructor and Destructor

Unit 6: Member Function

Structure

6.1: Introduction

This unit explores the **advanced features of Object-Oriented Programming (OOP) in C++**, focusing on how data and functions interact within a class and how memory is managed dynamically. Learners will gain a deeper understanding of **member functions**, which define the behavior of class objects, and how **arrays can be declared within a class** to manage collections of data efficiently.

The unit also introduces important concepts such as **memory allocation of objects**, which explains how objects are created and stored in memory, and **friend functions**, which allow controlled access to private class members. Additionally, learners will study **local classes**, which are classes defined within functions and are useful for encapsulating function-specific data.

Further, the unit provides an in-depth explanation of **constructors**, including **copy constructors** and **dynamic constructors**, along with **dynamic initialization of objects**, which enables flexible and efficient object creation during runtime. These concepts help students write more powerful, modular, and memory-efficient C++ programs, preparing them for advanced application design.

6.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the concept and purpose of **member functions** and their role in defining class behavior.
- Implement **arrays within classes** to handle multiple data elements as part of an object.
- Explain the **memory allocation process** for objects in C++, including stack and heap memory usage.
- Understand and apply **friend functions** to access private or protected class members while maintaining encapsulation.
- Define and use **local classes** within functions to create context-specific object types.
- Understand the concept of **constructors** and their importance in object initialization.
- Differentiate between **default**, **parameterized**, **copy**, and **dynamic constructors**.
- Implement **dynamic initialization of objects** using constructors and memory management techniques (new and delete).
- Apply **copy constructors** to duplicate objects safely and efficiently.
- Demonstrate how dynamic memory allocation enhances **program flexibility and efficiency**.
- Develop object-oriented C++ programs using **advanced class features and memory management** techniques.

6.3: 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:

```
#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.

6.4: 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() {
```

Notes

```
        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;
```

Notes

```
    }
};
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.

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.

6.5: 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.

Notes

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.

Notes

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.

Notes

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;
```

Notes

}

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.

6.6: 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;
    }
};
```

Notes

```
// 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**.

6.7: 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
        }
    };
```

Notes

```
    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 {
```

Notes

```
    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;
}
```

Notes

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.

6.8: 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.

Notes

- 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;
```

Notes

```
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 6.1 Comparison of Constructor Types

Constructor Type	Definition	Usage Example
Parameterized Constructor	Initializes an object with specific values	Student s1("John", 20);

	passed as arguments.	
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.

6.9: 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) {
```

Notes

```
        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;
   }
```

Notes

```
};

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 6.2 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++.

Notes

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() {
```

Notes

```
    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;
}
```

Notes

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.

CHECK YOUR PROGRESS:

1. What is the purpose of a member function in a class?

----

2. What are the advantages of defining member functions inside a class?

6.10: Summary

In C++, **member functions** are the functions defined inside a class that operate on its data members. They help implement encapsulation by allowing controlled access to private data. Member functions can be defined either **inside the class** (inline) or **outside the class** using the

scope resolution operator (::). These functions perform various tasks such as setting or displaying data, performing calculations, or managing object behavior.

A special type of member function is the **constructor**, which is automatically called when an object is created. It initializes the object's data members and has the same name as the class. There are three types of constructors:

1. **Default Constructor** – initializes data with default values.
2. **Parameterized Constructor** – allows passing arguments during object creation.
3. **Copy Constructor –** creates a copy of another object.

Similarly, a **destructor** is a special member function that is automatically called when an object is destroyed. It has the same name as the class, preceded by a tilde (~), and is mainly used for releasing resources or memory cleanup.

C++ also supports **local classes**, which are classes declared inside a function. These classes can only be used within the function where they are defined. Local classes are useful for modular programming, temporary data structures, or when a class's functionality is needed only in a specific function.

Together, **member functions**, **constructors**, **destructors**, and **local classes** enhance **object control, code modularity, and memory safety**, making C++ a powerful and efficient object-oriented language.

6.11: Exercises

Multiple Choice Questions:

1. A member function that initializes objects automatically is called a:

A. Destructor

B. Constructor

C. Accessor

D. Mutator

Ans: b)

2. Which operator is used to define a member function outside the class?

A. .

B. ->

Notes

C. ::

D. #

Ans: c)

3. Which function is automatically invoked when an object goes out of scope?

A. Constructor

B. Destructor

C. Inline function

D. Static function

Ans: b)

4. What is a local class?

A. A class inside a namespace

B. A class declared inside a function

C. A subclass of another class

D. A class in a separate file

Ans: b)

5. Which of the following is true about destructors?

A. Can take parameters

B. Can be overloaded

C. Cannot take parameters

D Must be static

Ans: c)

Descriptive Questions:

1. What is a member function?
2. What is the purpose of a destructor?
3. Explain member functions and how they are defined inside and outside a class with an example.
4. Explain the concept and uses of local classes with an example.
5. Describe constructors and their types with examples.

6.12: References and Suggested Reading

- Horstmann, C. S. (2019). Core Java, Volume I: Fundamentals (11th ed.). Pearson.
- Sierra, K., & Bates, B. (2005). Head First Java (2nd ed.). O'Reilly Media.

- Deitel, P., & Deitel, H. (2017). Java: How to program (11th ed.). Pearson.
- Fowler, M. (2018). Refactoring: Improving the design of existing code (2nd ed.). Addison-Wesley Professional.
- Lippman, S. B., Lajoie, J., & Moo, B. E. (2012). C++ primer (5th ed.). Addison-Wesley Professional.

BLOCK 3: Operator Overloading and Inheritance

UNIT 7: Operator Overloading: Unary and Binary

Structure

7.1: Introduction

This unit introduces one of the most powerful features of C++ — **Operator Overloading**, which allows developers to redefine the behavior of existing operators for user-defined data types. By using operator overloading, programmers can make objects of a class operate just like basic data types, thereby improving code readability and reusability.

Learners will explore how **unary and binary operators** can be overloaded to perform customized operations on class objects. Unary operators (like ++, --, !) work on a single operand, while binary operators (like +, -, *, /) operate on two operands. This unit emphasizes both **member function** and **friend function** approaches to overloading operators, ensuring a strong understanding of the underlying mechanisms.

By mastering these concepts, students will be able to write more intuitive and efficient programs, creating objects that can interact naturally through overloaded operators — just like built-in types. This understanding also lays the foundation for **advanced C++ programming and object manipulation techniques**.

7.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the concept and purpose of **operator overloading** in C++.
- Explain how **operators can be redefined** to work with user-defined data types (classes and objects).

- Implement **unary operator overloading** (e.g., ++, --, !, -) using both **member functions** and **friend functions**.
- Implement **binary operator overloading** (e.g., +, -, *, /, ==) to perform operations between two objects.
- Understand the **syntax and rules** for overloading operators safely and correctly.
- Recognize the **limitations** of operator overloading (e.g., which operators cannot be overloaded).

7.3: 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.

7.4: 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:
```

Notes

```
    Counter() { value = 0; }
    s
    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

Example: Overloading Unary Minus –

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

class Number {
   int value;

public:
   // Constructor
   Number(int v) {
      value = v;
   }

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

   // Unary minus operator overloading
   Number operator-() {
      Number temp(-value);  // negate the value
      return temp;
   }
};

int main() {
   Number n1(10);
   Number n2 = -n1;  // Calls overloaded unary minus

   cout << "Original: ";
   n1.display();

   cout << "After Unary -: ";
   n2.display();

   return 0;
}
```

OUTPUT:

Original: Value: 10
After Unary -: Value: -10

7.5: 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();
```

Notes

```
    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

EXAMPLE: Overloading – Operator

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

class Number {
    int value;

public:
    Number(int v = 0) { value = v; }

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

    // Overload binary - operator
    Number operator-(Number n) {
        return Number(value - n.value);
    }
};

int main() {
    Number n1(50), n2(20);
    Number n3 = n1 - n2;

    cout << "n1: "; n1.display();
    cout << "n2: "; n2.display();
    cout << "n1 - n2: "; n3.display();

    return 0;
}
```

OUTPUT:

n1: Value: 50
n2: Value: 20
n1 - n2: Value: 30

POINTS:

1. Binary operators always involve two operands.
2. They can be member functions or friend functions.
3. Overloading allows objects to behave like primitive data types, making code readable and intuitive.
4. Common operators to overload: +, -, *, /, ==, <, >, !=, <=, >=.

CHECK YOUR PROGRESS:

1. Which operators cannot be overloaded?

----

2. Explain the **syntax of operator overloading** in C++.

7.6: Summary

Operator Overloading in C++ allows redefining the behavior of existing operators for user-defined data types (classes/objects). It makes object manipulation more intuitive and natural, similar to how operators work on primitive data types. Using the keyword **operator**, programmers can redefine operators like +, -, ++, --, *, ==, and others to perform custom operations on class objects.

There are two major types of operator overloading:

1. Unary Operator Overloading
2. Binary Operator Overloading

Operator overloading enhances **readability**, **code reusability**, and **object-oriented design**. However, not all operators can be overloaded (e.g., ::, sizeof, .). Proper use of overloading ensures that object operations are both meaningful and consistent with their real-world interpretations.

7.7: Exercises

Multiple Choice Questions:

1. Which keyword is used for operator overloading in C++?

A. overload

B. operator

C. function

D. redefine

Ans: b)

2. How many operands does a unary operator work on?

A. Two

B. One

C. Three

D. None

Ans: b)

3. What differentiates the prefix and postfix forms of operator++()?

A. Return type

B. Dummy integer parameter

C. Scope resolution

D. Constructor name

Ans: b)

4. In binary operator overloading, how many operands are involved?

A. One

B. Two

C. Three

D. None

Ans: b)

5. Which of the following operators cannot be overloaded?

A. +

B. =

C. ::

D. -

Ans: c)

Descriptive Questions:

1. What is the syntax of operator overloading?
2. How is postfix increment operator differentiated from prefix in C++?

3. Explain operator overloading in C++ with example with types.
4. Write a C++ program to overload both unary and binary operators in one class.
5. Differentiate between unary and binary operator overloading.

7.8: References and Suggested Reading

- Liskov, B., & Guttag, J. (2000). Program development in Java: Abstraction, specification, and object-oriented design. Addison-Wesley Professional.
- Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley Professional.
- 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.

Block 3: Operator Overloading and Inheritance

Unit 8: Operator Binary Operators using Friend Function

8.1: Introduction

This unit focuses on one of the most useful techniques in **C++ operator overloading** — the use of **friend functions** to overload **binary operators**. In general, operator overloading allows programmers to redefine the meaning of operators (like +, -, *, /) for user-defined types such as classes and objects. When two different objects of a class need to be operated on, a **friend function** is often used for overloading because it can access the **private and protected members** of both operands.

Unlike member functions, friend functions are **not part of the class scope**, but they are granted access to its private data by being declared with the friend keyword. This makes friend functions ideal for binary operator overloading when the **left-hand operand is not an object of the same class**, or when direct access to private members of multiple objects is required.

By studying this unit, learners will gain the ability to implement binary operator overloading using friend functions effectively, making object interactions more natural and expressive, similar to operations on built-in data types.

8.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the concept and purpose of **binary operator overloading** using **friend functions** in C++.
- Explain how **friend functions** allow access to **private and protected members** of a class.
- Implement binary operator overloading (e.g., +, -, *, /) using friend functions for class objects.

Notes

- Write programs that perform **operations between two different objects** using overloaded binary operators.
- Understand the **syntax and declaration** of a friend function within a class.

8.3: 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

Notes

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

Notes

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**.

CHECK YOUR PROGRESS:

1. Write the general syntax of a friend function

--
--
--

2. Can we use friend functions for unary operator overloading?

8.4: Summary

Operator overloading in C++ enables redefining existing operators to work with user-defined data types. When two different objects need to be used in an operation, we can use a friend function for operator overloading instead of a member function.

A friend function is a non-member function of a class but has access to its private and protected members. It is declared inside the class using the friend keyword and defined outside it. This feature is particularly useful for overloading binary operators, such as +, -, *, /, ==, !=, <, >, etc., which involve two operands.

In binary operator overloading using friend functions:

- The operator function takes two object parameters.
- It is declared as a friend in the class.
- It performs the desired operation (like multiplication, addition, or comparison) and returns the result.

8.5: Exercises

Multiple Choice Questions:

1. Which keyword is used to declare a friend function in C++?

A. Friend

B. Operator

C. Public

D. inline

Ans: a)

2. What is true about friend functions?

A. They are private members

B. They cannot access private members

C. They are not members but can access private data

D. They must return void

Ans: c)

3. How many parameters does a binary operator friend function take?

A. 0

B. 1

C. 2

D. 3

Ans: c)

4. Which of the following operators can be overloaded using friend functions?

A. +

B. *

C. = =

D. All of the above

Ans: d)

5. What is the output of the program where m1 = 4, m2 = 5 in multiplication example?

A. Result: 15

B. Result: 9

C. Result: 20

D. Compilation error

Ans: b)

Descriptive Questions:

1. What is a friend function?
2. Why use friend functions in operator overloading?
3. Explain binary operator overloading using friend functions with example.

Notes

4. Explain the concept of overloading comparison operators with an example.
5. Differentiate between member function and friend function in operator overloading.

8.6: References and Suggested Readings

- Liskov, B., & Guttag, J. (2000). Program development in Java: Abstraction, specification, and object-oriented design. Addison-Wesley Professional.
- Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley Professional.

Block 3: Operator Overloading and Inheritance

Unit 9: Rules of Overloading Operators, Type Conversion

Structure

9.1: Introduction

This unit focuses on **advanced Object-Oriented Programming (OOP) concepts in C++**, specifically **operator overloading, type conversion, and inheritance**. It begins by revisiting **operator overloading rules**, which define how and when operators can be redefined for user-defined types to make programs more intuitive and readable. Learners will also explore **type conversion**, the process of converting one data type into another—both implicitly (automatic) and explicitly (user-defined)—to ensure flexible and error-free data handling.

The unit further delves into the **core concept of inheritance**, which allows a class (derived class) to reuse properties and behaviors of another class (base class), promoting reusability and modular design. Learners will study various forms of inheritance and understand how **constructors in derived classes** manage initialization in inheritance hierarchies.

Additionally, this unit introduces **virtual base classes** and **abstract classes**, essential for solving ambiguity in multiple inheritance and enforcing polymorphism through abstract methods. Together, these topics form the foundation of advanced object-oriented design in C++, enabling the development of robust, reusable, and extensible applications.

9.2: Learning Outcomes

After completing this unit, learners will be able to:

- Understand the **concept and rules of operator overloading** in C++.
- Apply the correct **syntax and constraints** for overloading operators safely and efficiently.
- Explain **type conversion** and differentiate between **implicit, explicit, and user-defined conversions**.
- Implement **type conversion functions** in classes to convert one object type into another.
- Understand the concept of **inheritance** and its types (single, multiple, multilevel, hierarchical, and hybrid).
- Create and use **derived classes** that inherit data members and functions from base classes.
- Explain the purpose and use of **constructors in derived classes**, including the order of constructor and destructor execution.
- Understand and apply the concept of **virtual base classes** to resolve ambiguity in multiple inheritance.
- Describe and implement **abstract classes** and **pure virtual functions** to support polymorphism.
- Develop modular and reusable **C++ programs** applying inheritance, operator overloading, and type conversion techniques.

9.3: 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 Block 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

```
#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

9.4: 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.

Notes

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

9.5: Type Conversion in C++

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

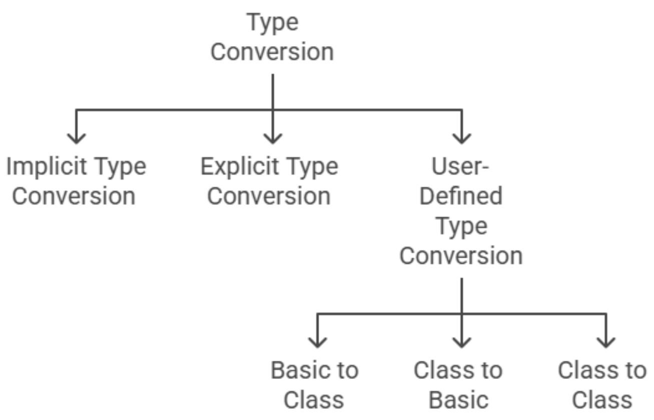


Figure 9.1: Type Conversion

1. **Implicit Type Conversion (Type Promotion)**
2. **Explicit Type Conversion (Type Casting)**
3. **User-Defined Type Conversion**

- Conversion from a basic data type to a class type
- Conversion from a class type to a basic data 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.

Notes

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>
```

Notes

```
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++**.

9.6: Inheritance and Derived Class 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 9.1 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

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 inheritance**: The public and protected members of the base class retain their access levels in the derived class.
- **protected inheritance**: The public and protected members of the base class become protected in the derived class.

Notes

- **private inheritance**: The public and protected members of the base class become private in the derived class.

Types of Inheritance

The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have 5 types of inheritances:

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

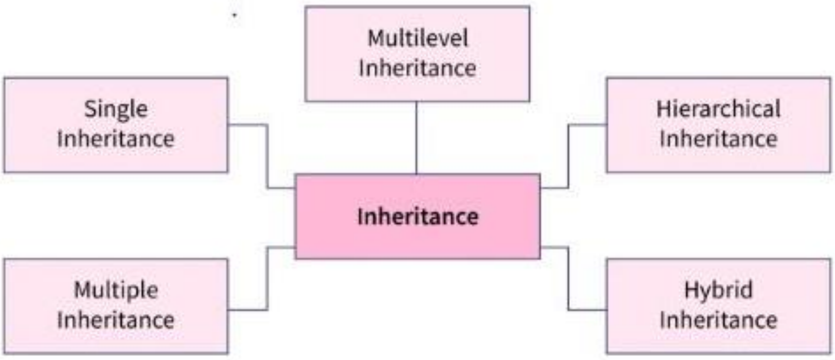


Figure 9.2: Types of Inheritance
[Source: https://medium.com]

1. **Single Inheritance**

In single inheritance, a class is allowed to inherit from only one class. i.e. one base class is inherited by one derived class only.

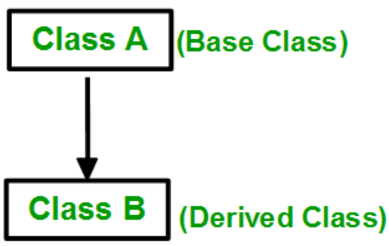


Figure 9.3: single Inheritance

Syntax:

class Parent {
public:

Notes

```
    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.

Example:

```
#include <iostream>
```

Notes

```
using namespace std;

// Base class
class Person {
protected:
    string name;
    int age;

public:
    void setData(string n, int a) {
        name = n;
        age = a;
    }

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

// Derived class
class Student : public Person {
    int rollNo;

public:
    void setRoll(int r) {
        rollNo = r;
    }

    void displayStudent() {
        displayPerson(); // Access base class function
        cout << "Roll No: " << rollNo << endl;
    }
};

int main() {
    Student s1;

    s1.setData("SIDHI", 20);  // Base class function
    s1.setRoll(101);          // Derived class function
```

```
    cout << "Student Details:" << endl;
    s1.displayStudent();      // Display all details

    return 0;
}
```

OUTPUT:

Student Details:
Name: SIDHI, Age: 20
Roll No: 101

2.Multilevel Inheritance

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

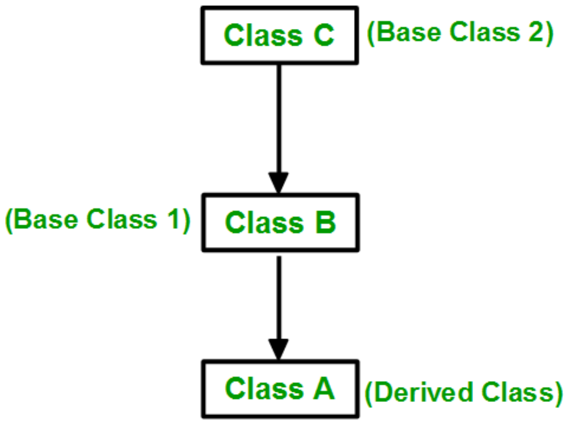


Figure 9.4: Multilevel Inheritance

Syntax:

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

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

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

Example:

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

Notes

```
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.

Example:

```
#include <iostream>

using namespace std;

// Base class

class Person {

protected:
```

```
    string name;
    int age;
public:
    void setPersonData(string n, int a) {
        name = n;
        age = a;
    }
    void displayPerson() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
// Derived class 1
class Student : public Person {
protected:
    int rollNo;

public:
    void setStudentData(int r) {
        rollNo = r;
    }
    void displayStudent() {
        cout << "Roll No: " << rollNo << endl;
    }
};

// Derived class 2
class Exam : public Student {
    int marks;
public:
    void setMarks(int m) {
        marks = m;
```

Notes

```
    }
    void displayExam() {
        displayPerson();   // From Base
        displayStudent();  // From Derived1
        cout << "Marks: " << marks << endl;
    }
};

int main() {
    Exam e1;

    e1.setPersonData("Sidhi", 20);
    e1.setStudentData(101);
    e1.setMarks(95);
    cout << "Exam Details:" << endl;
    e1.displayExam();

    return 0;
}
```

OUTPUT:

Exam Details:

Name: Sidhi, Age: 20

Roll No: 101

Marks: 95

3. Multiple Inheritance

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

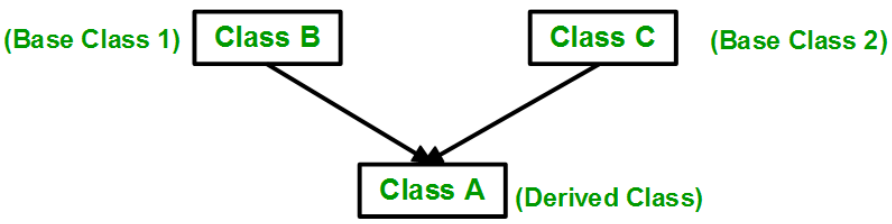


Figure 9.5: Multiple Inheritance

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.

EXAMPLE:

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

// Base class 1
class Person {
protected:
    string name;
    int age;

public:
    void setPersonData(string n, int a) {
        name = n;
        age = a;
    }

    void displayPerson() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
// Base class 2
class Address {
protected:
    string city;
    string country;

public:
    void setAddress(string c, string co) {
        city = c;
        country = co;
    }
```

Notes

```
    void displayAddress() {
        cout << "City: " << city << ", Country: " << country << endl;
    }
};

// Derived class
class Student : public Person, public Address {
    int rollNo;

public:
    void setRoll(int r) {
        rollNo = r;
    }

    void displayStudent() {
        displayPerson();   // from Person
        displayAddress();  // from Address
        cout << "Roll No: " << rollNo << endl;
    }
};

int main() {
    Student s1;

    s1.setPersonData("Sidhi", 20);
    s1.setAddress("Delhi", "India");
    s1.setRoll(101);

    cout << "Student Details:" << endl;
    s1.displayStudent();

    return 0;
}
```

OUTPUT:

Student Details:
Name: Arifa, Age: 20
City: Delhi, Country: India
Roll No: 101

4. Hierarchical Inheritance

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

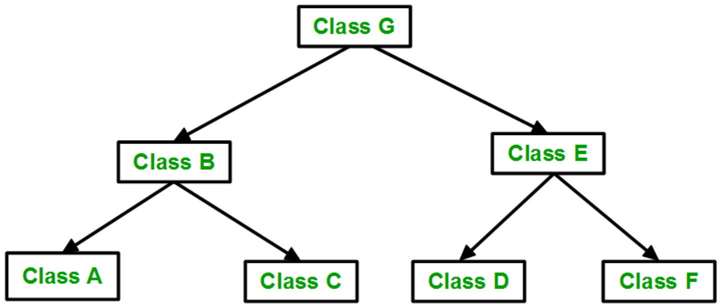


Figure 9.6: Hierarchical Inheritance

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).

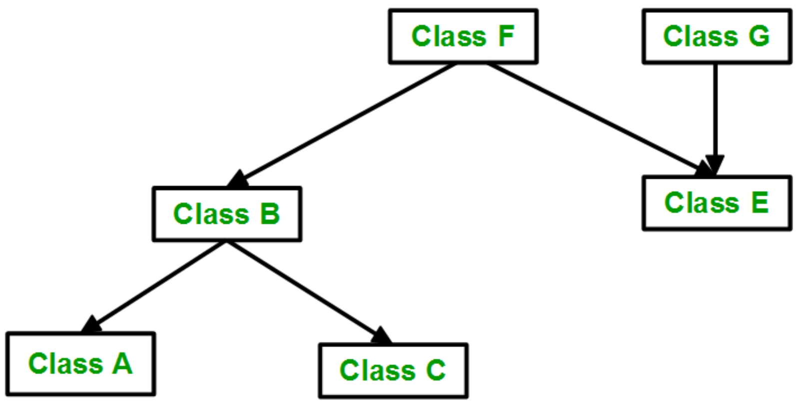


Figure 9.7: Hybrid Inheritance

Notes

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 Block 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)

9.7: 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
```

Notes

```
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;
```

Notes

```
    }
};

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 9.1 : 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++**.

9.8: 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.

Notes

The base class constructor is executed before the constructor of the derived class. This guarantees that all base class members are initialized correctly before the derived class introduces its own specific functionality.

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 {
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() {
```

Notes

```
    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>
```

Notes

```
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;
```

Notes

```
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

Notes

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 9.2: 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

Notes

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.

CHECK YOUR PROGRESS:

1. Define virtual base class.

---

2. Mention any two types of inheritance supported in C++.

9.9: Summary

This unit focuses on advanced object-oriented concepts in C++, including **operator overloading**, **type conversion**, **inheritance**, **virtual base classes**, **abstract classes**, and **constructors in derived classes**. Operator overloading allows redefining existing operators to work with user-defined types like objects while maintaining their original meaning. The rules ensure only predefined operators can be overloaded, with at least one operand being a user-defined type. Type conversion, on the other hand, refers to transforming one data type into another. It includes **implicit conversion (automatic type promotion)**, **explicit conversion (type casting)**, and **user-defined conversions** — such as conversion from basic type to class type, class to basic type, and class-to-class type.

Inheritance enables one class (derived) to acquire properties and methods of another (base), promoting **code reusability and modularity**. There are five types: **single, multilevel, multiple, hierarchical, and hybrid inheritance**. C++ also addresses the **Diamond Problem** using **virtual base classes**, ensuring only one instance of a base class is inherited even in complex hierarchies. **Abstract classes**, containing at least one pure virtual function, act as blueprints for other classes, enforcing implementation consistency across derived types.

Additionally, the unit covers **constructors in derived classes**, emphasizing that the base class constructor executes before the derived one to ensure proper initialization. **Member (nested) classes** allow a class to be defined inside another for better encapsulation and logical grouping. Together, these topics form the foundation for creating reusable, modular, and polymorphic object-oriented programs in C++.

9.10: Exercises

Multiple Choice Question:

1. Which of the following operators cannot be overloaded in C++?

A. +
B. =
C. ::
D. [ ]
Ans: c)

2. Type casting in which conversion is performed automatically by the compiler is called:

A. Explicit type conversion
B. Implicit type conversion
C. Manual conversion

D. Data coercion

Ans: b)

3. Which inheritance type allows a class to inherit from multiple base classes?

A. Single

B. Multilevel

C. Multiple

D. hybrid

Ans: c)

4. An abstract class must contain:

A. At least one static function

B. At least one pure virtual function

C. Only constructors

D. Only friend functions

Ans: b)

5. In C++, virtual base classes are primarily used to:

A. Increase memory usage

B. Solve diamond inheritance problems

C. Implement recursion

D. Prevent constructor execution

Ans: b)

Descriptive Questions:

1. What is the difference between implicit and explicit type conversion?
2. What is a pure virtual function?
3. Explain the rules of operator overloading in C++ with examples.
4. Differentiate between virtual base classes and abstract classes.
5. What are the different types of inheritance in C++? Explain each with examples.

9.11: References and Suggested Readings

- Liskov, B., & Guttag, J. (2000). Program development in Java: Abstraction, specification, and object-oriented design. Addison-Wesley Professional.
- Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley Professional.

Notes

- 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.
- Booch, G. (1994). Object-oriented analysis and design with applications (2nd ed.). Addison-Wesley Professional.

Block 4: Pointer, Virtual Function and Polymorphism

Unit 10: Pointers

Structure

10.1: Introduction

In Object-Oriented Programming (OOP) using C++, pointers play a vital role in accessing and manipulating objects dynamically. When working with inheritance, pointers can also be used to refer to objects of derived classes through base class pointers.

A pointer to a derived class is a pointer variable that can hold the address of a derived class object. However, due to the inheritance relationship, a base class pointer can also point to a derived class object, but the reverse is not true.

This feature allows runtime polymorphism when base class pointers are used to call virtual functions of derived classes. It provides flexibility in designing programs that can handle multiple object types through a single interface.

10.2: Learning Outcomes

After completing this unit, students will be able to:

1. Understand the concept of pointers in the context of inheritance.
2. Explain how a base class pointer can point to a derived class object.
3. Differentiate between pointer to base class and pointer to derived class.
4. Use base class pointers to access members of derived classes effectively.
5. Apply the concept of virtual functions to achieve runtime polymorphism using pointers.

6. Demonstrate object-oriented principles such as inheritance and polymorphism through pointer relationships.
7. Identify the limitations of base class pointers in accessing derived class members that are not part of the base class.

10.3: Pointers in C++

A pointer is a special variable that contains the address of another variable in memory. In C++, pointers are a powerful tool used for dynamic memory management, direct data manipulation, and implementing object-oriented programming concepts effectively.

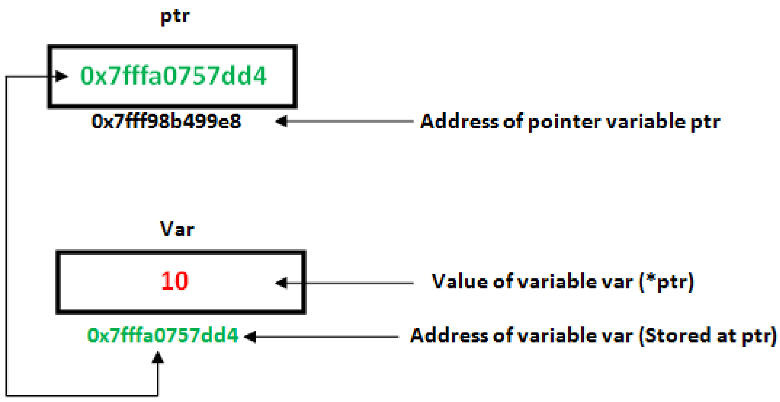


Figure 10.1: pointers in c++

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;
```

Figure 19: Concept of Pointers in OOP'S [Source https://www.scholarhat.com]

```
    cout << "Address of num: " << &num << endl;
    cout << "Value stored in pointer ptr: " << ptr << endl;
    cout << "Value accessed using pointer: " << *ptr << endl; // Dereferencing

    return 0;
}
```

Notes

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.

Notes

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

Notes

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 Block, 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 are essential for managing objects dynamically, especially in the context of inheritance. To achieve polymorphism, it is common to use pointers to base and derived classes. A base class pointer can point to a derived class object, allowing access to the base class members and enabling dynamic binding of overridden functions in the derived class..

10.4: Concept of Pointer to Derived Class

A base class pointer can store the address of a derived class object. However, when accessed through this pointer, only the base class members are accessible—unless virtual functions are used, which allow access to the overridden functions in the derived class.

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).

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**.

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.

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>
```

Notes

```
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() {
```

Notes

```
        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.

Table 10.1 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.

Notes

This topic is fundamental in **object-oriented programming (OOP)** and is widely used in designing **reusable and flexible software architectures**.

CHECK YOUR PROGRESS:

1. Mention one difference between a pointer and a reference in C++.

---

2. What is a pointer to an object?

10.5: Summary

Pointers are one of the most powerful and essential features in C++. They store the **memory address of another variable**, allowing direct access and manipulation of data in memory. This feature enhances performance and flexibility, making pointers crucial for dynamic memory allocation, data structures, and object-oriented programming. A pointer is declared using the * symbol, e.g., int *ptr;. Dereferencing (*ptr) allows access to the value stored at the pointed address. C++ also supports **pointers to objects**, enabling dynamic object management using new and delete. These are particularly useful for runtime memory handling and implementing concepts like **polymorphism**.

The **this pointer** is an implicit pointer available in all non-static member functions of a class, representing the address of the calling object. It resolves naming conflicts between local and member variables and allows **method chaining** and **operator overloading**. C++ also allows returning objects using the this pointer, making function chaining like obj.setName("A").setAge(20) possible.

In inheritance, **pointers to derived classes** enable **runtime polymorphism**. A **base class pointer** can point to a derived class object, allowing access to base class members. When **virtual functions** are used, the derived class version is executed, implementing **dynamic binding**.
However, if a destructor is not virtual in the base class, deleting a

derived object through a base pointer can lead to **memory leaks**. Hence, **virtual destructors** ensure proper cleanup.

Pointers also support multiple inheritance scenarios and typecasting between base and derived classes, making them indispensable for flexible and reusable object-oriented software design.

10.6: Exercises

Multiple Choice Questions:

1. Which of the following is the correct syntax to declare a pointer in c++?

A. int ptr;
B. int *ptr;
C. int &ptr;
D. pointer int;
Ans: b)

2. What is the main purpose of the this pointer in C++?
A. To access static members
B. To hold the address of the current object
C. To call base class constructor
D. To declare a friend function
Ans: b)

3. Which operator is used to access class members using a pointer?
A. .
B. ::
C. ->
D. *
Ans: c)

4. What will happen if the base class destructor is not virtual and we delete a derived object using a base pointer?
A. Both destructors are called
B. Only base destructor is called (memory leak)
C. Only derived destructor is called
D. Compilation error
Ans: b)

5. Which feature does a virtual function support in C++?

Notes

A. Function overloading
B. Compile-time binding
C. Runtime polymorphism
D. Multiple inheritance
Ans: c)

Descriptive Questions:

1. Define a pointer in C++.
2. What is the purpose of the this pointer?
3. What is a pointer to an object? Explain with a suitable example
4. Explain pointer to derived class with and without virtual functions.
5. Discuss the role of virtual destructors and multiple inheritance in pointers.

10.7: References and Suggested Readings:

- Freeman, E., & Robson, E. (2014). Head First design patterns. O'Reilly Media.
- Hunt, A., & Thomas, D. (2019). The pragmatic programmer: Your journey to mastery (20th anniversary ed.). Addison-Wesley Professional.
- Lasater, C. G. (2006). Design patterns. Jones & Bartlett Learning.
- McConnell, S. (2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.

Block 4: Pointer, Virtual Function and Polymorphism

Unit 11: Virtual Function and Pure Virtual Function

Structure

11.1: Introduction

In C++, a **virtual function** is a member function in a base class that you can **override** in a derived class to achieve **runtime polymorphism**. When a function is declared as virtual in the base class and redefined in a derived class, C++ determines at runtime which version of the function to execute — this process is known as **dynamic binding** or **late binding**.

A **virtual function** allows a base class pointer to call derived class functions, ensuring that the correct function is executed based on the **actual object type** being pointed to, not the type of the pointer.

11.2: Learning Outcomes

After completing this unit, students will be able to:

1. Understand the concept of virtual functions and their need in OOP.
2. Explain how runtime polymorphism is achieved using virtual functions.
3. Differentiate between normal, virtual, and pure virtual functions.
4. Implement virtual and pure virtual functions in C++ programs.
5. Demonstrate late binding using base class pointers or references.
6. Recognize the importance of pure virtual functions in creating abstract classes.

7. Apply virtual functions to design flexible and extensible object-oriented systems.

11.3: Virtual Function and Pure Virtual Function in C++

In C++, polymorphism enables objects of various derived classes to be treated as instances of a common base class. This behavior is made possible through the use of virtual functions.

virtual functions

A virtual function, also known as a virtual method, is a member function declared in a base class that can be overridden by a derived class. When a derived class object is accessed through a pointer or reference to the base class, the virtual function ensures that the overridden method in the derived class is executed.

This mechanism allows the program to determine the appropriate function implementation at runtime, enabling **runtime polymorphism**. To achieve this, the function in the base class must be declared using the **virtual** keyword. The decision about which version of the function to invoke is made during runtime, not at compile time.

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.

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).**

Virtual Function Behavior

If a virtual function is not overridden in the derived class, calling it will execute the base class version by default.

```
#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();

11.4: Pure Virtual Function (Abstract Class)

In some cases, a base class cannot provide implementations for all its functions because the actual behavior is unknown or varies across derived classes. Such a class is known as an abstract class. For instance, consider a base class Shape. While we know every shape must be drawn, we cannot define a general draw() function in the base class since the drawing behavior depends on the specific shape. Similarly, an Animal class might declare a move() function without implementation, as each animal moves differently.

To handle such scenarios, **pure virtual functions** are used. A pure virtual function is a virtual function that must be overridden in any derived class; otherwise, the derived class also becomes abstract. It is declared by assigning = 0 in the function declaration. Although a pure virtual function can have an implementation in the base class, it still enforces the requirement that derived classes must provide their own version. also, objects of an abstract class **cannot** be instantiated directly.

- Functions 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
};
```

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;
}
```

Notes

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**.

Key Differences between Virtual and Pure Virtual Functions

Table 11.1

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).

Real-Life Example: Employee Salary Calculation

```
#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**.

CHECK YOUR PROGRESS:

1. What is the purpose of the **virtual keyword** in C++?

---

2. What are the **characteristics** and **limitations** of virtual functions?

11.5: Summary

Polymorphism is a fundamental concept in **object-oriented programming (OOP)** that allows objects of different classes to be treated uniformly. In C++, **virtual functions** and **pure virtual functions** enable this behavior, making runtime polymorphism possible.

A **virtual function** is a member function declared in a **base class** using the virtual keyword, allowing derived classes to **override** it. When a base class pointer or reference is used to call a virtual function, C++ determines at **runtime** which function implementation (base or derived) should execute. This process is known as **dynamic binding** or **late binding**. Without the virtual keyword, the decision would occur at compile time (static binding).

If a virtual function is not overridden in the derived class, the base class version is executed by default. The base class function can still be explicitly accessed using the **scope resolution operator (::)**.

A **pure virtual function** is a virtual function with **no implementation in the base class**, declared by assigning = 0. It defines an **interface** that all derived classes must implement. A class containing at least one pure virtual function is called an **abstract class**, and **objects of abstract classes cannot be created directly**. Pure virtual functions enforce that derived classes provide their own specific implementation of the declared function, enabling strict polymorphic behavior.

These concepts are widely used in real-world applications, such as when creating a **base class Shape** with derived classes like Circle and Rectangle—each providing its unique implementation of a common draw() method. Similarly, **abstract base classes** like Employee with subclasses FullTime and PartTime demonstrate dynamic function behavior and flexible system design.

Thus, virtual and pure virtual functions make C++ code **modular, reusable, and scalable**, supporting the essence of runtime polymorphism in OOP.

11.6: Exercises

Multiple Choice Questions:

1. Which keyword is used to declare a virtual function in C++?

A. abstract
B. virtual
C. dynamic
D. override
Ans: b)

2. The process of deciding which function to call at runtime is known as:
A. Static binding
B. Early binding
C. Dynamic binding
D. Function overloading
Ans: c)

3. Which of the following makes a class abstract in C++?
A. Having a constructor
B. Having a friend function
C. Having a pure virtual function
D. Having a static member
Ans: c)

4. A pure virtual function is declared using:
A. virtual void fun();
B. virtual void fun() const;
C. virtual void fun() = 0;
D. void fun() virtual;
Ans: c)

5. Which of the following statements is true about abstract classes?
A. Objects of abstract classes can be created directly.
B. They cannot contain any data members.
C. They can contain constructors and destructors.
D. They cannot be inherited.

Notes

Ans: c)

Descriptive Questions:

1. Define a virtual function.
2. What is meant by dynamic binding?
3. Explain the concept of virtual functions in C++ with an example.
4. What is the difference between virtual and pure virtual functions?
5. Discuss how virtual functions enable runtime polymorphism.

11.7: References and Suggested Readings:

- Freeman, E., & Robson, E. (2014). Head First design patterns. O'Reilly Media.
- Hunt, A., & Thomas, D. (2019). The pragmatic programmer: Your journey to mastery (20th anniversary ed.). Addison-Wesley Professional.
- Lasater, C. G. (2006). Design patterns. Jones & Bartlett Learning.
- McConnell, S. (2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.

Block 4: Pointer, Virtual Function and Polymorphism

Unit 12: Polymorphism: Compile Time, Run Time

Structure

12.1: Introduction

In C++, **polymorphism** is one of the key principles of Object-Oriented Programming that allows a single function, operator, or object to behave in multiple ways depending on the context. The two major forms of polymorphism are **compile-time polymorphism** and **runtime polymorphism**, achieved through **overloading** and **overriding** respectively.

Overloading allows multiple functions or operators to have the same name but perform different tasks based on their parameters — either by changing the number, type, or order of arguments. This is known as **compile-time polymorphism** because the decision about which function to call is made by the compiler during program compilation. It increases code readability and provides flexibility in using the same function name for similar operations.

Overriding, on the other hand, is a feature of **runtime polymorphism**. It occurs when a **derived class** provides a new definition for a function already defined in its **base class** with the same name and parameters. Function overriding is typically achieved using **virtual functions**, which allow the program to decide at runtime which version of the function to execute. This enables more dynamic and flexible behavior, allowing derived classes to customize or extend the functionality of base classes without altering their structure.

12.2: Learning Outcomes

After studying this unit, students will be able to:

1. Understand the concept of polymorphism and its importance in C++.

Notes

2. Explain the difference between compile-time and runtime polymorphism.
3. Describe the concept of function and operator overloading and how it enhances program flexibility.
4. Explain the concept of function overriding and its role in achieving runtime polymorphism.
5. Differentiate between overloading and overriding in terms of execution time and implementation.
6. Apply overloading and overriding in C++ programs to design reusable and efficient code.
7. Analyze how polymorphism supports the principles of abstraction and inheritance in Object-Oriented Programming.

12.3: Polymorphism in C++

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)**,

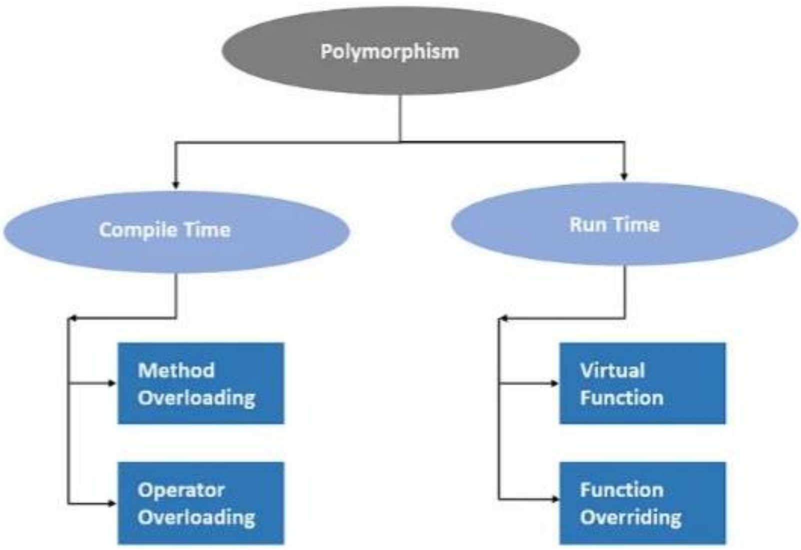


Figure 12.1: Types of polymorphism

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;
   }
};
```

Notes

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 12.1 Comparison: Compile-Time vs. Run-Time Polymorphism

Feature	**Compile-Time Polymorphism**	**Run-Time Polymorphism**
Binding Type	Early Binding (Static)	Late Binding (Dynamic)

Notes

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.

12.4: 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>
```

Notes

```
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**.

Notes

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 12.2 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.

Notes

CHECK YOUR PROGRESS:

1. Difference Between Function Overloading and Overriding

----

2. Explain Function Overloading Rules

12.5: Summary

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) that enables a single function, operator, or object to exhibit different behaviors depending on the context. The term “polymorphism” is derived from Greek words meaning “many forms.” In C++, polymorphism increases code flexibility, reusability, and maintainability. It is categorized into two types — **Compile-Time Polymorphism** and **Run-Time Polymorphism**. Compile-Time Polymorphism (also called *static binding*) occurs when the function call is resolved during compilation. It is achieved using **Function Overloading** and **Operator Overloading**, where multiple functions or operators behave differently based on the parameters or operands.

Run-Time Polymorphism (also called *dynamic binding*) is resolved during execution using **Function Overriding** and **Virtual Functions**. In this case, a derived class redefines a base class function, and the appropriate version is determined at runtime. Overall, polymorphism in C++ allows programmers to create flexible, extensible, and reusable code that can adapt to different scenarios dynamically. It helps achieve abstraction, reduces redundancy, and enhances software scalability.

12.6: Exercises

Multiple Choice Questions:

1. Which of the following best defines polymorphism in C++?

A. One name with multiple meanings

B. One function with one behavior

C. Same variable for different data types

D. Code duplication technique

Ans: a)

2. Which type of polymorphism is achieved through function overloading?

A. Run-time polymorphism

B. Compile-time polymorphism

C. Dynamic binding

D. Virtual function

Ans: b)

3. Which keyword is used to achieve runtime polymorphism in C++?

A. static

B. override

C. virtual

D. friend

Ans: c)

4. Function overriding occurs when:

A. Functions have the same name but different parameters in the same class.

B. Derived class provides its own version of base class function.

C. Two operators are used together.

D. A class has multiple constructors.

Ans: b)

5. Which of the following cannot be overloaded in C++?

A. +

B. ::

C. *

D. ==

Ans: b)

Descriptive Questions:

1. What is function overloading?
2. What is function overriding?
3. Explain polymorphism and its types in C++ with examples.
4. Explain function overloading and its rules with a suitable program.
5. What is function overriding? How does it help in runtime polymorphism?

12.7: References and Suggested Reading

- Freeman, E., & Robson, E. (2014). Head First design patterns. O'Reilly Media.
- Hunt, A., & Thomas, D. (2019). The pragmatic programmer: Your journey to mastery (20th anniversary ed.). Addison-Wesley Professional.
- Lasater, C. G. (2006). Design patterns. Jones & Bartlett Learning.
- McConnell, S. (2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.

Block 5: Exception Handling and File Handling

Unit 13: Stream Classes

Structure

13.1: Introduction

In C++, exception handling is an essential mechanism that allows a program to detect and manage unexpected runtime errors gracefully. Errors such as division by zero, file not found, invalid input, or accessing elements outside an array's range can cause a program to crash if not handled properly. Traditional methods like using if–else statements or error codes make the program lengthy, less readable, and difficult to maintain because the error-handling logic gets mixed with the main logic of the program.

C++ overcomes this limitation through a structured and object-oriented approach to error management using the try, throw, and catch keywords. With this mechanism, programmers can separate normal code from error-handling code, ensuring that the program remains clean, readable, and easy to maintain. Exception handling enables a function to throw an exception when an error occurs and allows another part of the program (often the caller) to catch and handle it.

13.2: Learning Outcomes

After completing this unit, students will be able to:

1. Understand the need and importance of exception handling in C++.
2. Explain how exception handling separates normal execution flow from error-handling code.
3. Identify the limitations of traditional error-handling methods using if–else statements.

Notes

4. Describe how try, throw, and catch blocks provide a structured approach to handling runtime errors.
5. Demonstrate how exceptions can be propagated and handled at different levels in a program.
6. Apply exception handling to manage runtime errors such as division by zero, file not found, and invalid input.
7. Analyze how exception handling improves program robustness, readability, and continuity.
8. Appreciate the role of exception handling in developing reliable and maintainable large-scale applications.

13.3: Exception Operations and File Handling

In C++, exceptions are unexpected problems or errors that occur while a program is running. For example, in a program that divides two numbers, dividing a number by 0 is an exception as it may lead to undefined errors.

The process of dealing with exceptions is known as exception handling. It allows programmers to make the program ready for any errors that may happen during execution and handle them gracefully so that it keeps running without errors.

try-catch Block

C++ provides an inbuilt feature for handling exceptions using try and catch block. It is an exception handling mechanism where the code that may cause an exception is placed inside the try block and the code that handles the exception is placed inside the catch block.

Syntax

```
try {
   // Code that might throw an exception
}
catch (ExceptionType e) {
   // exception handling code
}
```

When an exception occurs in try block, the execution stops, and the control goes to the matching catch block for handling.

Throwing Exceptions

Throwing exception means returning some kind of value that represent the exception from the try block. The matching catch block is found using the type of the thrown value. The throw keyword is used to throw the exception.

```
try {
   throw val
}
catch (ExceptionType e) {
   // exception handling code
}
```

There are three types of values that can be thrown as an exception:

1. Built-in Types
2. Standard Exceptions
3. Custom Exceptions
4. Throwing Built-in Types

Throwing built-in types is very simple but it does not provide any useful information. For example,

```
#include <bits/stdc++.h>
using namespace std;
int main() {
   int x = 7;
   try {
      if (x % 2 != 0) {

         // Throwing int
         throw -1;
      }
   }

   // Catching int
   catch (int e) {
      cout << "Exception Caught: " << e;
   }
   return 0;
}
```

Output

Notes

Exception Caught: -1

Here, we have to make decision based on the value thrown. It is not much different from handling errors using if else. There is a better technique available in C++. Instead of throwing simple values, we can throw objects of classes that contains the information about the nature of exception in themselves.

13.4: Throwing Standard Exceptions

Standard exceptions are the set of classes that represent different types of common exceptions. All these classes are defined inside <stdexcept> header file and mainly derived from std::exception class which act as the base class for inbuilt exceptions. The below image shows standard exceptions hierarchy in C++:

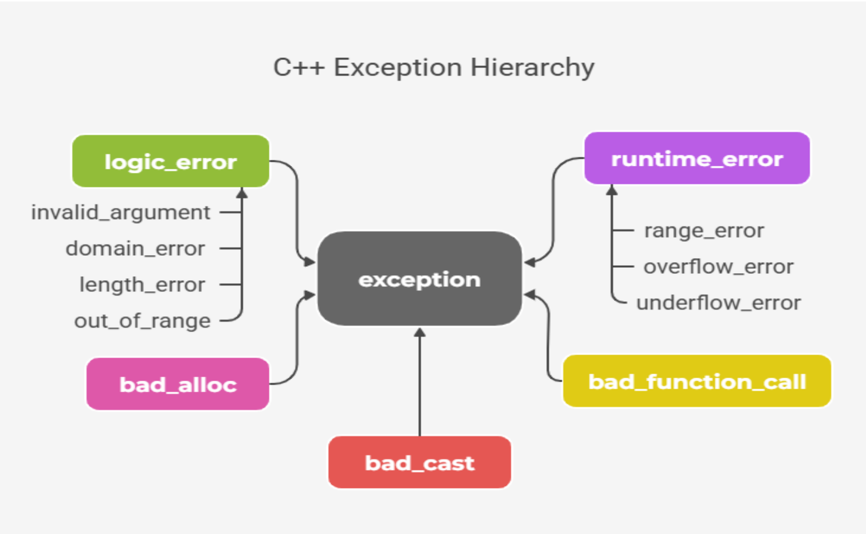


Figure 13.1: C++ Exception Hierarchy

These exceptions are thrown by C++ library components so we should know how to handle them. The what() method is present in every standard exception to provide information about the exception itself.

For example, the vector at() method throws an out_of_range exception when the element with given index does not exists.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3};
    try {
        // Accessing out of bound element
        v.at(10);
    }
```

```
    catch (out_of_range e) {
        cout << "Caught: " << e.what();
    }
    return 0;
}
```

Output

Caught: vector::_M_range_check: __n (which is 10) >= this->size() (which is 3)

We can also manually throw standard exceptions using throw statement.

Throwing Custom Exceptions

When the standard exceptions cannot satisfy our requirement, we can create a custom exception class. It is recommended to inherit standard exception in this class to provide seamless integrity with library components though, it is not compulsory.

Catching Exceptions

The catch block is used to catch the exceptions thrown in the try block. The catch block takes one argument, which should be of the same type as the exception.

```
catch (exceptionType e) {
    ...
}
```

Here, e is the name given to the exception. Statements inside the catch block will be executed if the exception of exceptionType is thrown in try block.

Catching Multiple Exceptions

There can be multiple catch blocks associated with a single try block to handle multiple types of exceptions. For example,

```
try {
    // Code that might throw an exception
}
catch (type1 e) {
    // executed when exception is of type1
```

Notes

```
}
catch (type2 e) {
   // executed when exception is of type2
}
catch (...) {
   // executed when no matching catch is found
}
```

In the above code, the last statement catch(...) creates a catch-all block which is executed when none of the above catch statements are matched. For example,

```
#include <bits/stdc++.h>
using namespace std;
int main() {
   // Code that might throw an exception
   try {
      int choice;
      cout << "Enter 1 for invalid argument, "
         << "2 for out of range: ";
      cin >> choice;

      if (choice == 1) {
         throw invalid_argument("Invalid argument");
      }
      else if (choice == 2) {
         throw out_of_range("Out of range");
      }
      else {
         throw "Unknown error";
      }
   }
   // executed when exception is of type invalid_argument
   catch (invalid_argument e) {
      cout << "Caught exception: " << e.what() << endl;
   }
   // executed when exception is of type out_of_range
   catch (out_of_range e) {
      cout << "Caught exception: " << e.what() << endl;
   }
      // executed when no matching catch is found
   catch (...) {
      cout << "Caught an unknown exception." << endl;
   }
   return 0;
}
```

Notes

Output 1

Enter 1 for invalid argument, 2 for out of range: 2

Caught exception: Out of range

Output 2

Enter 1 for invalid argument, 2 for out of range: 1

Caught exception: Invalid argument

Output 3

Enter 1 for invalid argument, 2 for out of range: 10

Caught an unknown exception.

EXAMPLE

```
#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main() {

    ifstream file;

    try {

        // Tell stream to throw exceptions on failure

        file.exceptions(ifstream::failbit | ifstream::badbit);

        cout << "Trying to open file..." << endl;

        // Try to open a file that does not exist

        file.open("nonexistent.txt");

        string line;

        while (getline(file, line)) {

            cout << line << endl;

        }
```

Notes

```
        file.close();
    }
    catch (const ifstream::failure &e) {
        cout << "Caught an ifstream::failure exception!" << endl;
        cout << "Error: " << e.what() << endl;
    }
    catch (const exception &e) {
        cout << "Caught a general exception!" << endl;
        cout << "Error: " << e.what() << endl;
    }
    cout << "Program continues safely..." << endl;
    return 0;
}
```

OUTPUT: (if file does NOT exist)

```
Trying to open file...
Caught an ifstream::failure exception!
Error: basic_ios::clear: iostream error
Program continues safely...
```

OUTPUT: (if file DOES exist and has text)

```
Hello
C++ Exceptions
Working Fine
```

Nested Try Catch Blocks

In C++, try-catch blocks can be defined inside another try or catch blocks. For example,

```
try {
   // Code...... throw e2
   try {
      // code..... throw e1
   }
   catch (eType1 e1) {
      // handling exception
   }
}
catch (eType e2) {
   // handling exception
}
```

Example: Nested Try-Catch

```
#include <iostream>

using namespace std;

int main() {

   try {

      cout << "Outer try block" << endl;

      try {

         cout << "Inner try block" << endl;

         int x = 10, y = 0;

         if (y == 0)

            throw "Division by Zero in Inner Block!";

      }

      catch (const char* msg) {

         cout << "Caught in Inner catch: " << msg << endl;

         // Rethrow exception to outer block
```

```
            throw;
        }
    }
    catch (const char* msg) {
        cout << "Caught in Outer catch: " << msg << endl;
    }
    cout << "Program continues..." << endl;
    return 0;
}
```

OUTPUT:

Outer try block

Inner try block

Caught in Inner catch: Division by Zero in Inner Block!

Caught in Outer catch: Division by Zero in Inner Block!

Program continues...

13.5: Why do we need Exception Handling in C++?

Errors or abnormal conditions can also be handled without exception handling, like it is done in C using conditional statements. But a exception handling provides the following advantages over traditional error handling:

Separation of Error Handling Code from Normal Code: There are always if-else conditions to handle errors in traditional error handling codes. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try and catch blocks, the code for error handling becomes separate from the normal flow.

Functions/Methods can handle only the exceptions They choose: A function can throw many exceptions but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

Grouping of Error Types: In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize them according to their types.

To handle runtime errors gracefully

- Errors like *file not found*, *division by zero*, *out of memory*, *array index out of bounds* can occur at runtime.
- Without exception handling, such errors may cause program crashes.
- With exceptions, we can **catch and handle** them properly, keeping the program safe.

To provide a structured way of handling errors

- C++ provides try, throw, and catch blocks.
- Errors can be “thrown” from one function and “caught” in another.
- This creates a **clean, centralized error-handling mechanism**.

To separate error-handling code from normal code

- Normally, you would check error codes (if, return -1, etc.) after every operation.
- This makes code messy and harder to read.
- Exception handling allows us to keep **normal logic separate from error logic**.

To ensure program continuity

- Even after an error, exception handling ensures the program doesn’t crash suddenly.
- It lets the programmer decide **how to recover** and continue.

To improve program reliability and robustness

- Programs that use exception handling are more **robust** (can deal with unexpected situations).
- This is very important for **large-scale applications** (e.g., banking, e-commerce, operating systems).

EXAMPLE: With Exception Handling

```
#include <iostream>

using namespace std;

int main() {

    int a = 10, b = 0;

    try {

        if (b == 0)

            throw "Division by zero not allowed!";

        cout << "Result: " << (a / b) << endl;

    }

    catch (const char* msg) {

        cout << "Exception caught: " << msg << endl;

    }

    cout << "Program continues safely..." << endl;

    return 0;

}
```

OUTPUT:

Exception caught: Division by zero not allowed!

Program continues safely...

CHECK YOUR PROGRESS:

1. Explain multiple catch blocks with an example.

----

2. What is the use of the throw keyword in C++?

13.6: Summary

Exception handling in C++ is a structured mechanism to manage runtime errors or unexpected events that disrupt normal program execution. Instead of abruptly terminating, exception handling allows the program to handle errors gracefully and continue running safely. The core keywords used in C++ exception handling are **try**, **throw**, and **catch**.

The **try block** contains code that might generate an error, while the **catch block** handles the thrown exception. The **throw keyword** is used to signal that an exceptional event has occurred. C++ supports throwing **built-in types**, **standard exceptions** (like out_of_range, invalid_argument, etc.), and **custom exceptions** (defined by users). Standard exceptions are part of the <stdexcept> library and derived from the std::exception class.

C++ also supports **multiple catch blocks** to handle different types of exceptions and a **catch-all block (catch(...))** to catch unknown exceptions. It even allows **nested try-catch blocks**, where one exception can be rethrown and handled by an outer block. Exception handling is especially useful in **file operations** where errors like “file not found” or “read/write failure” can occur.

13.7: Exercises

Multiple Choice Questions:

1. Which keyword is used to throw an exception in C++?

A. try

B. catch

C. throw

D. except

Notes

Ans: c)

2. The what() function in standard exceptions returns:

A. The name of the exception

B. A description of the error

C. Line number of error

D. None of these

Ans: b)

3. Which header file defines standard exception classes in C++?

A. <iostream>

B. <fstream>

C. <stdexcept>

D. <cstdlib>

Ans: c)

4. What does catch(...) mean?

A. It catches all exceptions not caught by earlier catch blocks.

B. It catches only integer exceptions.

C. It throws a new exception.

D. It terminates the program.

Ans: a)

5. Which function enables streams to throw exceptions automatically?

A. enable_exceptions()

B. set_exceptions()

C. exceptions()

D. throw_stream()

Ans: c

Descriptive Questions:

1. What is exception handling in C++?
2. What are standard exceptions?
3. Explain the syntax and working of try, catch, and throw in C++ with an example.
4. Explain multiple catch blocks with an example.
5. Why is exception handling better than traditional error handling?

Notes

13.8: References and Suggested Readings

- Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java concurrency in practice. Addison-Wesley Professional.
- Lea, D. (1999). Concurrent programming in Java: Design principles and patterns (2nd ed.). Addison-Wesley Professional.
- Williams, A. (2019). C++ concurrency in action (2nd ed.). Manning Publications.

Block 5: Exception Handling and File Handling

Unit 14: File Handling in OOPS

Structure

14.1: 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.

14.2: Learning Outcomes

After studying this unit, learners will be able to:

1. Understand the concept of Object-Oriented File Handling
    - Explain how file operations can be encapsulated within classes and objects.
2. Apply OOP principles in file operations
    - Demonstrate the use of Encapsulation, Inheritance, Polymorphism, and Abstraction in handling files.
3. Implement file handling using classes and objects in C++ and Python
    - Create and manipulate files through class-based methods like openFile(), readFile(), writeFile(), and closeFile().
4. Develop reusable and maintainable file-handling modules
5. Integrate exception handling with OOP-based file management
    - Use try–catch (in C++) or try–except (in Python) blocks within classes for robust file error management.

6. Differentiate between procedural and object-oriented approaches
    - Compare traditional file handling with OOP-based methods in terms of reusability, readability, and scalability.

14.3: 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. Encapsulation (data + functions together)

- In OOP, file data (like filename, file stream object) and functions (like openFile(), readFile(), writeFile(), closeFile()) can be bundled inside a class.
- This keeps file-related operations **organized and secure**.

Example: Instead of managing file streams everywhere, a FileManager class can handle everything.

2. Reusability (using classes again and again)

- Once a file handling class is written, it can be **reused** in multiple projects.
- Example: A LogFile class can be used in banking, e-commerce, or student record systems without rewriting the logic.

3. Abstraction (hide complex details)

- OOP hides the **complex syntax of streams** from the user.
- Example: Instead of writing ifstream, ofstream everywhere, the user can just call myFile.readData().
- The internal complexity stays hidden.

4. Inheritance (extend functionality)

- You can create a **base class** for general file handling.
- Then extend it (inheritance) for specific use cases like TextFile, BinaryFile, or LogFile.

Notes

5. **Polymorphism (one interface, many forms)**

- Same function name (save(), load()) can work differently for **text files**, **binary files**, or **JSON files**.
- Makes code **flexible and scalable**.

Example: OOP-Based File Handling

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

// Base Class
class FileHandler {
protected:
   string filename;
public:
   FileHandler(string fname) : filename(fname) {}
   virtual void write(string data) = 0;   // pure virtual (abstract)
   virtual void read() = 0;
};

// Derived Class for Text File
class TextFile : public FileHandler {
public:
   TextFile(string fname) : FileHandler(fname) {}

   void write(string data) override {
      ofstream fout(filename);
      fout << data;
      fout.close();
   }

   void read() override {
      ifstream fin(filename);
      string line;
      while (getline(fin, line)) {
         cout << line << endl;
      }
      fin.close();
   }
```

```
};

int main() {
    // Using OOP for file handling
    TextFile file("example.txt");

    file.write("Hello, OOP with File Handling!\nThis is a text file.");
    cout << "File written successfully.\n\nReading File:\n";
    file.read();

    return 0;
}
```

OUTPUT:

```
File written successfully.

Reading File:
Hello, OOP with File Handling!
This is a text file.
```

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()
```

14.4: Advantages of OOP-based File Handling

Given are the benefit of the OOP, based on file handling.

Table 14.1 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.*

.

14.5: *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.

CHECK YOUR PROGRESS:

1. What are best practices for implementing file handling in OOP.

----

2. Explain how exception handling is used in OOP-based file handling.

14.6: Summary

File handling in Object-Oriented Programming (OOP) combines the power of object-based design with data storage and retrieval operations. Traditional procedural file handling works well for simple tasks, but as programs grow larger, OOP provides a more organized, reusable, and scalable approach.

In OOP-based file handling, classes encapsulate file-related data (like file name and file stream) and the functions that operate on them (like openFile(), readFile(), writeFile(), and closeFile()). This encapsulation keeps code organized and ensures that data and functions remain logically connected.

Through **inheritance**, specialized classes (e.g., TextFileHandler, BinaryFileHandler) can be created from a base file-handling class to perform specific operations. **Polymorphism** allows the same function names to behave differently based on the file type — for example, save() for text vs. binary files. **Abstraction** hides the complexities of file stream operations, providing a cleaner interface to users.

By applying OOP principles like **encapsulation**, **inheritance**, **abstraction**, and **polymorphism**, developers can create flexible, reusable, and maintainable file-handling systems — such as file managers, log systems, and data storage modules — that are more secure, modular, and easier to extend.

14.7: Exercises

Multiple Choice Questions:

1. What is the main advantage of using OOP for file handling?

A. Faster execution

B. Less memory usage

C. Better organization and reusability

D. No need for file streams

Ans: c)

2. Which OOP concept hides file operation details from the user?

A. Polymorphism

B. Inheritance

C. Abstraction

D. Encapsulation

Ans: c)

3. What does encapsulation mean in file handling?

A. Writing binary files

B. Combining data and functions into a class

C. Overloading file operators

D. Hiding file names

Ans: b)

4. Which C++ class is used for reading files?

A. ofstream

Notes

B. ifstream

C. fstream

D. iostream

Ans: b)

5. Which OOP feature allows extending file functionality without rewriting base logic?

A. Encapsulation

B. Inheritance

C. Abstraction

D. Composition

Ans: b)

Descriptive Questions:

1. What is file handling in OOP?
2. What is the role of inheritance in file handling?
3. Explain the concept of file handling in OOP with an example.
4. How does inheritance improve OOP-based file handling?
5. What are best practices for implementing file handling in OOP?

14.8: References and Suggested Readings

- Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java concurrency in practice. Addison-Wesley Professional.
- Lea, D. (1999). Concurrent programming in Java: Design principles and patterns (2nd ed.). Addison-Wesley Professional.
- Williams, A. (2019). C++ concurrency in action (2nd ed.). Manning Publications

GLOSSARY

1. Abstraction – Hiding internal implementation details and showing only the necessary features of an object.
2. Access Specifiers – Keywords that define the access level of class members (public, private, protected).
3. Application Area – Specific domain or use-case where a programming language is applied (e.g., web, mobile, AI).
4. Base Class – The class from which another class inherits.
5. Binary Operator – An operator that works on two operands (e.g., +, -, *, /).
6. Binary Operator Overloading – Defining custom behavior for binary operators in user-defined classes.
7. Catch Block – Used to handle exceptions thrown in a try block.
8. Class – A user-defined data type that serves as a blueprint for objects.
9. Class Member Function – A function defined inside or outside a class that operates on its objects.
10. Command Line Compilation – The process of compiling source code using a terminal or command prompt.
11. Compile-time Polymorphism – Polymorphism resolved during compilation (e.g., function overloading).
12. Compiler – A tool that translates source code into machine code.
13. Constructor – A special class function automatically called to initialize an object.
14. Constructor Overloading – Defining multiple constructors with different parameter lists.
15. Copy Constructor – A constructor that creates a new object as a copy of an existing object.
16. Control Statements – Statements like if, for, while, used to control program flow.
17. Data Abstraction – Representation of essential features without including background details.
18. Data Hiding – Keeping internal data private within a class to prevent unauthorized access.
19. Data Members – Variables declared inside a class.
20. Default Constructor – A constructor that takes no parameters.

Notes

21. Destructor – A special function invoked when an object goes out of scope or is deleted.
22. Dynamic Binding – Resolving method calls at runtime using virtual functions.
23. Encapsulation – Binding data and code together into a single unit (class).
24. Exception – An error that occurs during program execution.
25. Exception Handling – Mechanism to handle runtime errors using try, catch, and throw.
26. Explicit Type Conversion – Manually converting one type to another using casting.
27. File Handling – Reading from and writing to files using fstream, ifstream, ofstream.
28. File Mode – Flags like ios::in, ios::out, ios::app that control file operations.
29. File Stream – The stream used to handle file input/output.
30. Friend Function – A non-member function that can access private members of a class.
31. Function Overloading – Multiple functions with the same name but different parameters.
32. Getter Function – A member function used to access private data.
33. Header File – A file containing class declarations and function prototypes (.h/.hpp).
34. Heap Memory – Dynamically allocated memory managed during runtime.
35. Hybrid Inheritance – A combination of more than one type of inheritance.
36. IDE (Integrated Development Environment) – Software providing code editor, compiler, and debugger.
37. ifstream – Input file stream used to read from files.
38. Inheritance – Mechanism by which one class can acquire properties of another.
39. Inline Function – A function defined inside the class; suggested for inlining by the compiler.
40. Input/Output Streams – Mechanisms to handle I/O operations (cin, cout, fstream, etc.).
41. Instance – A specific object created from a class.

42. Interface – A structure that allows different objects to be manipulated through common methods.
43. ios::app – File mode to append data to a file.
44. ios::in – File mode to open a file for reading.
45. ios::out – File mode to open a file for writing.
46. Is-A Relationship – A relationship denoting inheritance (Dog is-a Animal).
47. Main Function – Entry point of any C++ program.
48. Member Access Operator (.) – Used to access members of an object.
49. Member Initialization List – Syntax used to initialize class members in a constructor.
50. Member Variable – Another term for data member.
51. Method Overriding – Re-defining a base class method in a derived class.
52. Multilevel Inheritance – A class inherits from a derived class, forming a chain.
53. Multiple Inheritance – A class inherits from more than one base class.
54. Namespace – A container for identifiers to avoid naming conflicts.
55. Object – An instance of a class containing its data and functions.
56. Object-Oriented Programming (OOP) – A programming paradigm based on the concept of classes and objects.
57. ofstream – Output file stream used to write to files.
58. Operator Function – A function used to overload an operator.
59. Operator Overloading – Defining custom behavior for C++ operators in classes.
60. Overloading Rules – Syntax and semantic constraints for operator/function overloading.
61. Overriding – Providing a new definition of a base class function in a derived class.
62. Parameter – Variable used in function declaration to accept input.
63. Pointer – A variable that stores the memory address of another variable.
64. Pointer to Object – A pointer that holds the address of an object.

65. Polymorphism – Ability of different objects to respond differently to the same function call.
66. Pure Virtual Function – A virtual function with no definition in base class; forces derived classes to override it.
67. Runtime Polymorphism – Achieved using virtual functions and inheritance.
68. Setter Function – A member function used to modify private data.
69. Single Inheritance – A derived class inherits from only one base class.
70. Stack Unwinding – Process of removing function calls from the call stack during exception handling.
71. Standard Exceptions – Predefined exceptions available in C++ standard library.
72. Static Binding – Method resolution done at compile time.
73. Template – A feature for creating generic functions or classes.
74. this Pointer – A pointer that refers to the calling object itself.
75. Throw – Used to signal the occurrence of an exception.
76. Try Block – Wraps code that may throw an exception.
77. Type Conversion – Changing data from one type to another.
78. Unary Operator – Operator that works with a single operand.
79. Virtual Function – A function declared with virtual keyword to support runtime polymorphism.
80. Virtual Table (vtable) – A table used internally to support dynamic dispatch of virtual functions.

MATS UNIVERSITY

MATS CENTRE FOR DISTANCE AND ONLINE EDUCATION

UNIVERSITY CAMPUS: Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 **Toll Free ODL MODE :** 81520 79999, 81520 29999

Website: www.matsodl.com