Unit-I

Principles of Objective Oriented Programming, Object Oriented Programming Paradigm, Basic Concepts of Object Oriented Programming, Benefits of Object Oriented Programming, Object Oriented Languages, Applications of Object Oriented Programming, Beginning with C++. Token Expressions & Control Structures: Tokens, Keywords, Identifiers and Constants, Data Types, Type Compatibility, Variables, Operators in C++, Implicit Conversions, Operator Overloading, Operator Precedence, Control Structures.

Unit-II

Functions in C++, Classes & Objects: The Main Function, Function Prototyping, Call by Reference, Return by Reference, Inline Functions, Function Overloading, Friend and Virtual Functions. Specifying a Class, Member Functions, Arrays within a Class, Static Member Functions, Arrays of Objects, Friendly Functions.

Unit-III

Constructors & Destructors, Operator Overloading, Inheritance: Constructors, Parameterized Constructors, Copy Constructors, Dynamic Constructors, Destructors, Defining Operator Overloading, Overloading Operators, Rules for Overloading Operators, Type Conversions. Inheritance: Single Inheritance, Multilevel Inheritance, Multiple Inheritance, Hierarchical Inheritance, Hybrid Inheritance.

Unit-IV

Pointers, Virtual Functions & Polymorphism, Working with Files, Exception Handling: Pointers, Pointers to Objects, this Pointer, Pointer to Derived Classes, Virtual Functions, Classes for File Stream Operations, Opening and Closing a File, File Modes, File Pointers, Input Output Operations, Updating a File.

Reference Books:

1. Object Oriented Design by Rumbaugh (Pearson publication)
2. Object-oriented programming in Turbo C++ By Robert Lafore ,Galgotia Publication.
3. Object-oriented programming with C++ by E.Balagurusamy, 2ndEdition, TMH

INDEX

UNIT – I: Object Oriented Programming & C++ Basics

1. Principles of Object-Oriented Programming .....
1.1 Object Oriented Programming Paradigm .....
1.2 Basic Concepts of Object Oriented Programming .....
- *Class* .....
- *Object* .....
- *Encapsulation* .....
- *Abstraction* .....
- *Inheritance* .....
- *Polymorphism* .....
- *Dynamic Binding* .....
- *Message Passing* .....

1.3 Benefits of Object-Oriented Programming .....
1.4 Object Oriented Languages .....
1.5 Applications of Object Oriented Programming .....
2. Beginning with C++ .....
2.1 Introduction to C++ .....
2.2 Features of C++ .....
2.3 Structure of a C++ Program and Simple C++ Program .....
2.4 Compilation and Execution Process .....
3. Token Expressions .....
3.1 Tokens in C++ .....
- *Keywords* .....
- *Identifiers* .....
- *Constants* .....

4. Data Types & Variables .....
4.1 Data Types in C++ .....
- *Basic Data Types* .....
- *Derived Data Types* .....
- *User Defined Data Types* .....

4.3 Variables .....
- *Local Variables* .....
- *Global Variables* .....
- *Static Variables* .....
- *Automatic Variables* .....
- *Register Variables* .....

5. Operators in C++
5.1 Operators in C++
- *Arithmetic Operators*
- *Relational Operators*
- *Logical Operators*
- *Assignment Operators*
- *Increment and Decrement Operators*
- *Bitwise Operators*
- *Conditional Operator*
- *Scope Resolution Operator*

5.2 Type Conversion
- *Implicit Conversion*
- *Explicit Conversion*

5.3 Operator Overloading
5.4 Operator Precedence
6. Control Structures
6.1 Control Structures in C++
6.2 Selection Statements
- *If*
- *if–else*
- *nested if*
- *switch*

6.3 Iteration Statements
- *for*
- *while*
- *do–while*

6.3 Jump Statements
- *Break*
- *Continue*
- *Goto*
- *return*

UNIT – II: Functions, Classes and Objects

1. Functions in C++
1.1 Function Introduction
1.2 Structure of Function
1.2 Types of Functions
- *Built in*
- *User Defined*

1.2.1 Classification of user defined functions

- *No arguments, No return*
- *Arguments, No return*
- *No arguments, With return*
- *Arguments, With return*

1.2.2 Differences between Library and User Defined Functions
1.2.3 Parameter Passing
- *Call by Value*
- *Call by Reference*
- *Return by Reference*

2. Special Functions
2.1 Friend Functions
2.2 Inline Functions
2.3 Polymorphism
- *Compile Time Polymorphism*
  - *Function Overloading*
  - *Operator Overloading*
- *Dynamic Polymorphism*
  - *Virtual Functions*

2.3.1 Comparison between Compile time and Runtime
2.4 Pure Virtual Function
2.4.1 Comparison between Pure Virtual Function Virtual Function
2.5 Abstract Class

3. Classes and Object
3.1 Class Introduction
3.1.1 Access Controls
- *Public*
- *Private*
- *Protected*

3.2 Creating Object
3.3 Arrays
3.3.1 Types of Arrays
- *Single*
- *Two Dimensional*
- *Multi-Dimensional*

3.3.2 Array of Objects
3.3.3 Array with in a class
3.4 Static Member Functions

3.5 Arrays of Objects

UNIT – III: Constructors, Destructors, Operator Overloading & Inheritance

1. Constructors and Destructors

1.1 Constructors

1.2 Different Types of Constructors

- Parameterized Constructors
- Copy Constructors
- Dynamic Constructors

1.3 Destructors

1.4 Difference between Constructor and Destructor

2. Operator Overloading & Type Conversion

2.1 Operator Overloading

2.2 Type Conversions

Implicit Conversion

Explicit Conversion

3. Inheritance

3.1 Introduction to Inheritance

3.2 Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritances
- Hierarchical Inheritance
- Hybrid Inheritance

UNIT – IV: Pointers, Polymorphism, Files & Exception Handling

1. Pointers in C++

1.1 Pointers

1.2 Pointers to Objects

1.3 this Pointer

1.4 Pointer to Derived Classes

2. File Handling in C++

2.1 File Streams

2.2 File Operations

2.3 File Modes

2.4 File Pointers

2.4 Updating a File

3. Exception Handling

3.1 Need for Exception Handling

3.2 Basic Exception Handling Mechanism

- *Try*.....
- *Catch*.....
- *Throw*.....

1. Principles of Object –Oriented Programing

1.1 Object Oriented Programming Paradigm

*The **Object Oriented Programming (OOP) paradigm** is a method of program design in which software is organized around **objects** rather than functions or procedures. An object represents a real-world entity and combines **data (attributes)** and **operations (methods)** that act on that data into a single unit. This approach models programs in a way that closely resembles real-life systems, making software easier to understand and manage.*

*In the object oriented paradigm, a program is viewed as a collection of interacting objects. Each object is responsible for performing specific tasks and communicates with other objects through **message passing**. This interaction-based design improves modularity and reduces interdependence between different parts of the program.*

*A key principle of the OOP paradigm is **encapsulation**, which restricts direct access to an object's internal data and allows interaction only through well-defined interfaces. This protects data from unintended modification and enhances program reliability. Another important principle is **abstraction**, which focuses on exposing only essential features while hiding implementation details, thereby reducing complexity.*

*The OOP paradigm also supports **inheritance**, enabling new classes to acquire properties and behavior from existing classes. This promotes code reusability and simplifies program maintenance. **Polymorphism** allows the same operation to behave differently depending on the object involved, making programs flexible and extensible.*

Overall, the object oriented programming paradigm provides a structured and systematic approach to software development. By emphasizing modularity, reusability, and data security, it helps in building large, complex, and maintainable software systems efficiently.

1.2 Basic Concepts of Object Oriented Programming

*Object Oriented Programming (OOP) is a programming methodology in which a software system is designed by modeling real-world entities as **objects**. These objects are created using **classes**, and they interact with one another to perform tasks. OOP focuses on important principles such as **modularity**, **data security**, **reusability**, and **easy maintenance**. The main concepts of Object Oriented Programming are **Class, Object, Encapsulation, Abstraction, Inheritance, Polymorphism, Dynamic Binding, and Message Passing**.*

Class:-

*A **class** is a blueprint or template used to create objects. It defines what data an object will store and what operations it can perform. The data members represent the properties of the object, while the member functions represent its behavior. A class is only a logical description and does not occupy memory until objects are created from it.*

Classes help programmers organize programs in a clear and systematic way. By grouping related data and functions into a single unit, classes improve readability and maintainability of the code.

Example:

A `Student` class may contain data members such as `rollNumber` and `name`, and member functions such as `readDetails()` and `displayDetails()`. The class only defines these properties and functions; actual values are stored only when objects are created.

Object:-

*An **object** is an instance of a class. It represents a real-world entity and contains actual values of the data members defined in the class. Objects occupy memory and are the active elements of a program. Every object has **identity** (name), **state** (data), and **behavior** (functions).*

Objects interact with each other during program execution by calling methods. Most of the work in an object-oriented program is done through objects.

Example:

If `Student` is a class, then `student1` and `student2` are objects of that class. Each object will have its own roll number and name, even though they are created from the same class.

Encapsulation:-

***Encapsulation** is the process of combining data and the methods that operate on that data into a single unit called a class. It also involves hiding the internal details of an object and restricting direct access to data using access specifiers such as private and protected.*

Encapsulation improves data security by preventing unauthorized access. It ensures that data can be modified only through well-defined methods, reducing errors and increasing reliability.

Example:

In a `BankAccount` class, the balance amount can be declared as private. It can be accessed or modified only through methods like `deposit()` and `withdraw()`. This prevents users from directly changing the balance.

Abstraction:-

Abstraction *is the concept of hiding internal implementation details and showing only the essential features to the user. It helps reduce complexity by allowing the user to focus on what an object does rather than how it does it.*

Abstraction is achieved using classes, interfaces, and abstract methods. It makes programs easier to understand and use.

Example:

A television remote is a good example of abstraction. When you press the power or volume button, the TV responds. You do not know how the signals travel or how the circuits work internally. You only use the buttons. The visible buttons represent abstraction, while the internal working is hidden.

Inheritance:-

Inheritance *is a mechanism in which one class acquires the properties and behaviors of another class. The class whose features are inherited is called the* ***base class****, and the class that inherits them is called the* ***derived class****.*

Inheritance promotes code reusability, reduces duplication, and supports hierarchical relationships. It allows programmers to build new classes using existing ones.

Example:

If `Animal` *is a class with a function* `eat()`*, then a class* `Dog` *can inherit from* `Animal`*. The* `Dog` *class can use the* `eat()` *function and also add its own function such as* `bark()`*.*

In simple words, inheritance means creating a new class by using the features of an existing class.

Polymorphism:-

Polymorphism *means "one name, many forms." It allows the same function or operation to behave differently depending on the object that calls it. Polymorphism improves flexibility and makes programs easier to extend and maintain.*

Polymorphism can be achieved using function overloading and function overriding.

Example:

A function named `draw()` *can draw different shapes such as a circle, rectangle, or triangle depending on the object used.*

In simple words, polymorphism allows the same function name to perform different actions.

Dynamic Binding:-

***Dynamic binding** is the process of linking a function call to its correct function definition at runtime rather than at compile time. It is closely related to runtime polymorphism and is achieved using virtual functions.*

Dynamic binding allows the program to decide which function to execute based on the type of object being referenced during execution.

Example:

If a class `Shape` has a function `draw()` and a class `Circle` overrides it, the `draw()` function of `Circle` is called at runtime when a `Shape` reference points to a `Circle` object.

In simple words, dynamic binding means selecting the correct function during program execution.

Message Passing:-

***Message passing** is the method by which objects communicate with each other. An object sends a message to another object by calling one of its methods. This allows objects to interact while keeping their data secure and independent.*

Message passing supports modularity and helps in building flexible and loosely coupled systems.

Example:

If one object wants to display student details, it sends a message by calling the `display()` method of another object. The receiving object performs the required action.

In simple words, message passing means one object requesting another object to perform a task.

1.3 Benefits of Object-Oriented Programming

Object Oriented Programming is a programming approach that organizes software using objects, which represent real-world entities. OOP provides several advantages that make software development easier, safer, and more efficient, especially for large and complex systems.

1. Modularity:-

*In Object Oriented Programming, a program is divided into small and independent units called **objects**. Each object is responsible for performing a specific task and contains its own data and functions. This separation of functionality makes the program well-organized and structured.*

Because the program is divided into modules, it becomes easier to understand the logic of the software. If an error occurs, the programmer can focus on the specific object instead of checking the entire program. Testing and debugging become faster and more accurate. Modularity also allows different programmers to work on different parts of the program at the same time, which is very helpful in large projects.

2. Code Reusability:-

*Code reusability is one of the most important advantages of OOP. Using **inheritance**, a new class can reuse the features of an existing class. This means the programmer does not need to write the same code again and again.*

For example, a base class can contain common properties, and multiple derived classes can use those properties. This reduces code duplication, saves development time, and lowers the chance of errors. Reusable code is easier to manage and makes software development more efficient and cost-effective.

3. Easy Maintenance:-

*Object Oriented Programming makes software maintenance simple and efficient. Through **encapsulation**, data and functions are bundled together inside a class. Through **abstraction**, only essential details are shown to the user, while internal details are hidden.*

Because of this structure, changes made in one class do not affect other classes. If a bug is found or a feature needs to be updated, the programmer can modify only the required class. This reduces the risk of introducing new errors and makes long-term maintenance easier, especially in large software systems.

4. Data Security:-

*OOP provides strong support for **data security**. Data members of a class are usually declared as private or protected, which prevents direct access from outside the class. Data can only be accessed or modified through public methods.*

This controlled access ensures that data is not accidentally or maliciously changed. By protecting important information, OOP increases the reliability and safety of programs. This feature is very important in applications such as banking systems, online transactions, and data management systems.

5. Scalability:-

*Object Oriented Programming supports the development of **scalable applications**. As software requirements grow, new classes and objects can be added without disturbing the existing code.*

Because objects are independent, the program can be extended easily. This makes OOP suitable for large projects that evolve over time. Scalability helps organizations upgrade their software without rewriting the entire system, saving both time and cost.

6. Real-World Representation:-

*OOP allows software to be designed using **real-world concepts** such as people, machines, accounts, and systems. These real-world entities are represented as objects with properties (data) and behaviors (functions).*

This approach makes program design more natural and easier to understand. Programmers can visualize how objects interact with each other, which improves clarity and reduces logical errors. Real-world modeling makes software more intuitive and meaningful.

7. Improved Productivity:-

By promoting modularity, reusability, and structured design, OOP increases programmer productivity. Developers can reuse existing code, reduce development time, and focus more on problem-solving rather than rewriting code.

Well-organized programs are easier to read, test, and debug. This leads to faster project completion and better software quality. As a result, OOP helps organizations deliver reliable software within shorter time frames.

8. Better Program Organization

OOP encourages a clear and systematic organization of code. Classes and objects are arranged logically, making the program easier to read and understand. A well-organized program improves teamwork and helps new programmers understand the system quickly.

This structure also improves documentation and makes future enhancements simpler. Good organization is essential for managing large software projects.

9. Reduced Complexity:-

By breaking a large problem into smaller objects, OOP reduces program complexity. Each object handles a small part of the problem, making the overall system easier to design and manage.

This reduction in complexity improves software reliability and reduces development errors. Programmers can focus on one object at a time, which simplifies problem-solving.

1.4 Object Oriented Languages

Object Oriented Languages *are programming languages that are designed to support the principles of* ***Object Oriented Programming (OOP)****. These principles include* ***class, object, encapsulation, abstraction, inheritance, polymorphism, dynamic binding, and message passing****.*
Using object oriented languages, programmers can design software by modeling real-world entities and their interactions. This results in programs that are ***well-structured, reusable, secure, scalable, and easy to maintain****.*

Object oriented languages help in developing large and complex applications because they divide the program into independent objects, each responsible for a specific task.

Some commonly used object oriented languages are explained below.

C++:-

C++ is an extension of the C programming language that supports both ***procedural programming*** *and* ***object oriented programming****. It allows programmers to write low-level system code as well as high-level object oriented code.*

C++ supports all major OOP concepts such as ***classes and objects*** *for program structure,* ***encapsulation*** *for data security,* ***inheritance*** *for code reusability, and* ***polymorphism*** *for flexibility. It also supports* ***dynamic binding*** *through virtual functions.*

Because of its efficiency and performance, C++ is widely used in ***system software, operating systems, game development, embedded systems, and application software****. It is suitable for programs where speed and memory control are important.*

Java:-

Java is a ***fully object oriented programming language*** *designed to be* ***platform independent****. It follows the principle of* ***"write once, run anywhere"****, which means a Java program can run on any system that supports the Java Virtual Machine (JVM).*

In Java, everything is written using ***classes and objects****. It strongly supports* ***encapsulation*** *by using access specifiers and* ***abstraction*** *through interfaces and abstract classes.* ***Inheritance*** *and* ***polymorphism*** *are used extensively to build reusable and flexible programs.*

Java is widely used in ***web applications, mobile applications (Android), enterprise systems, and large-scale software projects****. It is known for its security, robustness, and portability.*

Python:-

*Python is a **high-level programming language** that supports object oriented programming along with procedural and functional programming styles. Python allows programmers to create classes and objects easily using simple and readable syntax.*

*Python supports OOP concepts such as **encapsulation, inheritance, and polymorphism**, making it suitable for building modular and reusable programs. Its abstraction capability helps in writing clean and easy-to-understand code.*

*Due to its simplicity and powerful libraries, Python is widely used in **data science, artificial intelligence, machine learning, web development, automation, and scientific computing**. Python improves productivity by reducing development time.*

C#:-

*C# (C-Sharp) is a modern object oriented programming language developed as part of the **.NET framework**. It is designed to build secure, robust, and scalable applications.*

*C# fully supports OOP principles such as **classes, objects, encapsulation, inheritance, and polymorphism**. It also supports **type safety and abstraction**, which help in reducing programming errors.*

*C# is mainly used for **desktop applications, web applications, enterprise software, and game development** (using Unity). It provides good performance and strong integration with Microsoft technologies.*

Ruby:-

*Ruby is a **pure object oriented programming language**, where everything is treated as an object, including numbers and strings. It emphasizes simplicity, readability, and programmer happiness.*

*Ruby supports all OOP concepts such as **class, object, encapsulation, inheritance, and polymorphism**. Its simple syntax makes object oriented programming easy to learn and use.*

*Ruby is mainly used in **web development**, especially with the Ruby on Rails framework. It is popular for rapid application development because it allows programmers to build applications quickly and efficiently.*

1.5 Applications of Object-Oriented Programming

Object-Oriented Programming (OOP) is widely used in modern software development due to its ability to handle complexity through objects, classes, and modular design. By following OOP principles such as encapsulation, inheritance, abstraction, and polymorphism, software systems

become more reliable, reusable, and easy to maintain. The major application areas of Object-Oriented Programming are explained below.

Real-Time Systems:-

OOP is extensively used in real-time systems where correct results must be produced within strict time limits. Examples include air traffic control systems, medical monitoring systems, and industrial automation. Objects help in organizing system components efficiently, ensuring accuracy, reliability, and timely response.

Simulation and Modeling:-

Object-Oriented Programming is highly suitable for simulation and modeling of real-world systems. Complex systems such as weather forecasting, traffic control, power plants, and scientific experiments can be represented using objects. Each object models a real-world entity, making simulations easier to design, understand, and modify.

Graphical User Interface (GUI) Applications:-

In GUI-based applications, interface components such as windows, buttons, menus, icons, and dialog boxes are treated as objects. OOP allows these components to be reused and extended easily, helping in the development of interactive, user-friendly, and visually appealing applications.

Game Development:-

OOP plays a major role in game development. Game characters, weapons, levels, and environments are represented as objects. This approach makes game design flexible and supports reuse of code, enabling easy addition of new features, characters, or levels.

Web and Mobile Applications:-

Modern web and mobile applications use OOP concepts to manage user information, application logic, databases, and security features. OOP helps developers build scalable and maintainable applications that can be updated easily as user requirements change.

Embedded Systems:-

Object-Oriented Programming is used in embedded systems such as automobiles, washing machines, smart TVs, medical devices, and home automation systems. Objects help in organizing hardware control, sensor handling, and software logic in a structured manner.

Artificial Intelligence and Data-Oriented Applications:-

OOP is widely used in artificial intelligence, data analysis, and machine learning applications. It helps in managing complex data structures, algorithms, and intelligent agents in an organized way, improving clarity and reusability of code.

Enterprise and Business Applications:-

Large-scale enterprise applications such as banking systems, insurance software, payroll systems, inventory management, and customer relationship management systems rely heavily on OOP. These applications involve large amounts of data and complex operations, which are efficiently, handled using object-oriented design.

Database Management Systems:-

OOP is used in database management systems to model real-world data and relationships between data entities. Object-oriented databases and object-relational mapping techniques make data storage and retrieval more organized and efficient.

Distributed and Network Applications:-

OOP is applied in distributed and network-based applications where different systems communicate over a network. Objects represent network components and services, simplifying communication and coordination between systems.

2. Beginning with C++

2. 1 Introduction to C++

*C++ is a widely used **general-purpose programming language** designed to support both **procedural** and **object-oriented programming**. It was developed by **Bjarne Stroustrup** at AT&T Bell Laboratories as an enhancement of the C language. The primary goal of C++ was to provide the efficiency and low-level capabilities of C while introducing features that support object-oriented design.*

*C++ allows programmers to create programs using **classes and objects**, which helps in organizing code in a logical and structured manner. At the same time, it retains all the powerful features of C such as pointers, structures, and low-level memory access. Because of this combination, C++ is often described as a **hybrid language**.*

*One of the important characteristics of C++ is its **high performance**. Being a compiled language, C++ produces fast and efficient executable programs. It also gives programmers greater control over system resources like memory management, which is essential for developing system-level software.*

*C++ supports key object-oriented features such as **encapsulation, inheritance, polymorphism, and abstraction**. These features improve code reusability, reduce complexity, and make programs easier to maintain and extend. In addition, C++ provides a rich set of standard libraries that assist in developing applications efficiently.*

Due to its flexibility, efficiency, and powerful features, C++ is widely used in various application areas including application software, game development, operating systems, embedded systems, database systems, and real-time applications.

***In summary**, C++ is a powerful programming language that combines the strengths of procedural programming with object-oriented concepts, making it suitable for both small programs and large-scale software systems.*

2.2 Features of C++

*C++ is a powerful, flexible, and general-purpose programming language. It supports both **procedural programming** and **object-oriented programming**, which makes it suitable for developing small programs as well as large and complex software systems. Each feature of C++ helps programmers write efficient, well-structured, reusable, and high-performance programs. The important features of C++ are explained below.*

Object-Oriented Programming Support:-

*C++ provides strong support for object-oriented programming concepts such as **classes, objects, encapsulation, abstraction, inheritance, polymorphism, and dynamic binding**. These concepts help in dividing a large program into smaller, manageable units. By using OOP features, programmers can design software that is modular, reusable, secure, and easy to maintain. This makes C++ very useful for developing real-world and large-scale applications.*

Classes and Objects:-

*In C++, a **class** is a blueprint that defines data members and member functions, while an **object** is an instance of a class that contains actual data values. Classes help in grouping related data and functions into a single unit, and objects represent real-world entities. This approach makes program design logical, organized, and easy to understand. Using classes and objects improves clarity and structure of programs.*

Encapsulation:-

*Encapsulation is the mechanism of binding data and the functions that operate on that data into a single unit called a class. In C++, access specifiers such as **private, protected, and public** are*

used to restrict direct access to data. Encapsulation protects data from accidental or unauthorized modification and improves program security. It also helps in maintaining control over how data is accessed and modified.

Abstraction:-

Abstraction focuses on hiding complex internal implementation details and exposing only the essential features to the user. In C++, abstraction allows programmers to work with objects without knowing how their internal operations are performed. This reduces program complexity and makes software easier to understand, use, and maintain. Abstraction helps programmers focus on functionality rather than internal logic.

Inheritance:-

Inheritance allows a new class to acquire the properties and behaviors of an existing class. The existing class is called the ***base class****, and the new class is called the* ***derived class****. Inheritance promotes code reusability by allowing existing code to be reused in new classes. It reduces redundancy and supports hierarchical classification, making program design more systematic and efficient.*

Polymorphism:-

Polymorphism means "one name, many forms." In C++, the same function or operator can perform different actions depending on the context or object used. Polymorphism increases flexibility and allows programs to be easily extended without modifying existing code. It also improves readability and maintainability of programs by reducing complexity.

Dynamic Binding:-

Dynamic binding links a function call to its actual function definition at ***runtime*** *rather than at compile time. It is mainly achieved using* ***virtual functions*** *in C++. Dynamic binding supports runtime polymorphism and allows the program to decide which function to execute based on the object being referenced during execution. This makes programs more flexible and dynamic.*

Hybrid Language:-

C++ is called a ***hybrid language*** *because it supports both* ***procedural programming*** *and* ***object-oriented programming****. Programmers can write traditional C-style programs as well as object-oriented programs using classes and objects. This flexibility allows developers to choose the best programming style according to the problem being solved.*

High Performance:-

C++ is a ***compiled language****, which means programs are converted directly into machine code. This results in fast execution and efficient use of system resources. C++ also provides low-level*

memory access, making it suitable for developing performance-critical applications such as operating systems, game engines, and embedded systems.

Rich Standard Library:-

*C++ provides a rich set of **standard libraries** that support input/output operations, string handling, mathematical calculations, file handling, and data processing. These libraries provide ready-made functions that reduce programming effort and development time. They improve productivity and make program development faster and easier.*

Memory Management:-

*C++ allows **dynamic memory allocation and deallocation**, giving programmers direct control over memory usage. Memory can be allocated when required and released after use. This feature helps in developing resource-efficient and high-performance applications. Proper memory management is especially important in system-level programming.*

Compatibility with C:-

*C++ is highly compatible with the **C programming language**. Most C programs can be compiled using a C++ compiler without major changes. This compatibility allows programmers to reuse existing C code and enhance it with object-oriented features. It makes the transition from C to C++ easier.*

2.3 Structure of a C++ Program

A C++ program is organized into several sections, each having a specific purpose. This structured approach helps programmers write clear, readable, and well-maintained programs. The main sections of a C++ program are explained below.

Documentation Section

*The documentation section appears at the beginning of the program and contains **comments** written by the programmer. This section is used to explain the purpose of the program, the logic involved, and other related details such as author name or date. The compiler ignores everything written in this section, so it does not affect program execution. Although this section is optional, it is highly recommended because it improves program clarity and helps others understand the code easily.*

Linking Section

*The linking section is used to connect the program with **standard library files** that contain predefined functions, classes, and other programming elements. These library files are included*

using preprocessor directives. This section ensures that the program can use built-in input, output, and other utility functions required during execution.

Header Files

Header files contain declarations of standard functions and classes provided by the C++ library. To use these features, the required header files must be included at the beginning of the program. Header files are included using the `#include` *directive. For example, the input-output operations are supported through a specific header file. Including header files allows the compiler to recognize predefined features used in the program.*

Namespace Section

A namespace is used to group related identifiers such as variables, functions, and classes under a single name. This avoids naming conflicts in large programs. The standard library elements of C++ are placed inside a standard namespace. By using a namespace declaration, programmers can directly use standard library features without repeatedly specifying their full names.

Definition Section

The definition section is used to define constants and create alternative names for existing data types. This section improves code readability and maintainability. Compiler directives can be used to define symbolic constants, and type definitions allow programmers to create new data type names using existing primitive types. These definitions remain valid throughout the program.

Global Declaration Section

The global declaration section contains variables, constants, and class definitions that are accessible throughout the program. Variables declared in this section have a global scope, which means they can be used inside all functions of the program. The lifetime of these variables lasts until the program finishes execution. This section is useful when data needs to be shared among multiple functions.

Function Declaration Section

This section contains declarations or prototypes of functions that are used in the program. Function declarations inform the compiler about the function name, return type, and parameters before the function is actually defined. This allows functions to be defined after the main function while still being accessible within it.

Main Function

The `main()` *function is the most important part of a C++ program. Program execution always begins from the main function. All executable statements that control the program flow are*

written inside this function. The body of the main function is enclosed within curly braces. Once all statements inside the main function are executed, the program terminates and control returns to the operating system.

User-Defined Functions

User-defined functions are functions created by the programmer to perform specific tasks. These functions help in dividing a large and complex program into smaller, manageable units. By using user-defined functions, code repetition can be avoided, and program readability is improved. They also support code reusability and make debugging and testing easier.

Simple C++ Program

A simple C++ program demonstrates the basic syntax and execution flow of the language. It helps beginners understand how a C++ program is written, compiled, and executed. Below is a basic example of a C++ program that prints a message on the screen.

C++ Program

```
/********** Documentation Section **************/
/* Program to demonstrate the structure of a C++ program */

#include <iostream>       // Linking Section
                             // Header File
using namespace std;      // Namespace Section

#define PI 3.14           // Definition Section
typedef int number;       // Definition Section

number globalVar = 10;    // Global Declaration  Section

// Function Declaration Section
void display();

int main()                // Main Function
{
   cout << "Value of PI: " << PI << endl;
   display();
   return 0;
}
// User-Defined Function
void display()
{
```

```
    cout << "Global Variable: " << globalVar << endl;
}
```

Output: -

Value of PI: 3.14
Global Variable: 10

Explanation of the Program

Documentation Section

Comments at the top explain the purpose of the program. They are ignored by the compiler.

Linking Section

#include <iostream> links the program with the input–output library.

Header Files

<iostream> provides predefined objects like cout and endl.

Namespace Section

using namespace std; allows direct use of standard library elements.

Definition Section

#define PI 3.14 and typedef int number; define constants and new type names.

Global Declaration Section

globalVar is accessible throughout the program.

Function Declaration Section

void display(); informs the compiler about the function before use.

Main Function

Program execution starts from main().

User-Defined Functions

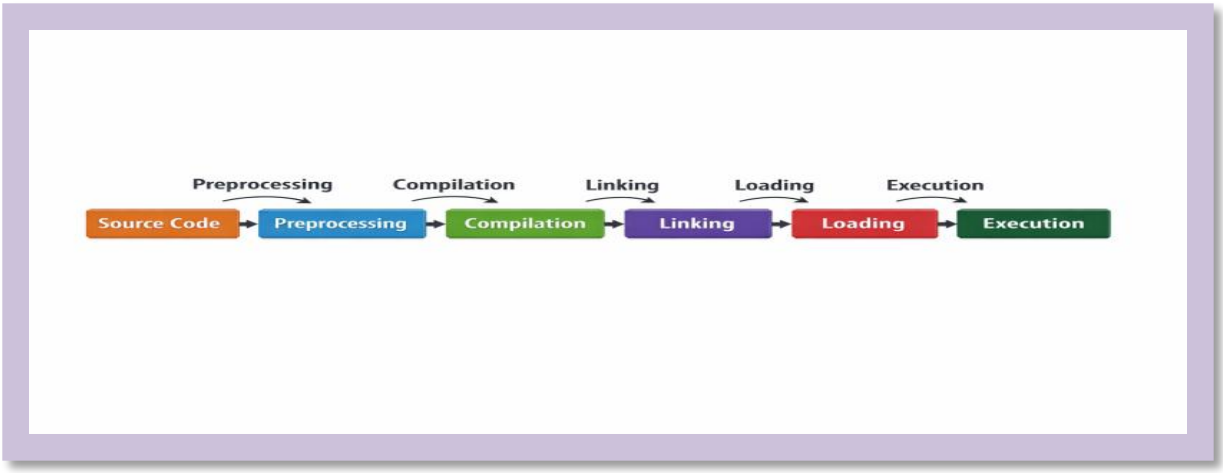
display() is a programmer-defined function called from main().

2.4 Compilation and Execution Process in C++

*The **compilation and execution process in C++** describes the complete journey of a C++ program—from the moment a programmer writes the source code until the program finally runs and produces output.Since C++ is a **compiled language**, the program must pass through multiple well-defined stages before execution.*

Steps involved in Compilation and Execution Process in C++

1. ***Writing the Source Code***
2. ***Preprocessing***
3. ***Compilation***
4. ***Linking***
5. ***Loading***
6. ***Execution***



1. Writing the Source Code

Writing the source code is the first step in the compilation process. In this step, the programmer writes the program instructions using the C++ programming language in a text editor or IDE. The program is saved with a `.cpp` *extension and is written in a form that humans can understand but the computer cannot execute directly.*

Example:

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

int main() {
    cout << "Welcome to C++";
    return 0;
}
```

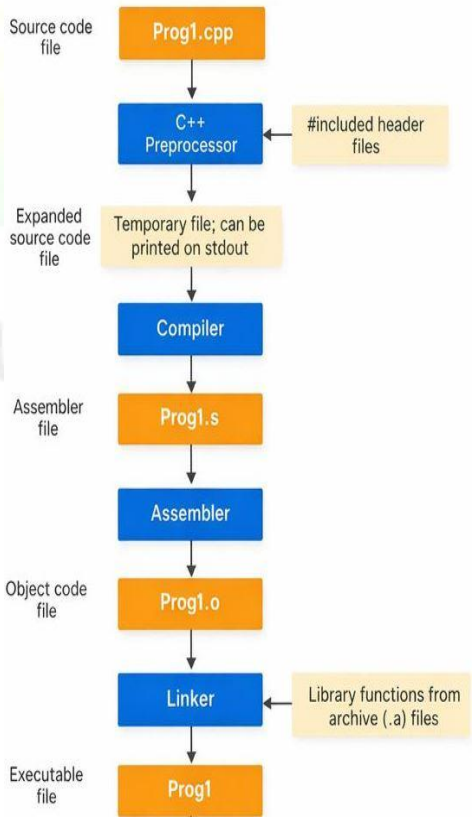
2. Preprocessing

Preprocessing is the stage where all preprocessor directives are handled before actual compilation begins. Statements starting with `#` *are processed, header files are included, macros are expanded, and comments are removed. This step prepares the program for the compiler by simplifying the source code.*

Example:

```
#define PI 3.14
#include <iostream>
```

Here, `PI` *is replaced with* `3.14`*, and the contents of* `iostream` *are added to the program during preprocessing.*



3. Compilation

In the compilation stage, the compiler checks the preprocessed code for syntax and semantic errors. If errors are found, the compiler displays error messages and stops the process. If no errors are found, the compiler converts the code into object code, which is in machine language but not yet executable.

Example:

int a = 10

This statement causes a compilation error because the semicolon (;) is missing.

4. Linking

Linking is the process of combining the object code with the required library files. The linker connects function calls in the program to their actual definitions in the libraries. If the required libraries are not found, linking errors occur. After successful linking, an executable file is created.

Example:

The function `cout` is declared in the program, but its actual definition is linked from the standard C++ library during the linking stage.

5. Loading

Loading is the step where the operating system loads the executable file into the main memory. Memory is allocated for program instructions, variables, stack, and heap. The program is now ready to run.

Example:

When you double-click a C++ executable file or run it using a command like `./a.out`, the operating system loads it into RAM.

6. Execution

Execution is the final stage where the CPU starts executing the program. The execution begins from the `main()` function, and instructions are carried out one by one. The program performs calculations, takes input if required, and displays output until it terminates.

Example:

cout << "Welcome to C++";

*During execution, this statement displays the message **Welcome to C++** on the screen.*

3. Token Expressions

3.1 Tokens in C++

*In C++, a **token** is the **smallest meaningful unit** of a program that the compiler can recognize and understand. When a C++ program is compiled, the compiler does not read the program as a*

whole sentence. Instead, it ***breaks the source code into small pieces called tokens****. These tokens help the compiler analyze the* ***syntax, structure, and meaning*** *of the program.*

In simple words, ***tokens are the basic building blocks of a C++ program****. Without tokens, a program cannot be written or understood by the compiler.*

Example*:*

int sum = a + b;

Here, the tokens are: int, sum, =, a, +, b, ;

Types of Tokens in C++

1. ***Keywords***
2. ***Identifiers***
3. ***Constants (Literals)***
4. ***Operators***
5. ***Punctuators (Separators)***
6. ***Special Symbols***

1. Keywords

Keywords *are* ***reserved words*** *in C++ that have* ***special meanings*** *predefined by the language. They are used to perform specific tasks such as defining data types, controlling program flow, declaring classes, and specifying access permissions.*

Because keywords are reserved, ***they cannot be used as identifiers*** *(names of variables, functions, or classes). The compiler recognizes keywords and treats them differently from normal user-defined names.*

Examples:

auto, break, case, char, class, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, namespace, new, private, public, return, short, signed, sizeof, static, struct, switch, template, this, throw, try, typedef, union, unsigned, virtual, void, while

```
class Student                    // 'class' is a keyword
{
  int marks;                     // 'int' is a keyword
};
```

2. Identifiers

Identifiers are ***user-defined names*** *assigned to program elements such as variables, functions, arrays, classes, and objects. They are used to uniquely identify these elements in a C++ program*

and allow the programmer to store and access data effectively. Meaningful identifiers improve program readability and maintainability.

Rules for identifiers:

- *An identifier must start with a letter (A–Z or a–z) or an underscore (_). It must not start with a digit.*
- *Digits are allowed in identifiers, but they can appear only after the first character.*
- *An identifier must not contain spaces. It should be written as a single continuous word.*
- *Special characters such as @, #, %, &, *, and ! are not allowed. Only letters, digits, and underscore can be used.*
- *C++ keywords cannot be used as identifiers because they have predefined meanings.*
- *Identifiers are case-sensitive, so uppercase and lowercase letters are treated differently.*
- *Identifiers should be meaningful and descriptive to improve program readability.*
- *Identifiers should not be too long; short and meaningful names are preferred.*
- *A consistent naming style should be followed throughout the program.*

Examples

- ***Valid:*** *sum, totalMarks, _count, value1*
- ***Invalid:*** *2sum, total marks, float, @value*

3. Constants (Literals)

*Constants, also called literals, represent **fixed values** that do not change during the execution of a program. These values are directly written in the source code. C++ supports different types of constants such as integer, floating-point, character, string, and boolean constants. The use of constants makes programs easier to understand and reduces errors.*

Types of Constants:

- ***Integer:*** `10, -45, 0x1F`
- ***Floating-point:*** `3.14, -0.05, 2.5e3`
- ***Character:*** `'A', '9', '\n'`
- ***String:*** `"Hello", "C++"`
- ***Boolean:*** `true, false`

Example*:*

```
const float PI = 3.14159;   // PI is a constant
```

4. Operators

*Operators are **symbols used to perform operations** on variables and constants. They enable arithmetic calculations, comparisons, logical decisions, and data assignments. C++ provides various operators such as arithmetic operators, relational operators, logical operators, assignment operators, and increment or decrement operators. Operators are essential for manipulating data and controlling program behavior.*

Main Categories of Operators:

- ***Arithmetic:*** `+ - * / %`
- ***Relational:*** `== != > < >= <=`
- ***Logical:*** `&& || !`
- ***Assignment:*** `= += -= *= /=`
- ***Increment/Decrement:*** `++ --`
- ***Bitwise:*** `& | ^ ~ << >>`
- ***Conditional:*** `?:`
- ***Scope Resolution:*** `::`

Example:
```
int c = a + b;    // '+' operator
if (c > 10)       // '>' operator
```

5. Punctuators (Separators)

*Punctuators, also known as separators, are **special characters used to organize and structure the program**. They separate statements, define blocks of code, and group expressions. Common punctuators in C++ include semicolons, commas, parentheses, braces, and brackets. Proper use of punctuators ensures correct syntax and program flow.*

Examples:

- `;` → ***statement terminator***
- `{ }` → ***block of code***
- `( )` → ***function call or expression grouping***
- `[ ]` → ***array subscripts***
- `,` → ***separates variables or parameters***

Example:
```
int x, y;          // ',' is a punctuator
if (x > y)         // '{}' are punctuators
{
cout << x;
}
```

6. Special Symbols

*Special symbols are characters that have **specific functional meanings** in C++. They are used for tasks such as preprocessing, scope resolution, pointer declaration, and input-output operations. Examples include the hash symbol (#), scope resolution operator (::), address operator (&), pointer symbol (*), and stream operators (<<, >>). These symbols support advanced programming features in C++.*

Examples:

- `#` → ***preprocessor directive***
- `::` → ***scope resolution***
- `->` → ***pointer to member***
- `.` → ***member access***

Example Program with Tokens

```
#include <iostream>        // #, < > → special symbols; include → keyword
using namespace std;       // using, namespace → keywords; std → identifier
int main()                 // int → keyword; main → identifier; ( ) → punctuators
{
   int a = 10, b = 20;     // identifiers, constants, operator, punctuator
```

```
    int sum = a + b;        // operators and identifiers
    cout << "Sum = " << sum;  // << operator, string constant
    return 0;               // return → keyword; 0 → constant
}
```

Total Number of Tokens=40

4.1 Data Types in C++

*In C++, **data types** specify the **kind of data** a variable can store in memory. Every variable must be declared with a data type before it is used. The data type tells the compiler:*

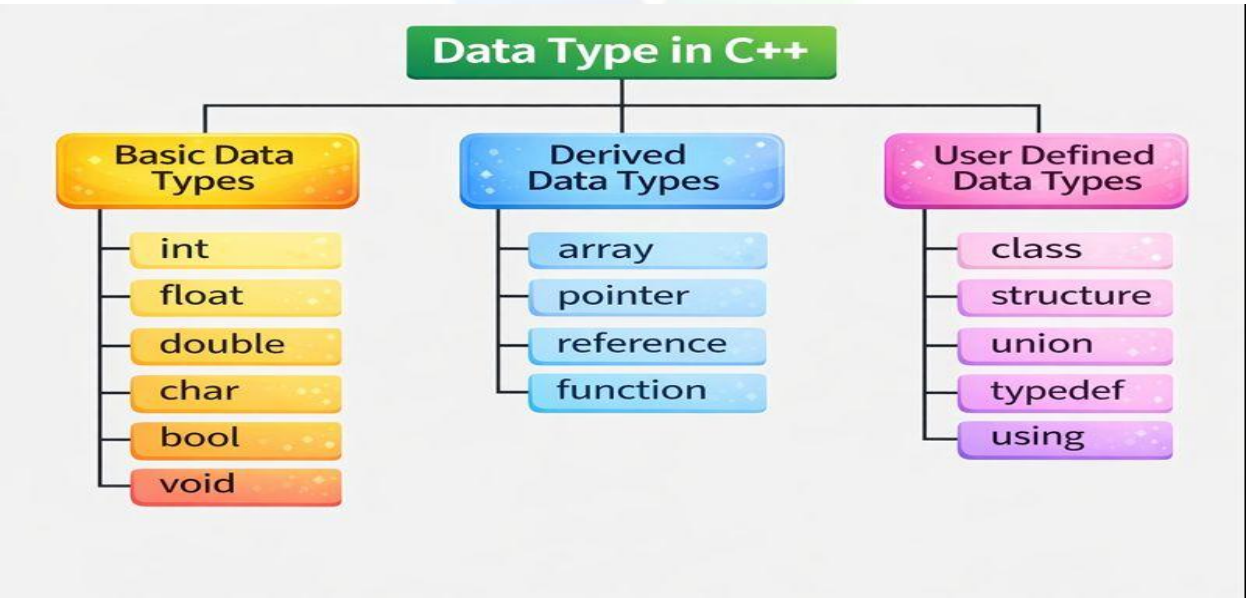
- *How much **memory** to allocate*
- *What type of **values** can be stored*
- *What kind of **operations** can be performed on the data*

*Data types help the compiler understand the nature of data and ensure **correct and efficient program execution**.*

Classification of Data Types in C++

*C++ data types specify **the type of data a variable can hold**. They are mainly divided into:*

1. ***Basic (Primitive) Data Types***
2. ***Derived Data Types***
3. ***User-Defined Data Types***



1. Basic Data Types

*Basic or primitive data types are the **core data types** available in the C++ programming language. They are used to store **simple, single values** such as numbers, characters, and logical conditions. These data types require a **fixed amount of memory**, and the compiler directly*

understands and processes them. All other complex data types in C++ are ultimately built using these basic data types.

The commonly used basic data types in C++ are:

- ***int***
- ***float***
- ***double***
- ***char***
- ***bool***

Each of these is explained in detail below.

- ***Int: -****The int data type is used to store **whole numbers** that do not contain decimal values. It can store both **positive and negative integers**. The int type is most commonly used in programs for calculations, counters, loops, and indexing operations.*

  Example:

  int count = 10;

- ***Float: -****The float data type is used to store **decimal numbers** with single-precision accuracy. It is suitable when values require fractional representation but do not need very high precision. This type is commonly used for measurements, averages, and scientific values with limited accuracy.*

  Example:

  float averageMarks = 78.5f;

- ***Double: -****The double data type is used to store **decimal values with higher precision** than float. It occupies more memory and provides more accurate results. Because of its precision, double is widely used in scientific computations, engineering applications, and financial calculations.*

  Example:

  double distance = 12345.6789;

- ***Char : -****The char data type is used to store **a single character**, such as a letter, digit, or special symbol. Characters are written inside **single quotation marks**. Internally, each character is stored as a numeric value based on the ASCII code.*

  Example:

  char section = 'B';

- ***Bool : -****The bool data type is used to store **logical values**. It can hold only one of two values: true or false. Boolean variables are mainly used in **conditions, decision-making statements, and logical expressions**.*

  Example:*bool isLoggedIn = false;*

Common Data Types in C++

Category	Data Type	Example Declaration	Size (Bytes)	Range / Values
Integer	int	int i = -200000;	4	−2,147,483,648 to 2,147,483,647

Integer	unsigned int	unsigned int u = 500;	4	0 to 4,294,967,295
Integer	short int	short int s = 30000;	2	−32,768 to 32,767
Integer	unsigned short int	unsigned short int us = 50000;	2	0 to 65,535
Integer	long int	long int l = 100000L;	4 / 8	System dependent (large range)
Integer	unsigned long int	unsigned long int ul = 500000UL;	4 / 8	0 to very large
Integer	long long int	long long int ll = 10000000000LL;	8	-9.22×10^{18} to 9.22×10^{18}
Integer	unsigned long long int	unsigned long long int ull = 10000000000ULL;	8	0 to 1.84×10^{19}
Character	char	char c = 'A';	1	−128 to 127 / 0 to 255
Character	signed char	signed char sc = -100;	1	−128 to 127
Character	unsigned char	unsigned char uc = 200;	1	0 to 255
Floating Point	float	float f = 3.14f;	4	1.2×10^{-38} to 3.4×10^{38}
Floating Point	double	double d = 3.14159;	8	2.3×10^{-308} to 1.7×10^{308}
Floating Point	long double	long double ld = 3.14L;	10 / 12 / 16	3.4×10^{-4932} to 1.1×10^{4932}
Boolean	bool	bool b = true;	1	true or false
Void	void	void myFunction();	—	No value

2. Derived Data Types

*Derived data types are data types that are **formed from basic (primitive) data types**. They are used when a program needs to store **multiple values**, work with **memory locations**, or organize code into reusable units. Derived data types help in writing **efficient, structured, and manageable programs**, especially when handling large and complex data.*

- ***Array***
- ***Pointer***
- ***Reference***
- ***Function***

- ***Array:-*** *An array is a collection of elements of the **same data type** stored in **consecutive memory locations**. Each element of an array is identified by an index, starting from zero. Arrays are commonly used to store multiple related values under a single variable name, such as student marks, salaries, or temperature readings.*

  Example:

```
int numbers[4] = {10, 20, 30, 40};
```

- ***Pointer****:-A pointer is a special variable that stores the **address of another variable** in memory. Instead of storing a direct value, it points to the memory location where the value is*

stored. Pointers provide greater control over memory and are widely used in dynamic memory management and efficient data handling.

Example:

```
int x = 25;
int* p = &x;
```

❖ ***Reference:-****A reference is an* ***alternate name*** *given to an already existing variable. Once a reference is created, it always refers to the same variable and cannot be changed to refer to another. References are mainly used in function calls to improve performance and allow direct modification of variables.*

Example:

```
int value = 10;
int& r = value;
```

❖ ***Function:-****A function is a* ***self-contained block of code*** *designed to perform a specific task. It may take input values and may return a result to the calling function. Functions promote* ***modularity***, ***code reuse***, *and better readability by dividing a program into smaller logical units.*

Example:

```
int square(int n)
{
    return n * n;
}
```

3. User-Defined Data Types

User-defined data types are data types that are ***defined by the programmer*** *to meet the specific requirements of a program. They allow the programmer to create* ***new data types*** *by combining existing data types in a meaningful way. User-defined data types help in organizing data efficiently, improving program readability, and managing large and complex programs.*

- ***Struct***
- ***Class***
- ***Enum***
- ***typedef / using***

❖ ***Structure (struct):-****A* ***structure*** *is a user-defined data type that groups* ***variables of different data types*** *under a single name. Each variable inside a structure is called a* ***member*** *of the structure. Structures are widely used to represent real-world entities where different kinds of information are related to each other.*

Syntax

```
struct structure_name
{
```

Example:

```
struct Student
{
   int rollNo;
   char grade;
};
```

```
    data_type member1;
    data_type member2;
    data_type member3;
};
```

❖ ***Union (union):-****A* ***union*** *is a user-defined data type similar to a structure, but all its members* ***share the same memory location****. At any given time, only one member of a union can store a value. Unions are mainly used when memory usage needs to be minimized.*

Syntax

```
union union_name
{
    data_type member1;
    data_type member2;
    data_type member3;
};
```

Example:

```
union Data
{
    int number;
    float value;
};
```

❖ ***Enumeration (enum):-****An* ***enumeration*** *is a user-defined data type that consists of a set of* ***named integer constants****. Enumerations improve program readability by replacing numeric values with meaningful names. They are commonly used where a variable can take only a fixed set of values.*

Syntax

```
enum enum_name
{
    constant1,
    constant2,
    constant3
};
```

Example:

```
enum Day
{
    Monday, Tuesday, Wednesday, Thursday, Friday
};
```

❖ ***Class (class):-****A* ***class*** *is a user-defined data type that forms the foundation of* ***object-oriented programming****. It combines* ***data members and member functions*** *into a single unit. Classes support important concepts such as encapsulation and data hiding, which help in building secure and well-structured programs.*

Syntax

```
class class_name
{
    access_specifier:
        data_member1;
        data_member2;

        member_function1();
        member_function2();
```

Example:

```
class Student
{
    int rollNo;          // private by default
public:
    void setRoll(int r);
};
```

};

❖ ***typedef / using:-****The typedef and using keywords are used to create a* ***new name (alias)*** *for an existing data type. This simplifies complex declarations and makes programs easier to understand.*

Syntax

typedef existing_data_type new_name;

Syntax

using new_name = existing_data_type;

Example:

typedef int Marks;

Example

using Distance = float;

4.2 Variables in C++

In C++, a ***variable*** *is a* ***named memory location*** *used to store data that can change during the execution of a program. Every variable holds a value, and this value can be updated as the program runs. Variables allow programs to store information, perform calculations, and produce meaningful results.*

Before using a variable in C++, it must be ***declared with a specific data type****. The data type tells the compiler:*

- *What kind of data the variable will store*
- *How much memory should be allocated*
- *What operations are allowed on the variable*

Rules for Naming Variables (Identifiers) in C++

1. *An identifier must use only letters (A–Z or a–z), digits (0–9), and the underscore (_) character.*
2. *An identifier must start with a letter or an underscore and must not begin with a digit.*
3. *Digits can be used in an identifier, but only after the first character.*
4. *An identifier must not contain spaces, because spaces are used to separate tokens in a program.*
5. *Special characters such as @, #, %, &, *, and ! are not allowed in identifiers.*
6. *C++ keywords cannot be used as identifiers, since they have predefined meanings in the language.*
7. *Identifiers are case-sensitive, so uppercase and lowercase letters are treated as different characters.*
8. *Identifiers should be meaningful and descriptive so that the purpose of the variable is easy to understand.*
9. *Identifiers should not be excessively long, as very long names reduce program readability.*

10. *Identifiers should follow a consistent naming style throughout the program to improve clarity.*
11. *Identifiers using double underscore (__) or starting with an underscore followed by an uppercase letter should be avoided, as these are reserved for the compiler.*

Examples

- ***Valid identifiers:*** *age, totalMarks, _count, value1*
- ***Invalid identifiers:*** *2age, total marks, int, @value*
- ***int age;*** *// Declares an integer variable named 'age'*
- ***float price = 9.99;*** *// Declares a float variable named 'price' and initializes it to 9.99*

1. Local Variables

Local variables are variables declared inside a function or a block. They can be used only within that function or block. They are created when the function starts and are destroyed when the function ends, so their values exist only for a short time.

- *Declared inside a function or block*
- *Accessible only within that function or block*
- *Created when the function starts executing*
- *Destroyed when the function ends*
- *Values are temporary and not retained*

Example:

```
void login()
{
   int otp = 1234;   // local
variable
   cout << otp;
}
```

2. Global Variables

*Global variables are variables that are declared **outside all functions**. They can be accessed and used by **any function** in the program. Memory for a global variable is allocated once, and it remains available for the **entire execution of the program**, from start to end.*

- *Declared outside all functions*
- *Accessible by all functions*
- *Memory allocated only once*
- *Exist throughout the program execution*

Example:

```
int total;   // global variable
```

3. Static Variables

*Static variables are variables that are declared using the **static** keyword. They are usually declared inside a function, but unlike normal local variables, they **do not lose their value** when the function execution ends. A static variable is initialized only once, and it retains its value between multiple function calls. Although its scope is limited to the function in which it is declared, its lifetime extends throughout the entire execution of the program*

- *Declared using the `static` keyword*
- *Initialized **only once***

Example:

```
void count()
{
   static int c = 0;   // static
variable
}
```

- *Retain their value between function calls*
- *Scope is limited to the function/block*
- *Lifetime lasts for the **entire program execution***

4. Automatic Variables

*Automatic variables are variables that are declared **inside a function or a block without using any storage class keyword**. In C++, all local variables are automatic by default. These variables are created automatically when the function or block is entered and destroyed when the block is exited. Because of this, their lifetime is limited to the block in which they are declared, and their values do not remain after the execution of the block.*

- *Declared without any storage class keyword*
- *Local to the function or block*
- *Created when the block starts executing*
- *Destroyed when the block ends*
- *Value does not persist after execution*

Example:

```
void test()
{
   int a = 5;   // automatic variable
}
```

5. Register Variables

*Register variables are variables that are declared using the `register` keyword. They are used to request the compiler to store the variable in a **CPU register** rather than in main memory so that the variable can be accessed more quickly. Since CPU registers are very limited in number, the compiler may ignore this request if sufficient registers are not available.*
*Because register variables are stored in CPU registers, they **do not have a memory address**. Therefore, the address of a register variable cannot be accessed using pointers. Register variables are generally used for variables that are accessed repeatedly, such as loop counters, to improve*
execution speed

- *Declared using the `register` keyword*
- *Intended to be stored in CPU registers for faster access*
- *Memory address cannot be obtained*
- *Commonly used for frequently accessed variable*

Example:

```
void speed()
{
   register int i;
            }
```

Only for understanding, not for exams.

Example –Local Variables

*A local variable is like **the score you get in one match of a game**. That score matters only for that particular match. When the match ends, the score is cleared and a new match starts with a fresh score. In the same way, a local variable is*

Demonstration Example –Variables

```
#include <iostream>
using namespace std;
// Global variable
int totalStudents = 0;
void classCount()
{
   static int classVisits = 0;   // static variable
   int presentStudents = 30;     // local / automatic
variable
   register int i;               // register variable

   classVisits++;
   totalStudents += presentStudents;

   for(i = 1; i <= 3; i++)
   {
      cout << i << " ";
   }
}
int main()
{
   classCount();
   classCount();
   return 0;
}
```

Output:

When the program is executed, the `main()` function calls the `classCount()` function two times. Inside the `classCount()` function, a `for` loop uses the register variable `i` to print numbers from 1 to 3. Each time the function is called, the loop runs independently and prints the sequence `1 2 3`. Since the function is invoked twice, the same sequence is printed two times consecutively. Therefore, the final output of the program is **`1 2 3 1 2 3`**.

5. Operators in C++

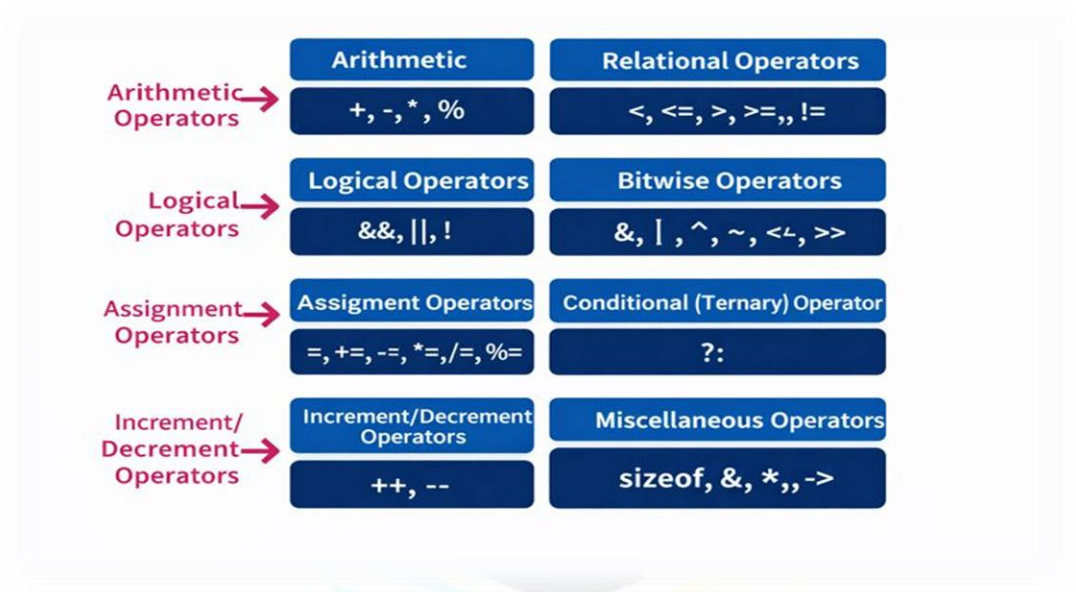
5.1 Operators in C++

An **operator** in C++ is a special symbol that performs a specific operation on one or more operands (variables or values). Operators are used to manipulate data, perform calculations, compare values, and make decisions in a program. C++ provides a wide variety of operators which are classified based on their functionality.

C++ provides a wide range of operators:

1. **Arithmetic Operators**
2. **Relational Operators**
3. **Logical Operators**
4. **Bitwise Operators**
5. **Assignment Operators**
6. **Conditional (Ternary) Operator**

7. ***Increment/Decrement Operators***
8. ***Miscellaneous Operators***



1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, division, and modulus. They operate on numeric data types like `int`, `float`, *and* `double`. *These operators are widely used in calculations and expressions in C++ programs.*

Operator	***Meaning***	***Example***
+	*Addition*	*a + b*
-	*Subtraction*	*a - b*
*	*Multiplication*	*a * b*
/	*Division*	*a / b*
%	*Modulus (remainder)*	*a % b*

Example:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 3;
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "a / b = " << a / b << endl;
    cout << "a % b = " << a % b << endl;
    return 0;
        }
```

2. Relational Operators

Relational operators are used to compare two values or expressions. The result of a relational operation is a Boolean value, either `true` or `false`. They are mainly used in conditional statements and loops for decision making

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

Example

```
#include <iostream>
using namespace std;
int main()
{
int a = 10, b = 5;
cout << (a > b) << endl;
cout << (a == b) << endl;
cout << (a != b) << endl;
return 0;
}
```

3. Logical Operators

Logical operators are used to combine two or more conditions or to reverse a condition. They return a Boolean result and are commonly used with relational expressions. Logical operators play an important role in complex decision-making processes.

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	`a || b
!	Logical NOT	!a

Example

```
#include <iostream>
using namespace std;
int main()
{
int a = 5, b = 10;
cout << (a < b && b > 0) << endl;
cout << (a > b || b > 0) << endl;
cout << !(a > b) << endl;
return 0;
}
```

4. Bitwise Operators

Bitwise operators perform operations directly on the binary representation of data. They are used for low-level programming tasks such as hardware control and memory optimization. These operators work bit by bit on integer data types.

Operator	Meaning	Example
&	Bitwise AND	a & b
`	`	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

Example

```
#include <iostream>
using namespace std;
int main()
{
   int a = 5, b = 3;
   cout << (a & b) << endl;
   cout << (a | b) << endl;
   cout << (a ^ b) << endl;
   return 0;
`
```

5. Assignment Operators

Assignment operators are used to assign values to variables. In addition to simple assignment, compound assignment operators combine arithmetic operations with assignment. They help in writing concise and efficient code.

Operator	Meaning	Example
=	Assignment	a = b
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

Example

```
#include <iostream>
using namespace std;
int main()
{
   int a = 10;
   a += 5;
   cout << a << endl;
   a *= 2;
   cout << a << endl;
         return 0;
      }
```

6. Conditional (Ternary) Operator

The conditional operator is the only ternary operator in C++. It evaluates a condition and returns one of two values based on the result. It provides a compact alternative to simple `if-else` *statements.*

Operator	Meaning	Example
?:	Conditional operator	(a > b) ? a : b

Example Program

```
#include <iostream>
using namespace std;
int main()
{
   int a = 10, b = 20;
   int max = (a > b) ? a : b;
   cout << max << endl;
   return 0;
}
```

7. Increment and Decrement Operators

Increment and decrement operators are used to increase or decrease the value of a variable by one. They can be used in prefix or postfix form. These operators are commonly used in loops and counters.

Operator	Meaning	Example
++	Increment	a++
--	Decrement	a--

Example Program

```
#include <iostream>
using namespace std;
int main()
{
   int a = 5;
   a++;
   cout << a << endl;
   a--;
   cout << a << endl;
   return 0;
}
```

8. Miscellaneous Operators

Miscellaneous operators perform special operations such as finding the size of a data type, accessing memory addresses, and working with pointers and objects. They provide additional functionality beyond basic arithmetic and logical operations. These operators are useful in advanced C++ programming.

Example Program

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

int main()
{
   int a = 10;
   cout << sizeof(a) <<
endl;
   cout << &a << endl;
   return 0;
}
```

Operator	Meaning	Example
sizeof	Size of data type	sizeof(int)
&	Address of variable	&a
*	Pointer dereference	*p
.	Member access	obj.x
->	Pointer member access	ptr->x

5.2 Type Compatibility

Type compatibility *in C++ refers to the ability of the compiler to* ***convert one data type into another*** *during an assignment or expression. When values of different data types are used together, C++ performs* ***type conversion*** *so that the operation can be carried out correctly. Type compatibility helps in maintaining flexibility while working with different data types.*
In C++, type compatibility is achieved in ***two ways****:*

1. ***Implicit type conversion***
2. ***Explicit type conversion (type casting)***

1. Implicit Type Conversion

Implicit type conversion is a process in which the ***compiler automatically changes one data type into another****. The programmer does not need to give any special instruction for this conversion. It usually occurs when two different data types are used together in an expression. To perform the operation correctly, the compiler converts the smaller data type into a larger or suitable data type.*

This type of conversion helps in avoiding errors and loss of data during calculations. For example, when an integer value is combined with a floating-point value, the integer is automatically converted into a floating-point type so that the result is accurate.

- *Done automatically by the compiler*
- *No explicit instruction is required*
- *Occurs when different data types are used together*
- *Smaller data types are converted into larger ones*

Example:

```
#include <iostream>
```

Rules of Implicit Conversion

From Type	To Type	Example
char	int	'A' → 65
int	float	10 → 10.0
float	double	3.5f → 3.5
Smaller type	Larger type	short → long

```
using namespace std;
int main()
{
   int a = 10;
   float b = 3.5;
   // Implicit type conversion
   float result = a + b;
   cout << "Value of a (int): " << a << endl;
   cout << "Value of b (float): " << b << endl;
   cout << "Result after addition: " << result << endl;
   return 0;
}
```

Output

Value of a (int): 10
Value of b (float): 3.5
Result after addition: 13.5

Understanding purpose:

*Implicit type conversion is like **pouring water from a small glass into a big bottle**. The water fits easily without spilling. Similarly, the compiler safely converts a smaller data type into a larger one automatically .*

2. Explicit Type Conversion (Type Casting)

*Explicit type conversion, also called **type casting**, is when the **programmer changes one data type into another by writing instructions in the code**. The compiler does not do this conversion automatically. The programmer clearly tells the program which data type is needed.*

*This type of conversion is mainly used when we want to **control the result**, especially when converting a bigger type (like `float`) into a smaller type (like `int`). During this conversion, some data (such as decimal values) may be lost.*

- *Explicit type conversion is performed by the programmer.*
- *It is also known as **type casting**.*
- *The programmer specifies the desired data type in the program.*
- *The compiler follows the conversion as instructed.*
- *This conversion may lead to loss of data, such as removal of the fractional part.*
- *It is used when precise control over data type conversion is required.*

Example Program (Explicit Conversion)

```
#include <iostream>
using namespace std;
int main()
{
   float x = 5.7;
   int y = (int)x;   // explicit type casting
   cout << "y = " << y << endl;
   return 0;
}
```

Rules of Explicit Conversion		
From Type	***To Type***	***Example***
float	*int*	*5.7 → 5*
double	*int*	*9.8 → 9*
Larger type	*Smaller type*	*long → int*

Output:
y = 5

Understanding purpose:
*Explicit type conversion is like **withdrawing cash from an ATM**. If your account balance is ₹2500.75, the ATM gives you only whole notes such as ₹2500 and ignores the remaining ₹0.75. This is a conscious decision made by the system. In the same way, in explicit type conversion, the programmer deliberately converts a value into another data type, even if some part of the value is discarded.*

5.3 Operator Precedence

*Operator precedence refers to the **order in which operators are evaluated** in an expression. When an expression contains more than one operator, the compiler follows a set of predefined rules to decide **which operation is performed first**. Operators with higher precedence are evaluated before operators with lower precedence.*

*If operators have the **same precedence**, then **associativity** (left to right or right to left) decides the order of evaluation. Understanding operator precedence helps in writing correct expressions and avoiding logical errors.*

Consider the expression: `5 + 3 * 2`

Without precedence rules, it could be evaluated as `(5 + 3) * 2 = 16` *or* `5 + (3 * 2) = 11`. *C++ follows the standard mathematical precedence, so* `3 * 2` *is evaluated first, resulting in* `5 + 6 = 11`.

1. Precedence Table

Precedence	Operators	Associativity	Description
1 (Highest)	() [] -> . ::	Left → Right	Parentheses, array subscript, member access, scope resolution
2	++ -- (postfix)	Left → Right	Postfix increment/decrement
3	++ -- + - ! ~ * & sizeof typeid new delete (unary)	Right → Left	Unary operators, dereference, address-of, object creation
4	* / %	Left → Right	Multiplication, division, modulus
5	+ -	Left → Right	Addition, subtraction
6	<< >>	Left → Right	Bitwise shift (left, right)
7	< <= > >=	Left → Right	Relational operators
8	== !=	Left → Right	Equality operators
9	&	Left → Right	Bitwise AND
10	^	Left → Right	Bitwise XOR
11	`	`	Left → Right
12	&&	Left → Right	Logical AND
13	`		`
14	?:	Right → Left	Conditional (ternary)
15	`= += -= *= /= %= <<= >>= &= ^=	=`	Right → Left
16 (Lowest)	,	Left → Right	Comma operator

Example 1: Arithmetic Precedence

```
#include <iostream>
using namespace std;
int main()
{
   int a = 10, b = 5, c = 2;
   int result = a + b * c; // * has higher precedence than +
   cout << "Result = " << result << endl; // 10 + (5*2) = 20
   return 0;
}
```

Example 2: Parentheses Override Precedence

```
int result2 = (a + b) * c; // Parentheses change order
```

```
cout << "Result2 = " << result2 << endl; // (10+5)*2 = 30
```

Output:

```
Result2 = 30
```

Example 3: Logical Operators

```
bool x = true, y = false, z = true;
bool result3 = x || y && z; // && has higher precedence than ||
cout << result3 << endl; // true
```

---

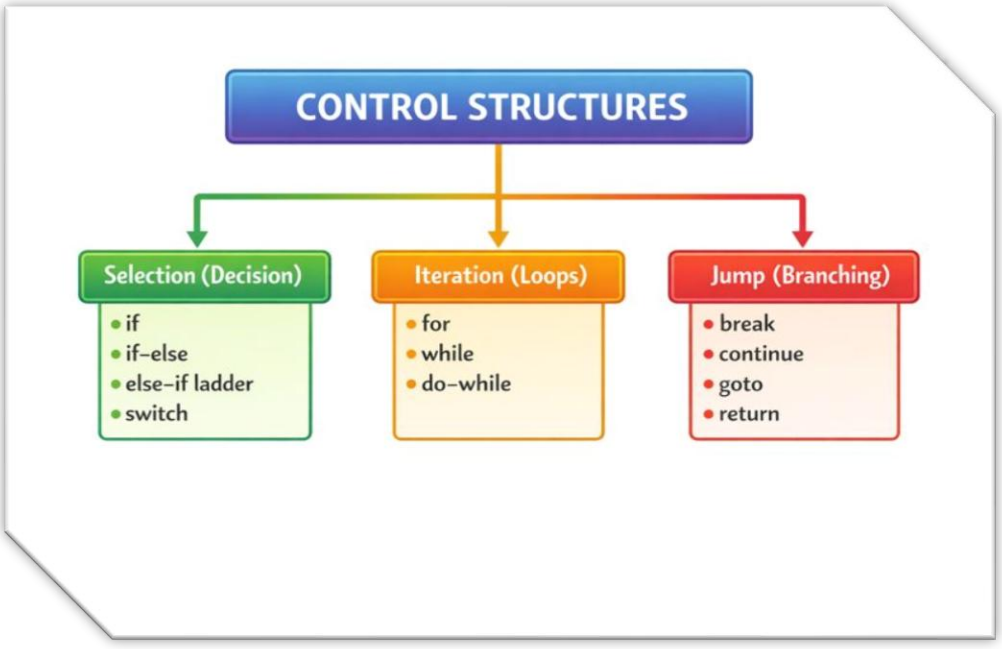
6. Control Structures

6.1 Control Structures

*Control structures in C++ are programming constructs that allow the programmer to **control the flow of execution** of a program. They determine the order in which statements are executed and enable decision making, repetition of statements, and branching based on conditions.*

*The **three main control structures** in C++ are:*

1. ***Selection (Decision-Making) Structures***
2. ***Iteration (Looping) Structures***
3. ***Jump (Branching) Structures***



6.2 Selection (Decision-Making) Structures

1. Selection (Decision-Making) Structures

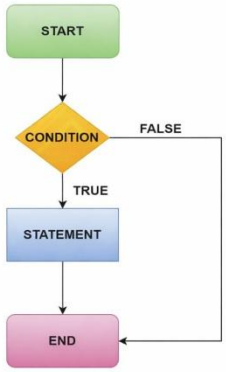
Decision-making (selection) statements *in C++ are used to* ***control the flow of execution*** *of a program based on a condition. These statements evaluate a given condition and execute specific blocks of code depending on whether the condition is* ***true or false****. They help a program make choices and perform different actions under different situations.*

Decision-making statements are an essential part of programming, as they allow the program to respond dynamically to input and logical conditions.

Types of Decision-Making / Selection Statements in C++:

- ***if***
- ***if–else***
- ***else–if ladder***
- ***switch.***

❖ if Statement



The ***simple `if` statement*** *is a decision-making statement used to execute a block of code* ***only when a specified condition is true****. The condition is evaluated first, and if it results in true, the statements inside the `if` block are executed. If the condition is false, the statements are skipped and control moves to the next statement in the program. The simple `if` statement does not contain an `else` part and is mainly used for checking single conditions.*

Syntax:

```
if(condition)
 {
   // code to execute
}
```

Example:

```
int x = 10;
if(x > 0)
 {
   cout << "x is positive";
}
```

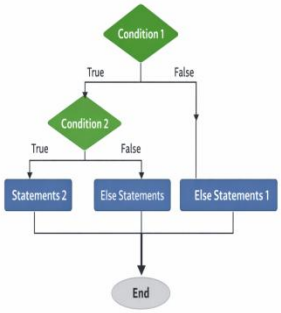
❖ if-else Statement

The ***if–else statement*** *is a decision-making statement used to execute* ***one block of code when a condition is true*** *and* ***another block when the condition is false****. It provides an alternative path of execution, ensuring that one of the two blocks is always executed. The if–else statement is commonly used when a program must choose between two different actions based on a condition.*

- ***If the condition is true**, one block executes.*
- ***If the condition is false**, another block executes.*

Syntax:

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



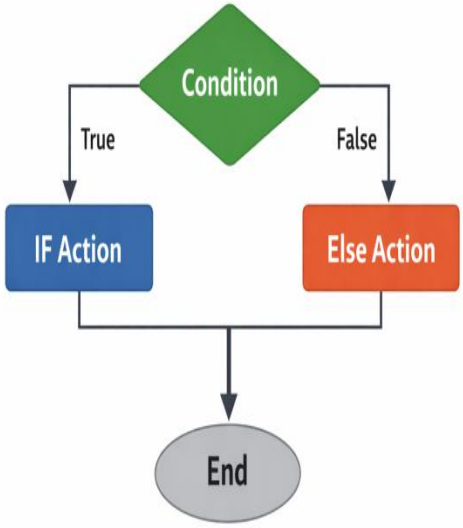
Example:

```
int x = -5;
if(x >= 0)
{
  cout << "Non-negative";
}
else
{
  cout << "Negative";
}
```

Output:Negative

3. else-if Ladder

*The **else–if ladder** is a decision-making statement used to test **multiple conditions** one after another. When a condition evaluates to true, the corresponding block of code is executed and the remaining conditions are skipped. If none of the conditions are true, the final else block (if present) is executed. The else–if ladder is useful when a program must choose **one option from several alternatives**.*



Step-by-Step Execution

1. ***Start from the top**: The program evaluates the first `if` condition.*
2. ***Check if true**:*

- If **true**, the code inside this `if` executes.
- The program **skips all remaining `else-if` and `else` blocks**.

3. **Check next condition**:
   - If the first `if` is false, the program checks the first `else-if` condition.
   - Repeat this for each `else-if` in the ladder.
4. **Execute `else` (optional)**:
   - If **none of the conditions are true**, the code inside the `else` block executes (if it exists).

Syntax:

```
If (condition1)
{
   // code1
}
 else if(condition2)
{
   // code2
}
else if(condition3)
{
   // code3
} else
 {
   // default code
}
```

Example:

```
#include <iostream>
using namespace std;
int main()
{
int marks = 82;
if(marks >= 90)
{
cout << "Grade A";
}
else if(marks >= 75)
{
cout << "Grade B";
}
else if(marks >= 50)
{
cout << "Grade C";
}
else
{
cout << "Grade F";
}
return 0;}
```

Step-by-Step Execution:

1. marks >= 90 → **82 >= 90?** → false → skip this block.
2. marks >= 75 → **82 >= 75?** → true → execute cout << "Grade B"
3. **Skip the rest** (marks >= 50 and else) → program continues after the ladder.

Output:
Grade B

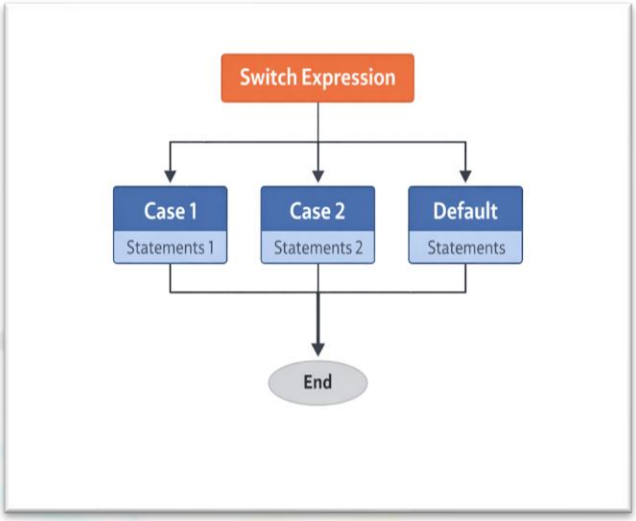
4. Switch Statement

*The **switch statement** is a decision-making statement used to execute one block of code from **multiple alternatives** based on the value of an expression. The expression is evaluated once and compared with different `case` values. When a match is found, the corresponding case block is executed until a `break` statement is encountered. The `default` case is executed if none of the case values match.*

Syntax:

```
switch(expression)
 {
   case value1:
      // code
      break;
   case value2:
      // code
      break;

 default:
      // default code
}
```



***Example**:*

```
#include <iostream>
using namespace std;
int main()
 {
   int choice;
   cout << "===== MENU =====" << endl;
   cout << "1. Tea" << endl;
   cout << "2. Coffee" << endl;
   cout << "3. Juice" << endl;
   cout << "4. Water" << endl;
   cout << "Enter your choice (1-4): ";
   cin >> choice;
   switch(choice)
 {
      case 1:
         cout << "You selected Tea ";
         break;
      case 2:
         cout << "You selected Coffee ";
         break;
```

OUTPUT:

```
===== MENU =====
1. Tea
2. Coffee
3. Juice

Enter your choice (1-4):
3
You selected Juice
```

```
        case 3:
            cout << "You selected Juice ";

            break;

        case 4:
            cout << "You selected Water ";
            break;
        default:
            cout << "Invalid choice!";
    }
 return 0;
}
```

6.3 Looping / Iteration Statements in C++

2. Looping / Iteration Statements in C++

*Looping or iteration statements in C++ are **control structures** that allow a block of code to be executed **repeatedly** either while a given condition remains true or for a fixed number of times. Loops help in reducing code repetition, improving program efficiency, and making programs easier to read and maintain.*

*C++ provides **three main types of loops**:*

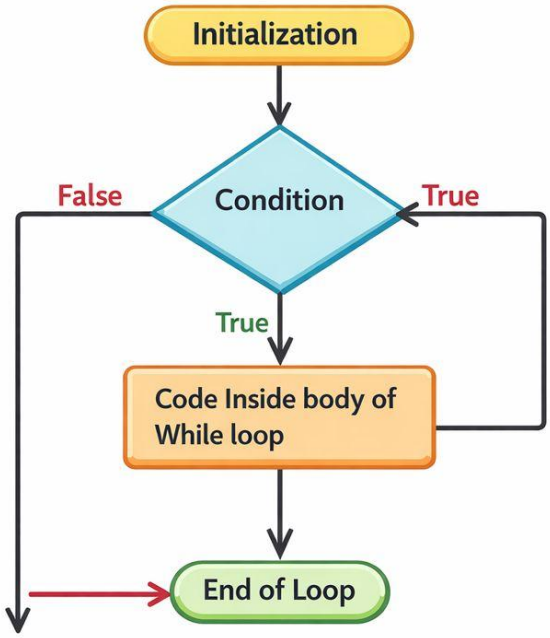
1. ***while loop***
2. ***do-while loop***
3. ***for loop***

1. while Loop: *The while **loop** in C++ is a looping statement that is used to execute a block of statements repeatedly as long as a specified condition is true. In this loop, the condition is evaluated before each iteration, and if it is true, the statements inside otherwise, the loop terminates. Since the condition is tested before entering the loop body, the while loop is known as an entry-controlled loop. It is generally used when the number of iterations is not known beforehand and depends on the given condition.*

Execution Process

1. ***Check Condition First***
   - *The `condition` is evaluated **before each iteration**.*
2. ***If Condition is TRUE***
   - *The loop body (statements inside `{ }`) executes.*
   - *After executing, control goes **back to check the condition again**.*

3. ***If Condition is FALSE***
    - *The loop stops immediately.*
    - *Control moves to the **next statement after the loop**.*



Syntax:

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

Example: Print Numbers from 1 to 5

```
#include <iostream>
using namespace std;
int main() {
   int i = 1;  // initialization

   while(i <= 5) {          // condition check
       cout << i << " ";     // loop body
       i++;                  // increment (update step)
   }
   return 0;
}
```

Step-by-Step Execution

- ***Iteration 1*** → *i = 1, condition 1 <= 5 → true → print 1, then i = 2*
- ***Iteration 2*** → *i = 2, condition 2 <= 5 → true → print 2, then i = 3*
- ***Iteration 3*** → *i = 3, condition 3 <= 5 → true → print 3, then i = 4*
- ***Iteration 4*** → *i = 4, condition 4 <= 5 → true → print 4, then i = 5*
- ***Iteration 5*** → *i = 5, condition 5 <= 5 → true → print 5, then i = 6*
- ***Iteration 6*** → *i = 6, condition 6 <= 5 → false → loop ends*

Output:

```
1 2 3 4 5
```

2. do-while Loop

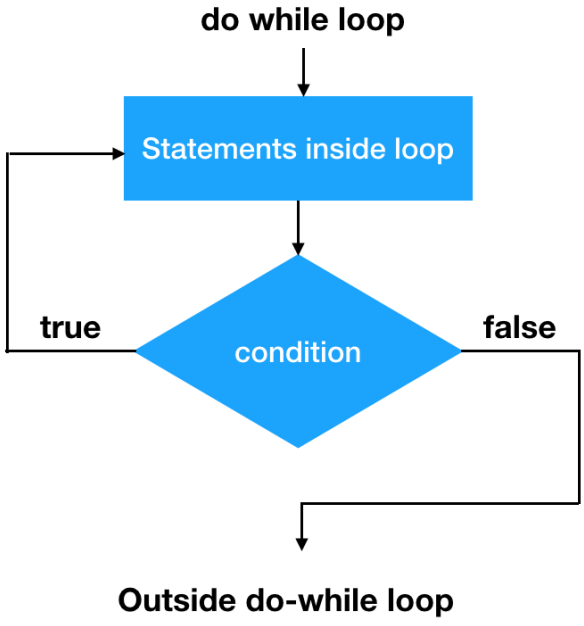
*The **do–while loop** is a looping statement that executes a block of code **at least once**, regardless of whether the condition is true or false. After executing the loop body, the condition is checked at the **end of the loop**, and the loop continues to repeat as long as the condition remains true .*

Execution Process

1. *The **loop body executes first**, **without checking the condition**.*
2. *After execution, the **condition is checked**.*
   - *If condition is **true**, the loop repeats.*
   - *If condition is **false**, the loop **terminates**.*
3. *This guarantees that the loop body executes **at least once**, even if the condition is false initially.*

Syntax:

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



Example:

```
#include <iostream>
using namespace std;
int main()
{
   int i = 1;

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

   return 0;
}
```

Step-by-Step Execution

- ***Iteration 1*** → *executes body first → print 1 → then check i <= 5 → true → continue*
- ***Iteration 2*** → *print 2 → check i <= 5 → true*
- ***Iteration 3*** → *print 3 → check i <= 5 → true*
- ***Iteration 4*** → *print 4 → check i <= 5 → true*
- ***Iteration 5*** → *print 5 → check i <= 5 → now i = 6 → condition false → stop*

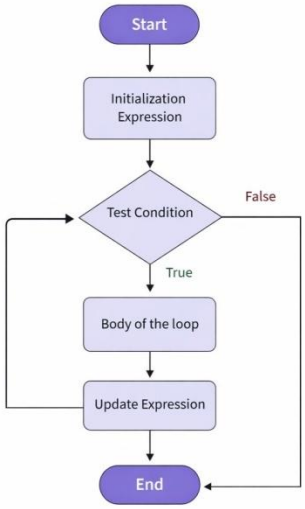
Output: 1 2 3 4 5

3. for Loop

*The **for loop** is a looping statement that repeats a block of code for a **known or fixed number of iterations**. It includes initialization, condition checking, and increment or decrement in a single statement. The loop continues to execute as long as the condition remains true.*

Execution Process

1. ***Initialization***
   - *This step runs **only once** at the beginning (e.g., `int i = 1;`).*
2. ***Condition Check***
   - *If the condition is **true**, the loop body executes.*
   - *If the condition is **false**, the loop terminates immediately.*
3. ***Loop Body Execution***
   - *Statements inside `{ }` execute.*
4. ***Update***
   - *After the body executes, the **update expression** runs (e.g., `i++`).*
5. ***Repeat***
   - *Control goes back to the **condition check**, and steps 2–4 repeat until the condition becomes false.*



Syntax:

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

Example: Print Numbers from 1 to 5

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

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

Step-by-Step Execution

Initialization: *int i = 1 (happens once)*
Iteration 1: *condition i <= 5 → true → print 1 → update i = 2*
Iteration 2: *condition 2 <= 5 → true → print 2 → update i = 3*
Iteration 3: *condition 3 <= 5 → true → print 3 → update i = 4*
Iteration 4: *condition 4 <= 5 → true → print 4 → update i = 5*
Iteration 5: *condition 5 <= 5 → true → print 5 → update i = 6*
Iteration 6: *condition 6 <= 5 → false → loop ends*
Output: 1 2 3 4 5

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

6.5 Difference between Loops:-

Basis of Comparison	for Loop	while Loop	do-while Loop
Meaning	Used to repeat a block of code a fixed number of times	Used to repeat statements while a condition remains true	Used to repeat statements at least once and then based on a condition
Condition checking	Checked before entering the loop	Checked before entering the loop	Checked after executing the loop body
Type of loop	Entry-controlled loop	Entry-controlled loop	Exit-controlled loop
Number of iterations	Known in advance	Not known in advance	Not known, but executes once
Initialization	Done in the loop statement	Done before the loop	Done before the loop
Update of variable	Done in the loop statement	Done inside loop body	Done inside loop body
Execution at least once	No	No	Yes
Code structure	Compact and well-organized	Simple but less compact	Slightly longer syntax
Common use	Counting, tables, arrays	Condition-based repetition	Menu-driven programs

6.6 Loop Control Statements /Jump Statements

*Loop control statements are used to **alter the normal flow of execution** inside looping and selection structures. When program control leaves a particular block or scope, all automatic variables created within that scope are destroyed. In C++, loop control statements help the programmer **exit a loop, skip certain statements, or jump to another part of the program**. The main loop control statements supported by C++ are **break, continue, and goto.***

1. Break Statement

*The `break` **statement** is used to **immediately terminate the execution of a loop or a switch statement**. When a `break` statement is encountered inside a loop, control is transferred to the statement that follows the loop. In the case of nested loops, the `break` statement only terminates the **innermost loop** in which it appears. It is also commonly used in `switch` statements to end the execution of a particular case.*

Syntax:

```
break;
```

When the program encounters `break`, it stops the current loop execution instantly, even if the loop condition is still true, and continues execution with the next statement after the loop.

```
#include <iostream>
using namespace std;
int main()
{
   int num = 1;
   while(num <= 10)
   {
      if(num == 7)
      {
         break;   // exits the loop when num becomes 7
      }
      cout << num << " ";
      num++;
   }
   return 0;
}
```

Output :1 2 3 4 5 6

In this program, the `while` loop starts executing with the value of `num` equal to 1. During each iteration, the value of `num` is printed and then increased by 1. When `num` becomes 7, the condition inside the `if` statement becomes true, and the `break` statement is executed. This causes the loop to terminate immediately, and the program stops printing further numbers. Hence, only numbers from 1 to 6 are displayed.

2. Continue Statement

*The **`continue` statement** is used to **skip the remaining statements of the current loop iteration** and move directly to the next iteration. Unlike `break`, it does not terminate the loop; instead, it forces the loop to proceed with the next cycle. In a `for` loop, control moves to the update and condition-checking part, while in `while` and `do-while` loops, control jumps directly to the condition test.*

Syntax:

```
continue;
```

When continue is executed, the statements following it in the loop body are skipped for that iteration, and the loop condition is checked again to decide whether the next iteration should run.

Example

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

int main()
{
   int i;
   for(i = 1; i <= 10; i++)
   {
      if(i == 5)
      {
         continue;   // skips printing when i is 5
      }
      cout << i << " ";
   }
   return 0;
}
```

Output: 1 2 3 4 6 7 8 9 10

In this program, the for loop runs from 1 to 10. When the value of i becomes 5, the condition inside the if statement is true, and the continue statement is executed. As a result, the cout statement is skipped for that iteration, and the loop immediately moves to the next iteration. Therefore, the value 5 is not printed, while all other numbers are displayed.

3. Goto Statement

The `goto` ***statement*** *is an unconditional control transfer statement that moves program execution to a* ***labeled statement*** *within the same function. Although it can simplify exiting from deeply nested loops, its use is generally discouraged because it makes programs difficult to read, understand, and maintain. Most programs that use* `goto` *can be rewritten using loops and conditional statements instead.*

Syntax:

```
goto label name;
...
Label name: statement;
```

When a `goto` statement is executed, control jumps directly to the labeled statement, bypassing the normal sequential execution of the program.

```
#include <iostream>
using namespace std;
int main()
{
   int i = 1;

START:
   if(i <= 5)
   {
      cout << i << " ";
      i++;
      goto START;   // jumps back to START label
   }
   return 0;
}
```

Output:*1 2 3 4 5*

In this program, the execution begins with the value of i equal to 1. The label START marks a position in the program. Each time the condition i <= 5 is true, the value of i is printed and incremented. The goto statement then transfers control back to the START label, causing the statements to repeat. When the condition becomes false, the program exits normally.

6.7 Difference Between break and continue

Basis	break Statement	continue Statement
Definition	The break statement is used to terminate the execution of a loop immediately.	The continue statement is used to skip the remaining statements of the current loop iteration.
Effect on loop	Causes complete termination of the loop.	Does not terminate the loop; control moves to the next iteration.
Flow of control	Transfers control to the statement following the loop.	Transfers control to the next iteration of the loop.
Condition checking	Loop condition is not checked again after break.	Loop condition is checked again after continue.
Use in nested loops	Terminates only the innermost loop.	Skips the current iteration of the innermost loop.
Use in switch	Used to terminate a case in a switch statement.	Not applicable in switch statements.
Purpose	Used when an immediate exit from the loop is required.	Used when certain iterations of the loop need to be skipped.

6.8 Conditional (Ternary) Operator in C++

*The **conditional operator** (also called the **ternary operator**) is a shorthand form of the if-else statement in C++.It uses the symbol ?: and evaluates a condition, returning one of two values depending on whether the condition is **true** or **false**.*

***Syntax**:*

condition ? expression1 : expression2;

Condition → a boolean expression (evaluated as true or false)
*If condition is **true** → expression1 executes (or value is chosen)*
*If condition is **false** → expression2 executes (or value is chosen)*

***Example** :*

```
#include <iostream>
using namespace std;
int main()
{
   int a = 10, b = 20;
// checks condition (a > b)
   int max = (a > b) ? a : b;
   cout << "Maximum is: " << max;
   return 0;
}
```

Execution:

- Condition a > b → 10 > 20 → false
- So, max = b → 20

Output:

Maximum is: 20

Long Questions

1. *Explain the Object Oriented Programming paradigm. Compare it with procedure-oriented programming.*
2. *Explain the basic concepts of Object Oriented Programming (Class, Object, Encapsulation, Abstraction, Inheritance, Polymorphism).*
3. *What is Encapsulation? Explain its advantages and implementation in C++.*
4. *Explain Inheritance in C++. Discuss different types of inheritance with examples.*
5. *Define Polymorphism. Explain compile-time and run-time polymorphism with examples.*
6. *Explain the structure of a C++ program with a neat diagram.*
7. *What are tokens in C++? Explain keywords, identifiers, constants, and operators.*
8. *Explain data types in C++. Classify them and explain with examples.*
9. *Explain various operators in C++.*

10. *What is operator overloading?*
    Explain rules, advantages, and limitations.
11. *Explain control structures in C++.*
    Discuss selection, iteration, and jump statements.
12. *Explain benefits and applications of Object Oriented Programming.*

Short Questions

1. *Define a class and object.*
2. *What is encapsulation?*
3. *What is abstraction?*
4. *Define inheritance.*
5. *What is polymorphism?*
6. *What is dynamic binding?*
7. *What is message passing?*
8. *List any four advantages of Object Oriented Programming.*
9. *List any four Object Oriented Programming languages.*
10. *Who developed C++?*
11. *Is C++ a pure object-oriented language? Why?*
12. *Write the structure of a C++ program.*
13. *What is the role of the `main()` function?*
14. *What is a namespace?*
15. *Define tokens in C++.*
16. *What are data types?*
17. *What are user-defined data types?*
18. *What is operator overloading?*
19. *What are control structures?*

UNIT –II

1. Functions in C++

*A **function** in C++ is a named block of statements that performs a specific task. Functions help in dividing a large program into smaller, manageable units. Each function is designed to perform a single, well-defined operation, which makes the program easier to understand, write, test, and maintain.*

*In C++, a program may need to perform the same operation multiple times. Writing the same code repeatedly increases program size and makes maintenance difficult. To overcome this problem, C++ uses **functions**. A function is written once and can be executed many times whenever it is required by calling it.*

*Functions support the idea of **modularity**, where a complex problem is broken into simpler sub-problems. Each sub-problem is solved using a separate function.*

*A function is a **self-contained unit of program code** that performs a particular task when it is called.*

1.1 Structure of a Function in C++

*A **function in C++** is a well-organized unit of code that performs a specific task. For proper execution and understanding, every function follows a defined structure. Generally, a C++ function consists of **three main parts**: **function declaration (prototype)**, **function definition**, and **function call**. Each part plays a crucial role in the functioning of a program.*

```
// Function declaration (prototype)
return_type function_name(parameter_list);

int main()
{
   // Function call
   function_name(arguments);
   return 0;
}

// Function definition
return_type function_name(parameter_list)
{
   // function body
   return value;   // if return type is not void
}
```

1. Function Declaration (Function Prototype)

*The **function declaration**, also known as the **function prototype**, introduces the function to the compiler before its actual use in the program. It specifies the **function name**, **return type**, and **number and type of parameters**. This information enables the compiler to perform type checking and ensure that the function is called correctly.*

*The function declaration does not contain the function body. It only tells the compiler **what the function looks like**, not **how it works**. This is especially important when the function is defined after the `main()` function or in a separate file.*

Key points:

- *Helps in error detection at compile time*
- *Ensures correct usage of function arguments*
- *Improves program clarity and organization*

2. Function Definition

*The **function definition** contains the actual code that performs the intended task. It defines **how the function works**. This part includes the function header and a body enclosed within braces.*

*When a function is defined, memory is allocated for its local variables, and the statements inside the function body are executed whenever the function is called. The function definition may include a **return statement** to send a value back to the calling function.*

Key points:

- *Contains executable statements*
- *Implements the logic of the function*
- *May return a value or be of type `void`*
- *Local variables exist only within the function*

3. Function Call

*A **function call** is a statement that invokes the execution of a function. When a function call is made, control is transferred from the calling function (usually `main()`) to the called function. The actual values passed during the call are known as **arguments**.*

After the function completes its execution, control returns to the statement immediately following the function call. If the function returns a value, that value is passed back to the calling function and can be stored or used in expressions.

Key points:

- *Activates the function execution*
- *Transfers control between functions*
- *Allows reuse of function logic*
- *Supports modular programming*

1.2 Types of Functions in C++

*Functions in C++ are classified according to their **source and manner of use**. On this basis, functions are broadly divided into **library functions** and **user-defined functions**. This classification helps programmers select appropriate functions for efficient program design*

C++ supports two main types of functions:

1. ***Built-in (library) functions***
2. ***User-defined functions***

2. Built-in (library) functions

***Library functions**, also called **built-in functions**, are predefined functions provided by the C++ Standard Library. These functions are already written, tested, and optimized, and they are used to perform common and frequently required operations in a program.*

Explanation

*Library functions are readily available to the programmer and can be used by including the appropriate **header files** in the program. The internal working of these functions is hidden from the user, which allows programmers to use them without knowing their implementation details.*

Features of Library Functions

- *Predefined and built into C++ libraries*
- *Easy to use and reliable*
- *Improve programming efficiency*
- *Reduce program length*
- *Tested for correctness and performance*

Common Header Files and Examples

<iostream> ⟶ cin, cout
<cmath> ⟶ sqrt(), pow()
<cstring> ⟶ strlen(), strcpy()
<cstdlib> → rand(), exit()

Example:

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

```
int main()
{
   double x = 16.0;
   cout << "Square root of " << x << " = " << sqrt(x);
   return 0;
}
```

Output:

Square root of 16 = 4

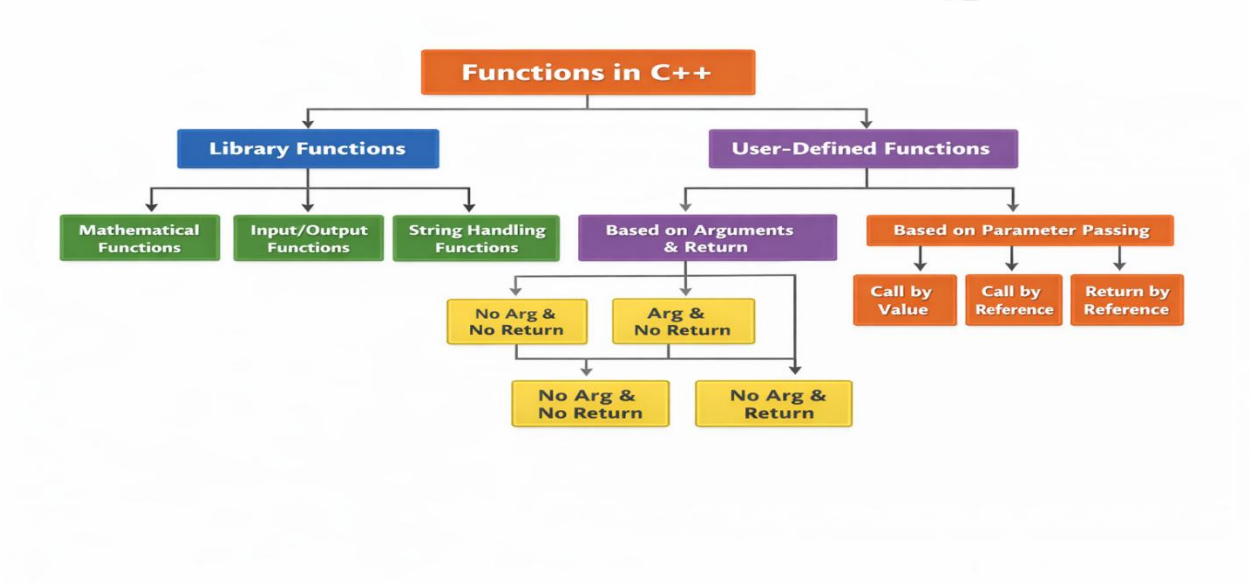
3. User-Defined Functions

User-defined functions *are functions that are* ***created by the programmer*** *to perform specific operations according to the requirements of a program. Unlike library functions, which are already provided by C++, user-defined functions allow programmers to design their own logic and control program behavior. They play a key role in* ***modular programming*** *and help in building large and complex programs in an organized manner.*

Need for User-Defined Functions

In practical programming, a task often needs to be performed repeatedly. Writing the same code again and again increases program length and makes it difficult to maintain. User-defined functions solve this problem by allowing the programmer to write the logic once and reuse it multiple times.

User-defined functions also make programs easier to read and understand, since each function performs a single, well-defined task.



1.2.1 Classification of User-defined Functions

1. Based on Arguments & Return Type

1. No arguments, No return
2. Arguments, No return
3. No arguments, With return
4. Arguments, With return

1. No arguments, No return:

A **no-argument, no-return value function** is a function that does not receive any data from the calling function and does not return any value after execution. Such functions are mainly used to perform a specific task like displaying messages or performing fixed operations.

Syntax :

```
void functionName()
{
   // Task (statements to execute)
}

int main() {
   functionName();  // Function Call
   return 0;
}
```

- The function does **not take parameters**, so no input is passed.
- The return type is **void**, which means it does not send any result back to the calling function.
- All operations are performed within the function itself.

Example:

```
void greet()
 {
   cout << "Hello, World!" << endl;
}
int main()
{
   greet();  // Function Call
   return 0;
}
```

Output: Hello, World!

2. Arguments, No return

A **function with arguments and no return value** is a function that receives data from the calling function in the form of parameters but does not return any value after execution. The function

performs operations using the passed arguments and usually displays the result within the function itself.

Syntax:

```
void functionName(type1 param1, type2 param2)
{
   // Task using arguments, no return
}
int main()
{
   functionName(value1, value2);   // Call with arguments
}
```

- *The function **accepts one or more arguments** from the calling function.*
- *The return type is **void**, so no value is sent back to the caller.*
- *Results are handled or displayed inside the function.*

Example:

```
#include <iostream>
using namespace std;
void printSum(int a, int b)
{
   cout << "Sum = " << a + b << endl;
}
int main() {
   printSum(5, 10);  // Function Call
   return 0;
}
```

***Output**:*

Sum = 15

3. No Arguments, With Return

*A **function with no arguments but with a return value** is a function that does not receive any input from the calling function, but after performing its operation, it returns a value to the calling function. The returned value can then be used for further processing or display.*

Syntax:

```
returnType functionName()
{
   // Task
   return value;   // Returning result
}
int main()
```

```
{
   returnType result = functionName();  // Call
}
```

- *The function **does not accept parameters**, so it works independently of the caller's input.*
- *The function **returns a value** using the return statement.*
- *The returned value is stored or used in the calling function.*

Example:

```
#include <iostream>
using namespace std;
int getNumber()
 {
   return 100;
}

int main()
{
   int num = getNumber();  // Function Call
   cout << "Number = " << num << endl;
   return 0;
}
```

***Output**:*

Number = 100

4. Arguments, With Return

*A **function with arguments and a return value** is a function that receives data from the calling function through parameters, processes the data, and returns the result to the calling function. This type of function is the most commonly used in C++ programming.*

Syntax:

```
returnType functionName (parameter1, parameter2, ...)
{
   // Statements / Computation
   return value;   // return the result
}
int main()
 {
   // Function Call
   returnType result = functionName(arg1, arg2, ...);
}
```

- *The function **accepts one or more arguments** as input.*

- *It **processes the input data** within the function body.*
- *The function **returns a value** to the calling function using the return statement.*
- *The returned value can be stored, displayed, or used for further computation.*

Example:

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

int add(int a, int b)
{
   return a + b;
}
int main()
{
   int sum = add(5, 10);  // Function Call
   cout << "Sum = " << sum << endl;
   return 0;
}
```

Output:

Sum = 15

1.2.2 Differences between Library Functions and User defined Functions

Library Functions	User-Defined Functions
Library functions are predefined functions provided by the C++ standard libraries.	*User-defined functions are functions written by the programmer to perform specific tasks.*
These functions are readily available and can be used by including the required header files.	*These functions are not built-in and must be explicitly defined by the user in the program.*
The internal logic of library functions is already implemented and hidden from the user.	*The internal logic of user-defined functions is written and controlled by the programmer.*
They are mainly used to perform common and standard operations such as input/output and mathematical calculations.	*They are used when a required operation is not available as a library function.*
Examples include cout, cin, sqrt(), strlen(), pow(), etc.	*Examples include add(), multiply(), isEven(), factorial(), etc.*
The programmer cannot modify the source code of library functions.	*The programmer can modify the function logic whenever required.*
They require appropriate header files such as <iostream>, <cmath>, <cstring>.	*They do not require external header files unless functions are defined in separate files.*
The programmer has limited control over the internal working of these functions.	*The programmer has complete control over the design and working of these*

	functions.

1.2.3 Based on Parameter Passing

Parameter Passing Methods in C++

1. ***Call by Value***
2. ***Call by Reference***

1. Call by Value

Call by value *is a method of passing arguments to a function in which the* ***actual value of the argument is copied*** *into the formal parameter of the function. In this method, the function works only on the copied data, not on the original variables. Therefore,* ***any changes made inside the function do not affect the actual arguments*** *used in the function call.*

By default, C++ uses call by value for passing arguments. Since the original data remains safe, this method is simple and secure, but it cannot be used when we want the function to modify the original values.

Key Points

- *A copy of the actual value is passed to the function*
- *Original variables remain unchanged*
- *Changes inside the function do not reflect outside*
- *Default method of parameter passing in C++.*

Example Program

```
#include <iostream>
using namespace std;
void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
int main()
{
   int a = 10, b = 20;

   cout << "Before swap: a = " << a << ", b = " << b << endl;
   swap(a, b);
```

```
    cout << "After swap: a = " << a << ", b = " << b << endl;
    return 0;
}
```

Output:
Before swap: a = 10, b = 20
After swap: a = 10, b = 20

In this program, the values of `a` and `b` are passed to the function as copies. The swapping occurs only within the function using these copied values. Since the original variables are not affected, the values of `a` and `b` remain the same after the function call.

Example2:

```
#include <iostream>
using namespace std;
void modify(int x) {
    x = x + 10;   // only local copy modified
}
int main() {
    int a = 5;
    modify(a);
    cout << "a = " << a;   // remains 5
    return 0;
}
```

Explanation :

1. *The variable **`a`** is passed to the function using **call by value**.*
2. *A **copy of `a`** is stored in the parameter **`x`**.*
3. *Any modification made to **`x`** affects **only the local copy**, not the original variable.*
4. *The original variable **`a`** **remains unchanged** after the function call.*
5. *Therefore, the output printed is **`a = 5`**.*

Conclusion: *Original variable is **unchanged**.*

2. Call by Reference

***Call by reference** is a method of passing arguments to a function in which the **reference (alias) of the actual argument** is passed to the formal parameter. In this method, the formal parameter becomes another name for the original variable. Therefore, **any change made inside the function directly affects the original variable**.*

This method does not create a copy of data, which makes it more efficient than call by value. It is commonly used when a function needs to modify the actual arguments.

Key Points

- *Reference of the variable is passed to the function*
- *No copying of data takes place*
- *Changes inside the function affect the original variables*
- *Improves performance and efficiency*

Example Program

```
#include <iostream>
using namespace std;
void swap(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
int main()
{
   int a = 10, b = 20;
   cout << "Before swap: a = " << a << ", b = " << b << endl;
   swap(a, b);
   cout << "After swap: a = " << a << ", b = " << b << endl;
   return 0;
}
```

Output

Before swap: a = 10, b = 20

After swap: a = 20, b = 10

In this program, the variables `a` and `b` are passed by reference to the `swap()` function. The function works directly on the original variables, so the changes made inside the function are reflected outside the function.

Example2:

```
#include <iostream>
using namespace std;
void modify(int &x)
 {
   x = x + 10;   // modifies original variable
}
int main()
 {
```

```
    int a = 5;
    modify(a);
    cout << "a = " << a;   // becomes 15
    return 0;
}
```

Explanation:

The variable `a` *is passed to the function using* ***call by reference****.*

1. *The parameter* `x` *becomes a* ***reference (alias)*** *to the original variable* `a`*.*
2. *Any modification made to* `x` *directly affects the original variable* `a`*.*
3. *Inside the function,* ***10 is added to*** `x`*, so the original value of* `a` *is updated.*
4. *After the function call, the value of* `a` ***becomes 15****.*
5. *Therefore, the output printed is* `a = 15`*.*

Conclusion: *Original variable* ***is changed.***

3. Return by Reference

Call by reference passes the memory address of a variable to a function. This allows the function to modify the original variable directly. Changes made to the parameter within the function will be reflected in the original variable outside the function.

- *In Return by Reference, a function* ***returns a reference to a variable*** *instead of returning a copy.*
- *This allows the returned reference to be used on the* ***left-hand side of an assignment****, and changes directly affect the original variable.*
- *Important: Function should return a reference to a variable that still exists (like global/static/array elements), not a local variable.*

Example:

```
#include <iostream>
using namespace std;
int& getElement(int arr[], int index)
 {
    return arr[index];   // return reference
}
int main()
 {
    int nums[3] = {10, 20, 30};
    getElement(nums, 1) = 50;  // modifies nums[1]
    cout << nums[1];    // prints 50
    return 0;
}
```

Conclusion: *Returned reference can be used to* ***modify original variable****.*

2. Special Functions in C++

2.1 Friend Functions

In C++, a ***friend function*** *is a function that is* ***not a member of a class****, but it is allowed to* ***access the private and protected members*** *of that class. Normally, private and protected members cannot be accessed from outside the class. However, when a function is declared as a* ***friend****, the class gives special permission to that function to access its internal data.*

A friend function is ***declared inside the class*** *using the keyword* `friend`*, but it is* ***defined outside the class****, like a normal function. Friendship is given explicitly by the class and cannot be forced by any external function.*

A ***friend function*** *is a non-member function that is allowed to access the private and protected members of a class by using the* `friend` *keyword.*

Key Points

- *A friend function is* ***not a member of a class****, but it is allowed to* ***access the private and protected members*** *of the class in which it is declared as a friend.*
- *Declaring a function as a friend* ***overrides the normal access restrictions*** *of a class.*
- *Normally, private and protected members of a class* ***cannot be accessed from outside*** *the class.*
- *A friend function can access these members* ***directly****, even though it is* ***defined outside the class****.*
- *Friendship is* ***granted by the class****, not taken by the function; the class decides which functions are friends.*
- *A friend function is* ***not part of the class****, therefore it* ***does not have a*** `this` ***pointer****.*
- *A friend can be a* ***normal function****, a* ***member function of another class****, or an* ***operator function****.*
- *Friendship is* ***not inherited****, so derived classes do not automatically get access to friends.*
- *Friend functions are* ***commonly used in operator overloading****, especially for binary operators.*
- *They are useful when a* ***single function needs access to private data of multiple classes****..*

Syntax:

```
class ClassName
 {
   private:
```

```
    int data;
  public:
    friend return Type function Name(parameters);
};
```

Advantages of Friend Functions

1. *Friend functions can access the **private and protected members** of a class, which is not possible with normal external functions.*
2. *They are useful when **two or more classes need to share data** and work closely together.*
3. *Friend functions make **operator overloading** (especially binary operators) easier and more flexible.*
4. *They allow functions to work with class data **without making the data public**, thus maintaining partial data hiding.*
5. *Friend functions can improve **program efficiency** by allowing direct access to data members.*

Disadvantages of Friend Functions

1. *Friend functions **break the concept of data hiding**, which is a key principle of object-oriented programming.*
2. *Excessive use of friend functions can make programs **less secure and harder to maintain**.*
3. *Since friend functions are not members of a class, they **do not have a `this` pointer**.*
4. *Friendship is **not inherited**, which can limit flexibility in derived classes.*
5. *Overuse of friend functions can lead to **poor program design** and tightly coupled code*

Example

```
#include <iostream>
using namespace std;
class Rectangle
{
   int length;   // private data member
public:
   void setLength(int l)
   {
      length = l;
   }
   friend void display(Rectangle r); // friend function declaration
};

// friend function definition
void display(Rectangle r)
```

```
{
   cout << "Length of rectangle is: " << r.length << endl;
}
int main()
{
   Rectangle r1;
   r1.setLength(10);
   display(r1);   // calling friend function
   return 0;
}
```

Output

Length of rectangle is: 10

In this program, the data member `length` is private and cannot be accessed directly outside the class. The function `display()` is declared as a **friend** inside the class `Rectangle`. Although `display()` is not a member function, it can still access the private data member `length`. The function is called like a normal function and prints the value successfully.

Example2:

```
#include <iostream>
using namespace std;
class Number
{
private:
   int value;
public:
   Number(int v) { value = v; }
   // Friend function declaration
   friend int add(Number n1, Number n2);
};
// Friend function definition
int add(Number n1, Number n2)
{
   return n1.value + n2.value;  // Can access private members
}
int main()
{
   Number n1(10), n2(20);
   int sum = add(n1, n2);  // Call friend function
```

```
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Output::30

2.2 Inline Functions

*An **inline function** is a function for which the compiler tries to insert the complete function body at every place the function is called.Because the function call is replaced by the function code itself, the overhead of calling and returning from a function is reduced.Inline functions are defined using the keyword* `inline`.

Necessity of Inline Functions

*One of the main purposes of using functions is **code reusability** and **better memory management**. However, when a function is called repeatedly, certain overheads are involved:*

- *Each function call requires extra time for:*
  - *Jumping to the function definition*
  - *Saving registers*
  - *Passing arguments*
  - *Returning control to the calling function*
- *If a function is **very small**, a large portion of execution time may be wasted in these overhead operations rather than doing actual work.*
- *In C, this problem was often handled using **macros**.*
  However, macros:
  - *Do not perform type checking*
  - *Do not follow function rules*
  - *Can lead to unexpected errors*
- *To overcome these drawbacks, **C++ introduces inline functions**, which behave like normal functions but avoid call overhead when possible.*

General Form of Inline Function

```
inline return_type function_name(parameter_list)
{
  // function body
}
```

Syntax

```
inline returnType  functionName(parameter_list)
{
```

```
    // function body
}
```

Example

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

inline int cube(int n)
{
   return n * n * n;   // short function, expanded at call site
}
int main()
{
   cout << "Cube of 3 = " << cube(3) << endl;
   cout << "Cube of 5 = " << cube(5) << endl;
   return 0;
}
```

Output:

Cube of 3 = 27
Cube of 5 = 125

Advantages of Inline Functions

1. ***Faster execution*** *→ avoids function call overhead.*
2. ***Improves performance*** *in short, repeated calculations.*
3. ***Increases readability*** *and modularity while still keeping efficiency.*
4. ***Helps in code reusability*** *without slowing execution.*

Limitations

- *Not suitable for* ***large/complex functions*** *(increases code size).*
- *If used inside loops → can cause* ***code bloat****.*
- *Compiler may* ***ignore*** `inline` ***request*** *(it's only a suggestion).*

2.3 Polymorphism in C++

Polymorphism *is one of the core concepts of* ***Object-Oriented Programming (OOP)****. The word polymorphism means* ***"many forms."*** *In C++, polymorphism allows* ***the same function name or operator to perform different tasks depending on the context****.*

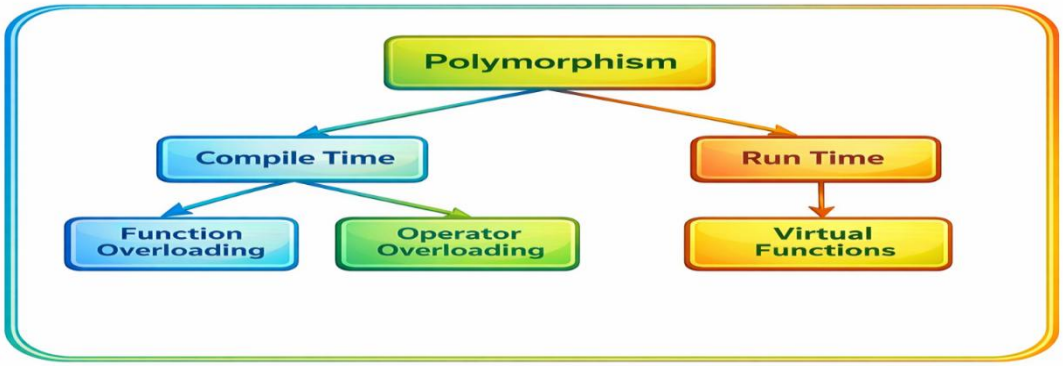
In simple words, ***one interface with multiple implementations*** *is called polymorphism.*

Types of Polymorphism in C++

Polymorphism is mainly classified into ***two types****:*

1. Compile-Time Polymorphism (Static Polymorphism)

2. Run-Time Polymorphism (Dynamic Polymorphism)



2.3.1 Compile-Time Polymorphism (Static Polymorphism)

I. Compile-Time Polymorphism (Static Polymorphism)

Compile-time polymorphism, also known as static polymorphism or early binding, is a type of polymorphism where the specific function or operator to be executed is determined at compile time. This means the compiler knows exactly which function will be called before the program is run. Compile-time polymorphism is primarily achieved through two mechanisms:

1. ***Function overloading***
2. ***Operator overloading.***

1. Function Overloading

Function overloading allows you to define multiple functions with the same name but different parameter lists (different number of parameters, different types of parameters, or different order of parameters) within the same scope. The compiler determines which function to call based on the arguments passed during the function call.

Need for Function Overloading

*In many programs, similar operations need to be performed on **different types of data** or with **different numbers of inputs**. Instead of creating different function names, function overloading allows the use of **a single meaningful name**, improving **code readability and maintainability**.*

Function Overloading Works

When a function is called, the compiler:

1. *Checks the **function name**.*
2. *Matches the **number and type of arguments**.*
3. *Selects the **appropriate overloaded function**.*
4. *This selection is done at **compile time**, hence it is called **compile-time polymorphism**.*

Rules for Function Overloading

Functions can be overloaded by:

1. Different number of parameters
2. Different data types of parameters
3. Different order of parameters

Functions **cannot be overloaded** using:

- Only different **return types**

```
#include <iostream>
using namespace std;
// Function 1: adds two integers
int add(int a, int b)
 {
   return a + b;
}
// Function 2: adds three integers
int add(int a, int b, int c)
 {
   return a + b + c;
}
// Function 3: adds two doubles
double add(double x, double y)
{
   return x + y;
}
int main()
{
   cout << "add(2, 3) = " << add(2, 3) << endl;         // calls int add(int, int)
   cout << "add(2, 3, 4) = " << add(2, 3, 4) << endl;    // calls int add(int, int, int)
   cout << "add(2.5, 3.5) = " << add(2.5, 3.5) << endl;  // calls double add(double, double)
   return 0;
}
```

Output:

```
add(2, 3) = 5
add(2, 3, 4) = 9
add(2.5, 3.5) = 6
```

Advantages of Function Overloading:

- **Improved Code Readability:** Using the same function name for similar operations with different data types makes the code more intuitive and easier to understand.

- ***Code Reusability:*** *Reduces code duplication by allowing a single function name to be used for multiple implementations.*
- ***Type Safety:*** *The compiler checks the argument types at compile time, preventing type-related errors during runtime.*

Disadvantages of Function Overloading:

- ***Increased Compilation Time:*** *The compiler needs to analyze all overloaded functions to determine the correct one to call, which can increase compilation time, especially with a large number of overloaded functions.*
- ***Potential for Ambiguity:*** *If the overloaded functions have parameter lists that are too similar, the compiler might not be able to determine which function to call, resulting in a compilation error.*

2. Operator Overloading in C++

Operator overloading *is a feature of C++ that allows* ***existing operators to be redefined*** *so that they can work with* ***user-defined data types (objects)****. It enables operators such as +, –, *, ==, etc., to perform* ***special operations on class objects****.*

In simple words, ***giving additional meaning to an operator for user-defined data types*** *is called operator overloading.*

Need for Operator Overloading

1. *Makes programs* ***more readable and natural***
2. *Allows* ***intuitive operations on objects***
3. *Simplifies* ***complex expressions***
4. *Supports* ***object-oriented programming concepts***
5. *Improves* ***code clarity and usability***

Syntax of Operator Overloading

```
return_type operator operator_symbol (parameter_list) {
   // function body
}
```

Example :

```
Complex operator + (Complex c);
#include <iostream>
using namespace std;
class Counter {
   int count;
public:
   Counter() {
```

```
        count = 0;
    }
    // Overloading ++ operator (prefix)
    void operator ++ () {
        count = count + 1;
    }
    void display() {
        cout << "Count = " << count << endl;
    }
};
int main() {
    Counter c1;
    ++c1;   // operator overloading
    c1.display();
    return 0;
}
```

Output

Count = 1

Explanation

- *The class Counter contains a data member count.*
- *The ++ operator is overloaded using the function operator++().*
- *When ++c1 is executed, the overloaded function is automatically called.*
- *The value of count is incremented by 1.*
- *The updated value is displayed using the display() function.*

Operators That Can Be Overloaded

```
+ - * / % < > <= >= == != = ++ -- [] () ->
```

Operators That Cannot Be Overloaded

```
:: . .* ?: sizeof
```

2.3.2 Run-Time Polymorphism (Dynamic Polymorphism)

II. Run-Time Polymorphism

*Run-time polymorphism, also known as **dynamic polymorphism** or **late binding,** is a type of polymorphism where the specific function to be executed is determined at runtime. This means the compiler does not know which function will be called until the program is running. Run-time polymorphism is primarily achieved through **virtual functions and inheritance**.*

Need for Run-Time Polymorphism

Run-time polymorphism allows:

- *Dynamic method binding*
- *Flexibility in program design*
- *Implementation of real-world hierarchical relationships*
- *Efficient handling of derived class objects through base class pointers*

Run-Time Polymorphism Works

- *A **base class pointer** points to a **derived class object**.*
- *The base class contains a **virtual function**.*
- *The derived class **overrides** the virtual function.*
- *During execution, the function corresponding to the **actual object type** is called.*

Syntax

```
class Base {
public:
   virtual void show() {
      // base class function
   }
};
class Derived : public Base {
public:
   void show() {
      // derived class function
   }
};
```

Example Program

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual void display() {
      cout << "This is Base class display function" << endl;
   }
};
class Derived : public Base {
public:
   void display() {
      cout << "This is Derived class display function" << endl;
   }
```

```
};
int main() {
    Base* b;
    Derived d;
    b = &d;
    b->display();   // dynamic binding
    return 0;
}
```

Output

This is Derived class display function

Explanation

1. *The base class function **display()** is declared as **virtual**.*
2. *The derived class **overrides** the display() function.*
3. *A **base class pointer** points to a **derived class object**.*
4. *During execution, the **derived class version** of the function is called.*
5. *This demonstrates **run-time polymorphism and dynamic binding**.*

Advantages of Run-Time Polymorphism:

- ***Flexibility:** Allows you to write code that can work with objects of different classes in a uniform way, without knowing their specific types at compile time.*
- ***Extensibility:** Makes it easy to add new classes to the system without modifying existing code.*
- ***Code Reusability:** Reduces code duplication by allowing you to write generic code that can be used with objects of different classes.*

Disadvantages of Run-Time Polymorphism:

- ***Performance Overhead:** Dynamic dispatch involves looking up the function to be called at runtime, which can introduce a performance overhead compared to static dispatch.*
- ***Increased Complexity:** Requires careful design and implementation to ensure that the virtual functions are properly defined and overridden.*
- ***Debugging Challenges:** Debugging code that uses run-time polymorphism can be more challenging, as the actual function being called is not known until runtime.*

Virtual Functions

Virtual functions are functions declared in a base class that are expected to be redefined in derived classes. When a virtual function is called through a pointer or reference to a base class object, the actual function executed is determined by the type of the object at runtime, not the type of the pointer or reference. This is known as dynamic dispatch.

Virtual functions are used to:

- *Achieve **dynamic binding***
- *Implement **run-time polymorphism***
- *Enable **base class pointers to call derived class functions***
- *Improve **program flexibility and extensibility***

Example (C++)

```
#include <iostream>
using namespace std;
class Shape
{
public:
  virtual void draw()// virtual function
{
       cout << "Drawing a shape" << endl;
   }
};
class Circle : public Shape
{
public:void draw() override
{
       cout << "Drawing a circle" << endl;
      }
};
int main()
{
   Shape* s;       // base class pointer
   Circle c;
   s = &c;
   s->draw();      // calls Circle::draw() at runtime
   return 0;
}
```

In this example, the `draw` function is declared as virtual in the `Shape` class. The `Circle` class overrides the `draw` function to provide their own specific implementations. When `draw` is called through a pointer to Shape object, the actual function executed depends on the type of the object being pointed to (i.e.,`circle`).

2.3.4 Comparison between Compile-Time and Run-Time Polymorphism

Basis of Comparison	Compile-Time Polymorphism	Run-Time Polymorphism
Definition	*It is a type of polymorphism in which the method call is resolved during the compilation stage.*	*It is a type of polymorphism in which the method call is resolved during program execution.*
Also Known As	*Static polymorphism or early binding.*	*Dynamic polymorphism or late binding.*
Binding Time	*Binding of function call to function definition occurs at compile time.*	*Binding of function call to function definition occurs at run time.*
Implementation	*Implemented using function overloading and operator overloading.*	*Implemented using inheritance and virtual functions.*
Inheritance Requirement	*Inheritance is not mandatory.*	*Inheritance is mandatory.*
Flexibility	*Provides limited flexibility since behavior is fixed at compile time.*	*Provides high flexibility as behavior can change at run time.*
Execution Speed	*Faster due to early binding and no run-time decision making.*	*Slightly slower due to dynamic binding overhead.*
Memory Overhead	*Low memory overhead.*	*Slightly higher memory usage due to virtual function tables.*
Complexity	*Simple to understand and implement.*	*More complex due to inheritance and dynamic binding.*
Examples	*Function overloading, Operator overloading.*	*Virtual functions, Method*

2.3.5 Pure Virtual Function

*A **pure virtual function** is a special type of virtual function in C++ that is **declared in a base class but not defined** there. It is declared by assigning = 0 to the function declaration. The purpose of a pure virtual function is to ensure that the derived class **must provide its own implementation** of that function.*

*When a class contains at least one pure virtual function, that class becomes an **abstract class**. Such a class cannot be used to create objects directly. Pure virtual functions are mainly used to define a **common interface** for derived classes and to achieve **runtime polymorphism**.*

Syntax of Pure Virtual Function

```
virtual return_type function_name() = 0;
```

Example of Pure Virtual Function

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

```
class Bank
{
public:
   virtual void interest() = 0;   // pure virtual function
};

class SBI : public Bank
{
public:
   void interest()
   {
      cout << "SBI interest rate is 6.5%" << endl;
   }
};

class HDFC : public Bank
{
public:
   void interest()
   {
      cout << "HDFC interest rate is 7.0%" << endl;
   }
};

int main()
{
   Bank *b;
   SBI s;
   HDFC h;
   b = &s;
   b->interest();
   b = &h;
   b->interest();
   return 0;
}
```

Output

SBI interest rate is 6.5%
HDFC interest rate is 7.0%

- *The class Bank contains a pure virtual function interest(), making it an abstract class.*
- *The derived classes SBI and HDFC provide their own implementations.*
- *A base class pointer is used to achieve runtime polymorphism.*
- *The correct function is called based on the object type at runtime.*

2.3.6 Abstract Class

*An **abstract class** in C++ is a class that is designed to act as a **base class** and cannot be used to create objects directly. It contains **at least one pure virtual function**, which does not have a definition in the base class. The main purpose of an abstract class is to define a **common structure and behavior** that must be followed by its derived classes.*

*An abstract class provides a **blueprint** for other classes. It specifies what functions a derived class should have, but it does not specify how those functions should work. The actual implementation of the pure virtual functions is done in the derived classes.*

Key Features of Abstract Class

- *Contains one or more **pure virtual functions**.*
- *Objects of an abstract class **cannot be created**.*
- *Used as a **base class** in inheritance.*
- *Derived classes must provide definitions for all pure virtual functions.*
- *Supports **runtime polymorphism** using base class pointers.*

Example of Abstract Class

```
#include <iostream>
using namespace std;
class Shape
{
public:
   virtual void area() = 0;   // pure virtual function
};

class Rectangle : public Shape
{
public:
   void area()
   {
      cout << "Area of Rectangle" << endl;
   }
};
int main()
{
   Shape *s;
   Rectangle r;
   s = &r;
   s->area();
   return 0;
}
```

2.3.7 Difference between Virtual Function and Pure Virtual Function

Feature	Virtual Function	Pure Virtual Function
Definition	A function in base class that can be overridden in derived class.	A function in base class with no implementation (= 0) that **must** be overridden in derived class.
Syntax	virtual void func();	virtual void func() = 0;
Implementation	Can have a body in base class.	Cannot have a body in base class (although C++ allows a body sometimes, conceptually it's abstract).
Class Type	Base class can be instantiated.	Base class becomes **abstract class** → cannot be instantiated.
Purpose	Enables **runtime polymorphism** but base class can still be used.	Forces derived class to provide its own implementation; defines **interface**.
Instantiation	Base class object **can be created**.	Base class object **cannot be created**.
Overriding	Optional in derived class.	Mandatory in derived class; otherwise derived class also becomes abstract.
Example	cpp class Base { virtual void display() { cout << "Base"; } };	cpp class Base { virtual void display() = 0; };
Feature	**Virtual Function**	**Pure Virtual Function**
Definition	A function in base class that can be overridden in derived class.	A function in base class with no implementation (= 0) that **must** be overridden in derived class.
Syntax	virtual void func();	virtual void func() = 0;
Implementation	Can have a body in base class.	Cannot have a body in base class (although C++ allows a body sometimes, conceptually it's abstract).
Class Type	Base class can be instantiated.	Base class becomes **abstract class** → cannot be instantiated.
Purpose	Enables **runtime polymorphism** but base class can still be used.	Forces derived class to provide its own implementation; defines **interface**.
Instantiation	Base class object **can be created**.	Base class object **cannot be created**.
Overriding	Optional in derived class.	Mandatory in derived class; otherwise derived class also becomes abstract.
Example	cpp class Base { virtual void	cpp class Base { virtual void display() = 0;

	display() { cout << "Base"; } };	};

Example:

```
#include <iostream>
using namespace std;
// Abstract base class
class Shape
{
public:
 // Pure virtual function (abstract function)
  virtual double area() = 0;
  virtual void display() = 0;
 // Regular function
  void printType()
 {
    cout << "This is a shape" << endl;
  }
};
// Derived class: Circle
class Circle : public Shape
{
private:
  double radius;
public:
  Circle(double r) : radius(r)
{
}
 // Must implement abstract functions
  double area() override
  {
    return 3.14159 * radius * radius;
  }
  void display() override
  {
    cout << "Circle with radius: " << radius << endl;
    cout << "Area: " << area() << endl;
  }
};
// Derived class: Rectangle
class Rectangle : public Shape {
```

```
private:
  double width, height;
public:
  Rectangle(double w, double h) : width(w), height(h) {}
 // Must implement abstract functions
  double area() override {
    return width * height;
  }
  void display() override {
    cout << "Rectangle with width: " << width << " and height: " << height << endl;
    cout << "Area: " << area() << endl;
  }
};
int main() {
 // Cannot create object of abstract class: Shape s;
  Circle c(5);
  c.display();
  c.printType();
  cout << endl;
  Rectangle r(4, 6);
  r.display();
  r.printType();
  cout << endl;
 // Polymorphism with pointers
  Shape* shape1 = &c;
  Shape* shape2 = &r;
  cout << "Using polymorphism:" << endl;
  shape1->display();
  shape2->display();
  return 0;
}
```

3. CLASS IN C++

3.1 Introduction

*A **class** is a collection of objects that share **common properties and relationships**. It represents a **logical entity** that is used to model real-world objects in a program. In **C++**, a class is a **user-defined data type** that consists of **member variables**, also known as **data members**, and **member functions** that operate on those variables.*

*A class is defined using the keyword class. One of the major advantages of using a class is that it supports **data hiding**, which means the internal details of a class can be protected from direct*

*access by the outside world. This improves **data security** and prevents accidental modification of data.*

*In **Object Oriented Programming (OOP)**, a class is considered the **most important building block**. A class acts as a **blueprint, template, or model** used to create objects. It clearly defines the **properties (data)** and **behaviors (functions)** that the objects of that class will possess.*

In C++, a class groups together:

- ***Data members** – variables that store information about an object*
- ***Member functions** – functions that perform operations on the data members*

*By combining data and functions into a single unit, a class helps in **organizing large programs**, **improving program security**, and **supporting code reusability**. When a class is defined, it creates a new **abstract data type**, which can be used in the same way as built-in data types such as int, float, or char.*

Syntax of Class Declaration

```
class class_name
    {
    private:
       variable declarations;
       function declarations;

    public:
       variable declarations;
       function declarations;
    };
```

3.1 Access Control in Classes

*In C++, **access control** is used to define **how and where the members of a class can be accessed**. This is achieved using **access specifiers**. Access specifiers help in protecting data and implementing the concept of **data hiding**, which is an important feature of object-oriented programming.*

*Access specifiers are always **followed by a colon (:)** and can be used multiple times in the same class to assign different access levels to different members. A class may use **one, two, or all three access specifiers**.*

The three access specifiers available in C++ are:

- ***public***
- ***private***

- ***protected***

Public Access Specifier

*Members declared under the **public** access specifier are **accessible from anywhere** in the program. Both data members and member functions declared as public can be accessed using objects of the class from other classes or functions. Public members form the **interface** of the class through which outside code interacts with the class.*

Example:

```
class PubAccess
{
public:
   int a;         // Data member
   void fun1();    // Member function
};
```

Private Access Specifier

*Members declared under the **private** access specifier **cannot be accessed outside the class**. Only the member functions of the same class can access private members. If an attempt is made to access a private member from outside the class, a **compile-time error** will occur. By default, **all data members and member functions of a class are private** in C++.*

Example:

```
class PriAccess
{
private:
   int a;         // Data member
   void func2();   // Member function
};
```

Protected Access Specifier

*Members declared as **protected** are **not accessible outside the class directly**, but they **can be accessed by derived (subclass) classes**.If class A is inherited by class B, then class B can access the protected members of class A. Protected access is mainly used in **inheritance**.*

Example:

```
class ProAccess
{
protected:
   int a;         // Data member
   void show();    // Member function
};
```

3.1.1 Basis of Comparison Public, Private, Protected

Basis of Comparison	Public	Private	Protected
Conceptual meaning	Members form the external interface of the class	Members represent the internal implementation of the class	Members support controlled access in inheritance
Keyword used	public	private	protected
Visibility scope	Entire program	Limited strictly to the class	Limited to class and its subclasses
Accessibility inside class	Accessible	Accessible	Accessible
Accessibility outside class	Accessible	Not accessible	Not accessible
Accessibility in derived class	Accessible	Not accessible	Accessible
Degree of data protection	Minimal	Maximum	Moderate
Role in data hiding	Does not enforce data hiding	Strongly enforces data hiding	Partially enforces data hiding
Relation to encapsulation	Exposes interface	Hides implementation details	Balances exposure and hiding
Default access specifier in class	No	Yes	No

3.2 Creating Objects

*Once a class has been defined, **any number of objects** belonging to that class can be created. An **object** is a variable whose type is a class. In object-oriented programming, a class represents a group of objects of similar type, while an object represents a specific instance of that class.*

*When an object is declared, the compiler **allocates the required memory space** for the data members defined in the class. Each object has its **own separate copy of data members**, even though all objects share the same member functions. Thus, multiple objects of the same class can store different values and behave independently.*

For example, if a class is defined and three objects named obj1, obj2, and obj3 are created, each object occupies separate memory locations and can store different data values.

Syntax for Creating Objects

ClassName object1, object2, object3;

Example

```
class Sample
{
public:
   int x;
};
Sample obj1, obj2, obj3;
```

Explanation:

- *Sample is a class*
- *obj1, obj2, and obj3 are objects of class Sample*
- *Separate memory is allocated for x in each object*

Example

```
#include <iostream>
using namespace std;
// Class definition
class Student
{
   int rollno;          // Data member (private by default)
   char name[20];       // Data member
   float marks;         // Data member

public:
   // Member function to read data
   void getdata()
   {
      cout << "Enter Roll No: ";
      cin >> rollno;
      cout << "Enter Name: ";
      cin >> name;
      cout << "Enter Marks: ";
      cin >> marks;
   }

   // Member function to display data
   void putdata()
   {
      cout << "\nStudent Details" << endl;
      cout << "Roll No : " << rollno << endl;
```

```
        cout << "Name    : " << name << endl;
        cout << "Marks   : " << marks << endl;
    }
};

int main()
{
    Student s;   // Object creation

    s.getdata(); // Accessing member function
    s.putdata(); // Displaying data

    return 0;
}
```

Output:

```
Enter Roll No: 12
Enter Name: Anu
Enter Marks: 85
Student Details
Roll No : 12
Name    : Anu
Marks   : 85
```

The program explains the concept of a class and object in C++. The class Student is a user-defined data type that groups data members and member functions into a single unit. The data members store student details and are private by default, ensuring data hiding. Public member functions are used to access and display the data in a controlled manner. An object of the class is created to allocate memory and use these functions, thereby demonstrating encapsulation and object-oriented programming principles.

Understanding Purpose:

*"A **class** can also be explained using the **example of a bank account system**. A bank designs a standard format for all bank accounts, which includes details such as account number, account holder name, and balance, along with operations like deposit and withdrawal. This standard format is similar to a **class** in C++, as it defines what data and operations every account will have. Individual bank accounts belonging to different customers are like **objects** created from the class. Although all accounts follow the same format, each account stores different values and functions independently. The bank account format itself does not represent any real account, just*

as a class does not occupy memory until objects are created from it. This analogy helps in understanding how classes define structure and behavior for real-world entities in C++."

3.3 ARRAYS

Arrays are one of the fundamental concepts in C++ and are widely used for storing and processing data efficiently. An array provides a way to store multiple values of the same data type under a single name. All the elements of an array are stored in continuous (contiguous) memory locations, which allows quick and direct access to any element.

Each value stored in an array is called an element. Every element is identified by a unique index (subscript) that represents its position in the array. The index is written inside square brackets after the array name. In C++, array indexing always begins from 0 and ends at size − 1.

An array is a collection of variables of the same data type that share a common name. Individual elements of the array are accessed by specifying the array name followed by an index enclosed in square brackets.

Syntax of Array Declaration

storage_class data_type array_name[size];

- *storage_class specifies the scope and lifetime of the array (auto, static, extern)*
- *data_type indicates the type of elements stored in the array*
- *array_name is the name given to the array*
- *size defines the total number of elements*

If the storage class is not mentioned, the compiler automatically treats it as auto.

Example

int marks[5] = {85, 79, 60, 87, 70};

In this example, marks is an integer array capable of storing five values. The elements are accessed as marks[0], marks[1], marks[2], marks[3], and marks[4].

Characteristics of Arrays

- *Arrays store elements of the same data type, which makes them a homogeneous data structure and simplifies data processing.*
- *All elements of an array are stored in contiguous memory locations, allowing efficient calculation of element addresses.*
- *Each array element is identified by an index (subscript), and in C++ indexing starts from zero and ends at size minus one.*
- *The size of an array is fixed at the time of declaration and remains constant during program execution.*
- *A single array name is used to represent multiple data elements, reducing the number of variable declarations in a program.*

- *Arrays support direct or random access, enabling any element to be accessed immediately using its index.*
- *Memory utilization in arrays is efficient because elements are stored consecutively without gaps.*
- *Arrays can represent data in multiple dimensions, such as one-dimensional, two-dimensional, and multi-dimensional forms.*
- *The order of elements in an array is preserved based on their index positions, which is important for many algorithms.*
- *Arrays form the basic foundation for implementing more complex data structures like stacks, queues, and matrices.*

3.3.1 Types of Arrays Based on Dimensions

*Arrays are classified based on the **number of subscripts (indices)** required to access their elements. Each subscript represents a dimension of the array. The dimensionality of an array determines how data is logically arranged in memory and how it is accessed. The commonly used types of arrays are **one-dimensional arrays**, **two-dimensional arrays**, and **multi-dimensional arrays***

1. One-Dimensional Array (1D Array)

A one-dimensional array is the simplest form of an array in C++. It is used to store a linear collection of elements of the same data type in contiguous memory locations. Each element in a one-dimensional array is identified by a single index (subscript), which represents the position of the element within the array.

The index of a one-dimensional array always starts from 0 and goes up to size − 1, where size is the number of elements in the array. Because the elements are stored consecutively in memory, accessing any element is fast and efficient.

Syntax

```
data_type array_name[size];
```

Example

```
int marks[5] = {85, 79, 60, 87, 70};
```

In this example, marks is a one-dimensional integer array with five elements. The elements are accessed as marks[0], marks[1], marks[2], marks[3], and marks[4].

2. Two-Dimensional Array (2D Array)

*A **two-dimensional array** stores data in the form of **rows and columns**, similar to a table or matrix. It is essentially an **array of one-dimensional arrays**. Each element is accessed using **two indices**, where the first index represents the row number and the second index represents the column number.*

Syntax

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

Example

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

- *matrix is a two-dimensional array with 3 rows and 3 columns*
- *Each row contains a one-dimensional array*
- *Elements are accessed as matrix[0][0], matrix[1][2], etc.*

3. Multi-Dimensional Array

*A **multi-dimensional array** is an array that has **more than two dimensions**. These arrays are used to represent complex data structures where data needs to be organized in multiple levels. Each additional dimension represents another level of indexing.*

Syntax

```
data_type array_name[size1][size2][size3];
```

Example

```
int data[2][3][4];
```

- *This array contains 2 blocks, each having a 3×4 structure*
- *Each element is accessed using three indices, such as data[1][2][3]*
- *Memory is allocated contiguously in row-major order*

Syntax:

```
class ClassName
{
   data_type array_name[size];
};
```

Example:

```
class Student
{
private:
   int marks[5];

public:
   void setMarks()
   {
      for(int i = 0; i < 5; i++)
         cin >> marks[i];
   }
   void showMarks()
   {
      for(int i = 0; i < 5; i++)
```

```
        cout << marks[i] << " ";
    }
};
```

In this example, marks is an array declared inside the class Student. Each Student object contains its own array of marks, and the data is accessed only through member functions.

- *Arrays can be declared as data members of a class*
- *Each object has its own separate array*
- *Memory is allocated when the object is created*
- *Improves organization of related data*
- *Supports encapsulation and data hiding*

3.3.2 Arrays of Objects

*An **array of objects** is an array in which **each element is an object of a class**. In other words, when variables of a class type are stored in an array, it is called an array of objects.*

*Just like an ordinary array stores variables of basic data types (int, float, etc.), an array of objects stores **objects of a class** in consecutive memory locations.*

Need for Arrays of Objects

Arrays of objects are used when:

- *We need to store **multiple objects of the same class***
- *We want to manage **large amounts of similar data***
- *Each object must store **its own data members***

Example: storing details of many students, employees, or products.

Syntax of a Class

```
class class_name
{
private:
    data_type data_members;
public:
    member_functions;
};
```

Syntax for Array of Objects

```
class_name object_name[size];
```

- *class_name → name of the class*
- *object_name → name of the array*
- *size → number of objects*

Example

MyClass obj[10];

*This statement creates an array of **10 objects** of class `MyClass`.*

Program: Initialize and Display Array of Objects

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

class MyClass
{
   int a;
public:
   void set(int x)
   {
      a = x;
   }
   int get()
   {
      return a;
   }
};
int main()
{
   MyClass obj[5];
   for(int i = 0; i < 5; i++)
      obj[i].set(i);
   for(int i = 0; i < 5; i++)
      cout << "obj[" << i << "].get(): " << obj[i].get() << endl;
   return 0;
}
```

Output

```
obj[0].get(): 0
obj[1].get(): 1
obj[2].get(): 2
obj[3].get(): 3
obj[4].get(): 4
```

- *MyClass obj[5]; creates an array of 5 objects of class MyClass.*
- *Each object has its **own copy of data member a**.*
- *The first loop initializes each object using the set() function.*
- *The second loop prints the values using the get() function.*

- *Objects are accessed using **array indexing**, just like normal arrays.*

3.3.3 Arrays With in a Class

A class can contain arrays as member variables. This allows the class to manage collections of data internally.

Syntax

```
class ClassName {
   data_type array_name[size];
};
```

Example Program

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

class Student {
   int marks[5];   // array within a class
public:
   void getMarks() {
      cout << "Enter 5 marks: ";
      for(int i = 0; i < 5; i++) {
         cin >> marks[i];
      }
   }
   void displayMarks() {
      cout << "Marks are: ";
      for(int i = 0; i < 5; i++) {
         cout << marks[i] << " ";
      }
   }
};
int main() {
   Student s1;
   s1.getMarks();
   s1.displayMarks();
   return 0;
}
```

- *The array `marks[5]` is declared inside the class **Student**.*
- *Each object of the class gets **its own copy of the array**.*
- *The function **getMarks()** is used to input values into the array.*
- *The function **displayMarks()** is used to display the array elements.*

- *This method helps in **grouping related data inside a single class**.*

3.4 Static Class Members

*In C++, when we create objects of a class, each object usually has its **own separate data members**. However, in some cases, all objects need to **share the same variable or function**. To handle such situations, C++ provides **static class members**. A static class member belongs to the **class itself**, not to individual objects, and therefore **all objects of the class use the same shared member**.*

Static class members are of two types:

1. ***Static Data Members***
2. ***Static Member Functions***

1. Static Data Members

*A **static data member** is a variable declared inside a class using the keyword `static`. Unlike normal data members, **only one copy** of a static data member exists for the entire class, irrespective of how many objects are created.*

Purpose of Static Data Members

1. *Static data members store data that is common to all objects of a class, so every object can access and share the same value.*
2. *Only one copy of a static data member is created in memory, regardless of how many objects of the class are instantiated.*
3. *Static data members are used to count the number of objects in a program because the value is shared and updated by all objects.*
4. *Static data members save memory by allocating storage only once instead of creating separate copies for each object.*
5. *Any change made to a static data member by one object is immediately reflected in all other objects of the same class.*
6. *Static data members exist for the entire lifetime of the program and retain their values until the program terminates.*

Characteristics of Static Data Members

1. *Static data members belong to the **class as a whole** and not to individual objects.*
2. *Only **one copy** of a static data member is created and shared by all objects of the class.*
3. *Static data members are **initialized only once**, even if multiple objects are created.*
4. *They are **defined outside the class** using the scope resolution operator.*
5. *Static data members have a **lifetime equal to the entire program execution**.*
6. *By default, static data members are **initialized to zero** if no explicit initialization is provided.*
7. *Static data members can be accessed using the **class name or object name**.*
8. *Any modification made by one object **affects all other objects** of the same class.*

9. *Static data members help in maintaining* ***class-level shared information***.

Syntax

```
data_type class_name::static_variable_name;
```

Example Program

```
#include <iostream>
using namespace std;
class Item
{
   static int count;   // static data member
   int number;

public:
   void getData(int a)
   {
      number = a;
      count++;
   }
   void showCount()
   {
      cout << "Count is " << count << endl;
   }
};
int Item::count;   // definition of static member
int main()
{
   Item a, b, c;
   a.showCount();
   b.showCount();
   c.showCount();
   a.getData(10);
   b.getData(20);
   c.getData(30);
   cout << "After reading data" << endl;
   a.showCount();
   b.showCount();
   c.showCount();

   return 0;
}
```

Output

Count is 0
Count is 0
Count is 0
After reading data
Count is 3
Count is 3
Count is 3

- *count is static, so only **one copy** exists.*
- *Initially, count = 0.*
- *Each call to getData() increases count.*
- *After three objects update it, count = 3.*
- *All objects display the same value because the variable is shared.*

2. Static Member Functions

*A **static member function** is a function declared using the keyword `static` inside a class. It belongs to the **class itself** rather than to any specific object and can be called using the **class name**. A static member function can access **only static data members** of the class.*

Purpose of Static Member Functions

1. *Static member functions are used to **access and operate on static data members** of a class.*
2. *They allow performing **class-level operations** without creating objects.*
3. *Static member functions help in **managing shared data** that belongs to the class.*
4. *They provide a way to define **utility functions** related to a class.*
5. *Static member functions help in **maintaining data consistency** across all objects.*
6. *They allow functions to be called using the **class name**, making the code clear and organized.*
7. *Static member functions are useful when a function does not depend on **object-specific data**.*

Characteristics of Static Member Functions

1. *Static member functions belong to the **class**, not to individual objects.*
2. *They can be called using the **class name** without creating any object.*
3. *Static member functions can access **only static data members** of the class.*
4. *They **cannot directly access non-static data members** of the class.*
5. *Static member functions are **created only once** and shared by all objects.*
6. *They are useful for performing **class-level operations**.*
7. *Static member functions exist for the **entire duration of the program**.*

8. *They help in keeping **shared operations** separate from object-specific operations*

Syntax (Calling)

```
class_name::static_function_name();
```

Example Program

```
#include <iostream>
using namespace std;
class Test
{
   int code;
   static int count;
public:
   void setCode()
   {
      code = ++count;
   }
   void showCode()
   {
      cout << "Object number " << code << endl;
   }
   static void showCount()
   {
      cout << "Count " << count << endl;
   }
};
int Test::count;
int main()
{
   Test t1, t2;
   t1.setCode();
   t2.setCode();
   Test::showCount();
   Test t3;
   t3.setCode();
   Test::showCount();
   t1.showCode();
   t2.showCode();
   t3.showCode();
   return 0;
}
```

Output

Count 2
Count 3
Object number 1
Object number 2
Object number 3

- *count is shared among all objects.*
- *Each object gets a unique code value.*
- *showCount() is called using the class name, proving it is independent of objects.*
- *Static member functions work only with static data.*

Advantages of Static Member Functions

1. *Static member functions can be called using the **class name**, so object creation is not required.*
2. *They are useful for performing operations that are related to the **class as a whole**.*
3. *They can efficiently access and modify **static data members**.*
4. *Only one copy of a static member function exists, which helps in saving memory.*
5. *They improve program organization by clearly separating class-level operations from object-level operations.*

Limitations of Static Member Functions

1. *Static member functions cannot directly access **non-static data members** of the class.*
2. *They do not have a `this` **pointer**, since they are not associated with any object.*
3. *They are not suitable for operations that depend on **individual object data**.*
4. *Their usage is limited when object-specific behavior is required.*

UNIT-III

1.Constructors and Destructors

1.1Constructor

*A **constructor** is a special member function of a class whose main purpose is to **initialize the objects of that class**. It allocates memory for the data members of an object and assigns them **appropriate initial values**. A constructor is **automatically invoked** whenever an object of the class is created.*

Special Features of a Constructor

- *The name of the constructor is **the same as the class name**.*
- *A constructor has **no return type**, not even `void`.*
- *It is **automatically called** at the time of object creation.*
- *Constructors can be **overloaded**, meaning a class can have multiple constructors with different parameter lists.*

Characteristics of Constructors

1. *Automatically executed when an object is created.*
2. *Cannot be called explicitly like normal functions (except through placement new).*
3. *They don't have a return type.*
4. *Constructors can be defined inside or outside the class.*
5. *Constructors can be overloaded.*
6. *They can have **default arguments**.*
7. *If you **do not define any constructor**, C++ provides a **default constructor**.*
8. *If you define **any constructor**, the compiler will **not** provide a default one.*

1.2 Types of Constructors

1. ***Default Constructor***
2. ***Parameterized Constructor***
3. ***Copy Constructor***
4. ***Dynamic Constructor***

1. Default constructor

*A **default constructor** is a constructor that **does not take any arguments**. It is used to initialize the data members of an object with **default values** at the time of object creation. The default constructor is **automatically invoked** when an object is created without passing any values.*

Syntax of Default Constructor

class ClassName

```
{
public:
   ClassName()
   {
      // initialization code
   }
};
```

Characteristics of Default Constructor

1. *The name of the default constructor is the **same as the class name**.*
2. *It **takes no parameters**.*
3. *It is **automatically called** when an object is created.*
4. *If no constructor is defined, the **compiler provides a default constructor**.*
5. *If any constructor is explicitly defined, the **compiler does not generate** a default constructor.*

Example Program

```
#include <iostream>
using namespace std;
class Number
{
   int x;
public:
   // Default Constructor
   Number()
   {
      x = 0;
   }
   void display()
   {
      cout << "Value of x = " << x << endl;
   }
};
int main()
{
   Number n1;   // Default constructor is called
   n1.display();
   return 0;
}
```

Output

Value of x = 0

2. Parameterized Constructor

*A **parameterized constructor** is a constructor that **accepts one or more arguments**. It is used to initialize the data members of an object with **values provided at the time of object creation**. This allows different objects of the same class to be initialized with **different data**.*

Syntax of Parameterized Constructor

```
class ClassName
{
public:
   ClassName(data_type parameter1, data_type parameter2)
   {
      // initialization using parameters
   }
};
```

Characteristics of Parameterized Constructor

1. *A parameterized constructor has the **same name as the class**.*
2. *It **takes parameters** to initialize data members.*
3. *It is **automatically invoked** when arguments are passed during object creation.*
4. *It allows **different objects to have different initial values**.*
5. *When a parameterized constructor is defined, the object must be created by **passing values**.*

Example1

```
#include <iostream>
using namespace std;
class Number
{
   int x, y;
public:
   // Parameterized Constructor
   Number(int a, int b)
   {
      x = a;
      y = b;
   }
   void display()
   {
      cout << "x = " << x << ", y = " << y << endl;
```

```
    }
};
int main()
{
    Number n1(10, 20);   // Parameterized constructor is called
    n1.display();
    return 0;
}
```

Output

```
x = 10, y = 20
```

- The constructor `Number(int a, int b)` is a parameterized constructor.
- Values `10` and `20` are passed during object creation.
- The constructor initializes the data members `x` and `y` with these values.
- A parameterized constructor initializes objects with values provided at the time of their creation.

Example 2:

```
class Rectangle
{
private:
  int length;
  int width;
public:
  // Parameterized Constructor
  Rectangle(int l, int w) : length(l), width(w)
 {
 }
  int area()
{ return length * width;
}
};
int main()
{
  Rectangle rect(10, 5); // Creating a Rectangle object with length 10 and width 5
  return 0;
}
```

In this example, the `Rectangle` class has a parameterized constructor that takes two integer arguments, `l` and `w`, representing the length and width of the rectangle, respectively. When a `Rectangle` object is created using `Rectangle rect(10, 5)`, the constructor is called with the values 10 and 5, which are then used to initialize the `length` and `width` data members of the

object. The initialization list `: length(l), width(w)` *is an efficient way to initialize member variables.*

3. Copy Constructor

*A **copy constructor** is a special constructor used to **create a new object by copying an existing object** of the same class. It takes a **reference to an object of the same class** as its argument. The copy constructor is automatically invoked whenever an object is initialized using another object.*

*If a copy constructor is not explicitly defined, the **compiler provides a default copy constructor**, which performs a **member-wise (shallow) copy** of the data members.*

Characteristics of Copy Constructor

1. *A copy constructor **initializes one object from another object** of the same class.*
2. *It takes a **reference to an object of the same class** as its parameter.*
3. *If no copy constructor is defined, the **compiler generates a default copy constructor**.*
4. *The default copy constructor performs a **shallow copy** of data members.*
5. *A **user-defined copy constructor is required** when the class uses **dynamic memory allocation**.*
6. *It helps in **safe copying of objects**, especially when pointers are involved.*
7. *It avoids **redundant re-initialization** by directly copying existing values.*

A copy constructor is called:

- *When an object is **initialized using another object** of the same class*
- *When an object is **passed by value** to a function*
- *When an object is **returned by value** from a function*

Syntax of Copy Constructor

```
class ClassName
{
public:
   ClassName(const ClassName &object)
   {
      // copy initialization
   }
};
```

Example Program

```
#include <iostream>
using namespace std;
class Sample
{
```

```
    int x;
public:
    Sample(int a)
    {
        x = a;
    }
    // Copy Constructor
    Sample(const Sample &s)
    {
        x = s.x;
    }
    void display()
    {
        cout << "Value of x = " << x << endl;
    }
};
int main()
{
    Sample A(100);
    Sample B(A);   // Copy constructor called
    A.display();
    B.display();

    return 0;
}
```

Output

Value of x = 100
Value of x = 100

4.Dynamic (Deep Copy) Constructor

*A **dynamic constructor** is a constructor that performs **dynamic memory allocation** using the `new` operator. When such a constructor is used to copy an object that contains dynamically allocated memory, it is called a **deep copy constructor**.*
*A deep copy constructor creates a **separate copy of dynamically allocated data**, ensuring that each object has its **own independent memory**.*

Need for Dynamic (Deep Copy) Constructor

*When a class contains **pointer data members**, the default copy constructor performs a **shallow copy**, which copies only the memory addresses. This may lead to problems such as **memory***

corruption and double deletion.
To avoid these issues, a ***dynamic (deep copy) constructor*** *must be defined.*

Characteristics of Dynamic (Deep Copy) Constructor

1. *It allocates memory* ***dynamically*** *using the* `new` *operator.*
2. *It creates a* ***separate copy of data****, not just the address.*
3. *It is required when a class uses* ***dynamic memory allocation****.*
4. *It prevents problems like* ***dangling pointers and double deletion****.*
5. *Each object maintains its* ***own copy of dynamically allocated memory****.*

Syntax of Dynamic (Deep Copy) Constructor

```
class ClassName
{
   data_type *ptr;
public:
   ClassName(const ClassName &obj)
   {
      ptr = new data_type;
      *ptr = *(obj.ptr);
   }
};
```

Program: Dynamic Constructor Example

```
#include <iostream>
#include <cstring>
using namespace std;
class String
{
   char *str;
public:
   // Dynamic Constructor
   String(const char *s)
   {
      str = new char[strlen(s) + 1];   // dynamic memory allocation
      strcpy(str, s);
   }
   void display()
   {
      cout << "String value: " << str << endl;
   }
```

```
    // Destructor
    ~String()
    {
        delete[] str;   // free allocated memory
    }
};
int main()
{
    String s1("Dynamic Constructor Example");
    s1.display();
    return 0;
}
```

Output

String value: Dynamic Constructor Example

*This program demonstrates the use of a **dynamic constructor** in C++. The class String contains a pointer data member str that stores a string using dynamic memory allocation. When an object is created, the constructor allocates memory at run time using the new operator and copies the given string into that memory. The display() function prints the stored string. A destructor is defined to release the dynamically allocated memory using delete[], which prevents memory leaks. Thus, the program shows how dynamic constructors allow run-time initialization of objects.*

1.3 Destructors

*A **destructor** is a special member function of a class that is used to **destroy objects** created by a constructor. It is automatically called when an object goes out of scope or when the program ends. The main purpose of a destructor is to **release memory and other resources** occupied by the object.*

*A destructor has the **same name as the class**, but it is preceded by a **tilde (~)** symbol.*

Example Syntax

```
~ClassName()
{
    // cleanup code
}
```

Characteristics of Destructors

1. *A destructor **does not take any arguments** and **does not return any value**.*
2. *It is **invoked automatically by the compiler** when an object is destroyed.*
3. *Destructors are used to **free memory and clean up resources**.*

4. *It is a **good programming practice** to define a destructor, especially when dynamic memory is used.*

Example:

```
class MyClass
{
public:
  MyClass()
{
 std::cout << "Constructor called" << std::endl;
}
  ~MyClass()
{
std::cout << "Destructor called" << std::endl;
}
};
int main()
{
  MyClass obj; // Constructor called when obj is created
  return 0;    // Destructor called when obj goes out of scope
}
```

In this example, the `MyClass` class has a constructor and a destructor. The constructor is called when an object of `MyClass` is created, and the destructor is called when the object goes out of scope (e.g., when the function in which it was declared returns).

1.4 Difference between Constructor and Destructor

Constructor	***Destructor***
*Arguments **can be passed** to a constructor.*	*Arguments **cannot be passed** to a destructor.*
*More than one constructor can be declared in a class using **constructor overloading**.*	*Only **one destructor** can be declared since it does not take arguments.*
*Constructors are **not virtual**.*	*Destructors **can be virtual**.*
*The name of the constructor is **the same as the class name**.*	*The name of the destructor is the same as the class name but **prefixed with the tilde (~) symbol**.*
*Constructors are used to **initialize objects**.*	*Destructors are used to **destroy objects and free memory**.*
*Constructors are called **when objects are created**.*	*Destructors are called **when objects are destroyed**.*

Program: Constructor and Destructor Demonstration (Student Example)

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

class Student
{
   char *name;
   int roll;

public:
   // Default Constructor
   Student()
   {
      name = nullptr;
      roll = 0;
      cout << "Default Constructor called" << endl;
   }

   // Parameterized Constructor
   Student(const char *n, int r)
   {
      name = new char[strlen(n) + 1];
      strcpy(name, n);
      roll = r;
      cout << "Parameterized Constructor called" << endl;
   }

   // Copy Constructor
   Student(const Student &s)
   {
      roll = s.roll;
      if (s.name != nullptr)
      {
         name = new char[strlen(s.name) + 1];
         strcpy(name, s.name);
      }
      else
      {
         name = nullptr;
      }
      cout << "Copy Constructor called" << endl;
   }
```

```
    // Destructor
    ~Student()
    {
        delete[] name;
        cout << "Destructor called" << endl;
    }

    // Display function
    void display()
    {
        cout << "Roll No: " << roll << endl;
        if (name != nullptr)
            cout << "Name: " << name << endl;
        else
            cout << "Name not assigned" << endl;
    }
};

int main()
{
    Student s1;                 // Default constructor
    s1.display();

    Student s2("Rahul", 101);   // Parameterized constructor
    s2.display();

    Student s3 = s2;            // Copy constructor
    s3.display();

    return 0;
}
```

Sample Output

```
Default Constructor called
Roll No: 0
Name not assigned
Parameterized Constructor called
Roll No: 101
Name: Rahul
```

Copy Constructor called
Roll No: 101
Name: Rahul
Destructor called
Destructor called
Destructor called

2. Operator Overloading and Type Conversion

2.1Operator Overloading

Operator overloading *is an important feature of C++ that allows a single operator to perform* ***different operations depending on the context****. It is one of the ways to implement* ***polymorphism*** *in C++. By operator overloading, existing operators can be given* ***new meanings*** *when applied to user-defined data types such as objects.*

Even after overloading, the ***precedence, associativity, and syntax of operators remain unchanged****.*

Operator overloading is a mechanism that allows C++ operators to be ***redefined*** *so that they can operate on objects of user-defined classes.*

Syntax of Operator Function

```
return_type class_name :: operator op (argument_list)
{
   function body
}
```

Rules for Operator Overloading

Operator overloading is subject to the following rules and limitations:

1. *Only* ***existing operators*** *can be overloaded; new operators cannot be created.*
2. *At least* ***one operand must be of a user-defined data type****.*
3. *The* ***basic meaning of an operator cannot be changed****.*
4. *The* ***syntax, precedence, and associativity*** *of operators cannot be modified.*
5. *Some operators* ***cannot be overloaded****.*
6. *Certain operators can be overloaded* ***only using member functions****, not friend functions.*

Operators That Cannot Be Overloaded

The following operators ***cannot be overloaded in C++****:*

- *: : (Scope resolution operator)*

- . *(Member access operator)*
- .* *(Pointer to member operator)*

All other operators can be overloaded.

Types of Operator Overloading

1.Unary Operator Overloading

*A **unary operator** operates on a single operand. For example, the unary minus (-) operator changes the sign of a basic data type value. When this operator is overloaded for a class, it should perform a similar operation on the object, such as changing the sign of all its data members.*

Unary operators can be overloaded so that objects behave like basic data types when such operators are applied.

2.Binary Operator Overloading

*A **binary operator** operates on two operands. For example, the + operator is commonly used to add two values. In earlier programs, functions such as:*
C = sum(A, B);
were used to add two objects. By overloading the + operator, the same operation can be written in a more natural form:
C = A + B;

*This improves **readability and clarity** of the program.*

Overloading Binary Operators Using Friend Functions

*Binary operators can be overloaded using either **member functions** or **friend functions**.*

- *When a **member function** is used, the left-hand operand is passed **implicitly**, and the right-hand operand is passed as an argument.*
- *When a **friend function** is used, **both operands are passed explicitly** as arguments.*

Thus:

- *Member function → one explicit argument*
- *Friend function → two explicit arguments*

Friend functions are useful when the left-hand operand is not an object of the class

Example:

```
#include <iostream>
using namespace std;
class Number
{
   int value;
```

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

    Number operator + (Number n)
            {
       return Number(value + n.value);
     }

    void display()
            {
            cout << value << endl;
            }
};
int main()
{
    Number n1(5), n2(10);
    Number n3 = n1 + n2;
    n3.display();  // Output: 15
    return 0;
}
```

In this example, the `+` operator is overloaded for the `Complex` class. When `n1 + n2` is evaluated, the overloaded `operator+` function is called, which adds the real and imaginary parts of the two `Complex` objects and returns a new `Complex` object representing the sum.

Advantages of Operator Overloading:

- ***Improved Code Readability:*** *Allows you to use familiar operators with user-defined types, making the code more natural and easier to understand.*
- ***Enhanced Expressiveness:*** *Enables you to define custom behavior for operators, allowing you to express complex operations in a concise and intuitive way.*
- ***Code Reusability:*** *Reduces code duplication by providing a single operator symbol for multiple implementations.*

Disadvantages of Operator Overloading:

- ***Potential for Misuse:*** *Overloading operators in a way that is inconsistent with their conventional meaning can lead to confusion and unexpected behavior.*

- ***Increased Complexity:*** *Overloading operators can make the code more complex, especially if the overloaded operators have complex implementations.*
- ***Limited Applicability:*** *Not all operators can be overloaded, and some operators have restrictions on how they can be overloaded.*

Overloading Operators

*You can overload most of the C++ operators, including arithmetic operators (+, -, *, /), relational operators (==, !=, <, >, <=, >=), logical operators (&&, ||, !), bitwise operators (&, |, ^, ~, <<, >>), assignment operators (=, +=, -=, *=, /=), and increment/decrement operators (++, --).*

2.2 Type Conversions

Type Conversion *in C++ is the process of converting a value of one* ***data type*** *to another* ***data type****.*

- *It can be* ***done automatically by the compiler*** *or* ***manually by the programmer****.*
- *Helps in* ***avoiding type mismatch errors*** *during arithmetic operations or assignments.*

Types of Type Conversion

1. ***Implicit Conversion***
2. ***Explicit Conversion***

1. Implicit Conversion (Type Promotion) in C++

Implicit conversion *is a type of conversion that is* ***automatically performed by the compiler*** *without any explicit instruction from the programmer. It commonly occurs when an expression contains operands of different data types or when a value of one data type is assigned to a variable of another data type. In such cases, the compiler converts the data type of the* ***right-hand operand*** *to match the data type of the* ***left-hand operand****.*

Implicit conversion generally takes place from a ***smaller data type to a larger data type*** *in order to* ***avoid loss of information****. This process is also known as* ***type promotion****. For example, when an integer value is assigned to a floating-point variable, the integer is automatically converted into a float. Implicit conversion helps keep programs simple by reducing the need for explicit type casting.*

Characteristics of Implicit Conversion

1. *No* ***explicit type cast*** *is required from the programmer.*
2. *Conversion is performed* ***automatically by the compiler****.*
3. *It is done to* ***ensure compatibility*** *between operands.*
4. *Usually converts a* ***smaller data type to a larger data type****.*

5. *Helps reduce the chances of **data loss** in expressions.*
6. *Improves **simplicity and readability** of programs.*

Example 1: Implicit Conversion in Assignment

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

int main()
 {
   int a = 10;
   double b = a;   // int → double (automatic)
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
   return 0;
}
```

Output:

```
a = 10
b = 10
```

2. Explicit Conversion

*In C++, sometimes the automatic type conversion performed by the compiler is **not sufficient or not desired**. In such cases, the programmer needs to **forcefully convert one data type into another**. This manual conversion of data types is known as **explicit conversion** or **type casting**.*

*Explicit conversion gives the programmer **complete control** over how and when a value is converted from one data type to another.*

***Explicit conversion** is a type of conversion in which the **programmer explicitly specifies the target data type** to which a value should be converted. Unlike implicit conversion, it does not happen automatically and must be written clearly in the program.*

Syntax of Explicit Conversion

Explicit conversion can be performed using the following syntax:

```
(type) expression;
```

Characteristics of Explicit Conversion

1. *Explicit conversion is **performed manually by the programmer**.*
2. *It requires an **explicit type cast** to specify the target data type.*
3. *The compiler **does not perform this conversion automatically**.*
4. *It may result in **loss of data or precision**, especially when converting from larger to smaller data types.*

5. *It provides **greater control** over type conversion.*
6. *It is commonly used when **implicit conversion is not suitable**.*
7. *Explicit conversion improves **accuracy in critical calculations** when used carefully..*

Forms of Explicit Type Casting

```
#include <iostream>
using namespace std;
int main()
{
   double pi = 9.87;
   // C-style cast
   int a = (int)pi;
   // Function-style cast
   int b = int(pi);
   // C++ style cast
   int c = static_cast<int>(pi);
   cout << "Original double: " << pi << endl;
   cout << "C-style cast: " << a << endl;
   cout << "Function-style cast: " << b << endl;
   cout << "C++ static_cast: " << c << endl;
   return 0;
}
```

Output:

```
Original double: 9.87
C-style cast: 9
Function-style cast: 9
C++ static_cast: 9
```

3. Inheritance

*Inheritance is an important property of **Object-Oriented Programming (OOP)** through which one class can **acquire the properties of another class**. It is a mechanism that allows the creation of a **new class from an existing class**. The existing class is known as the **base class** (or parent/super class), and the new class is known as the **derived class** (or child/sub class).*

*The derived class automatically inherits the **public and protected data members and member functions** of the base class. The **private members of a base class are not inherited**, but they can still be accessed indirectly through public or protected member functions of the base class.*

Definition

*Inheritance is the process by which a derived class acquires the **data members and member functions** of a base class, enabling **code reuse and extension of functionality**.*

1. ***Base Class (Parent Class)***
   - *The class whose members are inherited.*
2. ***Derived Class (Child Class)***
   - *The class that inherits members of the base class.*
3. ***Access Specifiers** in inheritance:*
   - `public` → *public members remain public, protected remain protected*
   - `protected` → *public & protected members become protected*
   - `private` → *public & protected members become private*
4. ***Private members** of the base class are **never inherited**, but can be accessed via public/protected functions of base class.*

Syntax

```
class Base
 {
   // base class members
};
class Derived : access_specifier Base
 {
   // derived class members
};
```

Need for Inheritance

1. *Inheritance is needed to **reuse existing code** without rewriting it.*
2. *It helps in **reducing code duplication** in programs.*
3. *Inheritance allows **incremental development** by extending existing classes.*
4. *It makes programs **easier to maintain and modify**.*
5. *Inheritance helps in **organizing classes hierarchically**.*
6. *It supports **polymorphism**, which improves program flexibility.*
7. *Inheritance helps in **modeling real-world relationships** between objects.*

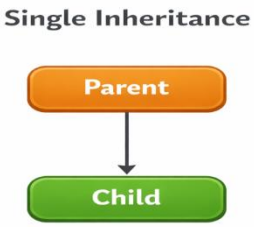
3.1 Types of Inheritance

1. ***Single Inheritance** – One base, one derived class*
2. ***Multiple Inheritance** – More than one base class*
3. ***Multilevel Inheritance** – Chain of inheritance (A → B → C)*
4. ***Hierarchical Inheritance** – Multiple derived classes from one base class*
5. ***Hybrid/Virtual Inheritance** – Combination of above types*

1. Single Inheritance

Single inheritance is the simplest form of inheritance, where a class inherits properties and behaviors from only one parent class. This creates a straightforward "is-a" relationship between the derived class (child class) and the base class (parent class).

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



Example:

```
#include <iostream>
using namespace std;
// Base class
class Person
{
public:
   string name;
   void showName()
   {
      cout << "Name: " << name << endl;
   }
};
// Derived class
class Student : public Person
{
public:
   int rollNo;
   void showRoll()
{
      cout << "Roll No: " << rollNo << endl;
    }
};

int main()
{
   Student s;
   s.name = "Alice";   // inherited from Person
```

Understanding Purpose not for Exam

Example: Parent and Child in a Family 🎎

*Think of a **family** where a **child has only one parent**.*

- ***Parent*** → *Has common qualities like* surname, eye color, family traditions
- ***Child*** → *Inherits these qualities from **only that one parent** and may also have its* own unique traits *(like hobbies or skills)*

*The child **cannot inherit from any other parent** in this case.*

```
    s.rollNo = 101;

    s.showName();      // inherited function
    s.showRoll();      // derived class function
}
```

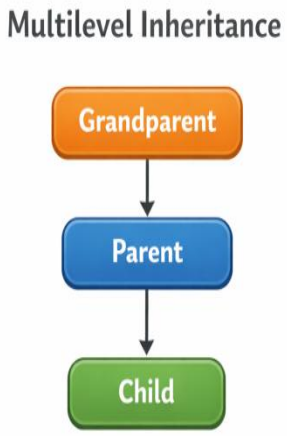
Output:
Name: Alice
Roll No: 101

2. Multilevel Inheritance

Multilevel inheritance involves a chain of inheritance, where a class inherits from another class, which in turn inherits from another class. This creates a hierarchy of classes with increasing specialization.

Syntax

```
class Base {
    // Base class members
};
class Derived1 : public Base {
    // Derived class 1 members
};
class Derived2 : public Derived1 {
    // Derived class 2 members
};
```



Example:

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

class Vehicle
{
public:
    void type()
        {
        cout << "This is a vehicle\n";
        }
};

class Car : public Vehicle
{
public:
```

Understanding Purpose not for Exam

Example-Multilevel Inheritance
Three Generations in a Family
👴 → 👵 → 👶
*Think about a **family tree with three generations**:*

- ***Grandparent** → has family name, traditions, and values*
- ***Parent** → inherits these from the grandparent and adds new qualities*
- ***Child** → inherits qualities from the parent (and indirectly from the grandparent)*

*So, the **child gets features step-by-step**, not directly from the grandparent.*

```
    void carType()
        {
        cout << "This is a car\n";
        }
};
class SportsCar : public Car
{
public: void speed()
        {
         cout << "High-speed car\n";
        }
};
int main()
{
   SportsCar sc;
   sc.type();    // Vehicle
   sc.carType(); // Car
   sc.speed();   // SportsCar
}
```

Output:

This is a vehicle
This is a car
High-speed car

3. Multiple Inheritance

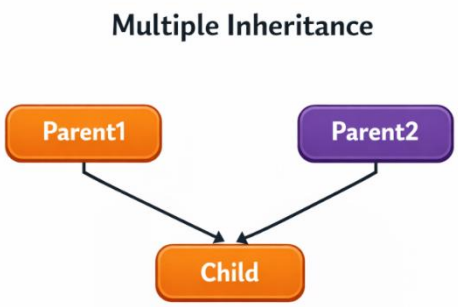
Multiple inheritance allows a class to inherit from multiple parent classes. This enables a class to combine features and behaviors from different sources. However, it can also lead to complexities, such as the "diamond problem," which arises when a class inherits from two classes that both inherit from a common ancestor.

Syntax

```
class Base1 {
  // Base class 1 members
};

class Base2 {
  // Base class 2 members
};

class Derived : public Base1, public
```



```
Base2 {
  // Derived class members
};
```

Example

```
#include <iostream>
using namespace std;
class Printer
{
public:
   void print()
         {
         cout << "Printing\n";
         }
};
class Scanner
 {
public:
   void scan()
         {
         cout << "Scanning\n";
         }
};
class AllInOne : public Printer, public Scanner { };

int main()
{
   AllInOne device;
   device.print();
   device.scan();
}
```

Output:

Printing
Scanning

Understanding Purpose not for Exam

Multiple Inheritance -Smartphone

*Think of a **smartphone** that combines features from different devices:*

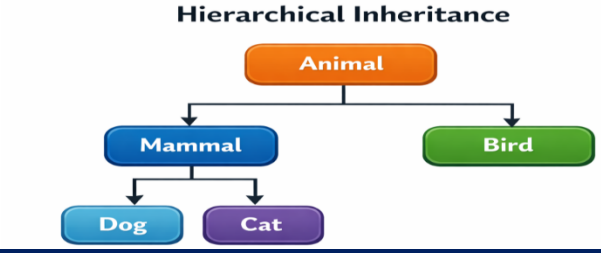
- ***Camera** → takes photos and videos*
- ***Music Player** → plays songs and audio*
- ***Smartphone** → has **camera features** + **music player features***

*So, one device gets abilities from **more than one source**.*

4. Hierarchical Inheritance

Hierarchical inheritance involves multiple classes inheriting from a single parent class. This creates a tree-like structure where the parent class serves as a base for multiple specialized classes.

Syntax



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

class Derived1 : public Base {
  // Derived class 1 members
};
class Derived2 : public Base {
  // Derived class 2 members
};
```

Example

```
#include <iostream>
using namespace std;
class Shape
 {
public:
   void display()
        {
        cout << "This is a shape\n";
        }
};
class Circle : public Shape
 {
public:
   void drawCircle()
        {
        cout << "Drawing Circle\n";
        }
};
class Rectangle : public Shape
 {
public:
   void drawRectangle()
        {
        cout << "Drawing
Rectangle\n";
        }
};
int main()
```

Understanding Purpose not for Exam

Hierarchical Inheritance-Vehicle System

- ***Vehicle*** → *common features (speed, fuel type)*
- ***Car*** → *inherits from Vehicle (air-conditioning)*
- ***Bike*** → *inherits from Vehicle (two-wheeler features)*
- ***Bus*** → *inherits from Vehicle (large seating capacity)*

✔ *One **Vehicle (parent class)** gives basic features to **many different child classes**.*
*This is a perfect example of **hierarchical inheritance***

OOP Mapping

- *Base class: **Vehicle***
- *Derived classes: **Car, Bike, Bu***

One-Line Exam Answer

Hierarchical inheritance occurs when multiple derived classes inherit from a single base class.

```
{
    Circle c;
    Rectangle r;
    c.display();
    c.drawCircle();
    r.display();
    r.drawRectangle();
}
```

Output:

```
This is a shape
Drawing Circle
This is a shape
Drawing Rectangle
```

5. Hybrid Inheritance

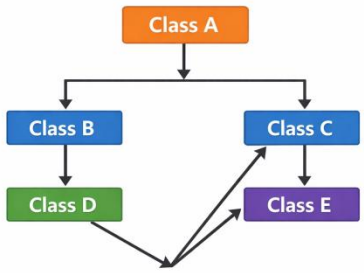
Hybrid inheritance is a combination of two or more types of inheritance. It can involve a mix of single, multilevel, multiple, and hierarchical inheritance to create complex class hierarchies. Due to the potential for complexity and ambiguity, hybrid inheritance should be used with caution.

```
class Base1 {
    // Base class 1 members
};

class Base2 {
    // Base class 2 members
};

class Derived1 : public Base1 {
    // Derived class 1 members
};

class Derived2 : public Base1, public Base2 {
    // Derived class 2 members
};
```



Hybrid Inheritance

Example:

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

```
class Person
{
public:
   void identity()
        {
        cout << "I am a person\n";
        }
};
class Father : virtual public Person
        {
        };
class Mother : virtual public Person
        {
        };
class Child : public Father, public Mother
        {
        };
int main()
{
   Child c;
   c.identity(); // only one copy of Person
}
```

Output:

I am a person

Understanding Purpose not for Exam

Hybrid Inheritance –

Smart Classroom System

*Think of a **smart classroom** that combines different systems:*

- ***Room*** *→ basic features (room number, lights)*
- ***DigitalRoom*** *→ inherits from Room (projector, smart board)*
- ***LabRoom*** *→ inherits from Room (computers, lab equipment)*
- ***AdvancedSmartLab*** *→ inherits from **DigitalRoom** + **LabRoom***

*Since it combines **multilevel** + **hierarchical** + **multiple inheritance**, it is called **hybrid inheritance***

UNIT - IV

1.Pointers in C++

1.1 Pointers

A pointer in C++ is a special type of variable whose purpose is to store the memory address of another variable. Unlike ordinary variables that hold actual data values, a pointer holds the location in memory where data is stored. This allows a program to access and manipulate data indirectly through memory addresses.

When a program is executed, all variables are stored in main memory, and each variable is assigned a unique address. Normally, a program accesses a variable by its name. However, in certain situations, it is more efficient or necessary to access the variable through its memory address.

Pointers provide this capability by storing and using addresses instead of direct values.

Thus, a pointer creates a link between the name of a variable and its location in memory, enabling low-level control over data.

Purpose of Pointers

- *To efficiently manage memory, especially when dealing with large data*
- *To allow direct manipulation of memory locations*
- *To enable functions to modify variables defined outside their scope*
- *To support dynamic memory allocation, where memory is assigned during program execution*
- *To form the foundation for advanced features such as arrays, strings, structures, and objects*

Without pointers, many powerful programming techniques in C++ would not be possible.

Characteristics of Pointers

- *A pointer always stores a memory address*
- *It must be declared with a specific data type*
- *The type of pointer determines the type of data it can access*
- *Changes made using a pointer affect the original data*

Declaration and Initialization

*datatype *ptr;* ***// declaration***

ptr = &variable; ***// initialization with address***

Example:

```
int a = 5;
int *p = &a;   // a stores data  pointer p stores the address of a
```

Pointer Operators

1. ***Address-of (&)*** → *gives the address of a variable.*
   Example: `p = &a;`

2. ***Dereference (*)*** → *gives the value stored at the address.*
   Example: `cout << *p;`

Example: Accessing a Variable via Pointer

```
#include <iostream>
using namespace std;
int main()
{
   int a = 10;
   int *p = &a;
   cout << "Address of a: " << p << endl;
   cout << "Value of a: " << *p << endl;
   *p = 20;   // modify value through pointer
   cout << "New value of a: " << a << endl;
   return 0;
}
```

Output:

Address of a: 0x7ffee1c
Value of a: 10
New value of a: 20

Types of Pointers in C++

Type of Pointer	***Definition***	***Syntax / Example***	***Key Points***
Null Pointer	*A pointer that does not point to any valid memory location.*	`int *p = nullptr;`	*• Safe to use• Indicates "points to nothing"*
Void Pointer	*A generic pointer that can store address of any data type.*	`void *vp;vp = &a;*(int*)vp;`	*• Typecasting required• Cannot be dereferenced directly*
Dangling Pointer	*A pointer pointing to memory that has been deallocated.*	`int *p = new int(5);delete p;`	*• Dangerous• Causes undefined behavior*
Wild Pointer	*An uninitialized pointer pointing to garbage memory.*	`int *p;`	*• Contains garbage address• Must be avoided*
Constant Pointer	*A pointer whose address cannot be changed.*	`int *const p = &a;`	*• Address fixed• Value can change*
Pointer to Constant	*A pointer that cannot modify the value it points to.*	`const int *p = &a;`	*• Value fixed• Address can change*
Pointer to Pointer (Double Pointer)	*A pointer that stores the address of another pointer.*	`int **pp = &p;`	*• Used in dynamic memory & data structures*
Function	*A pointer that points to a*	`void (*fp)();`	*• Used for callbacks*

Pointer	*function.*		
Object Pointer	*A pointer that stores the address of an object.*	`item *ptr = &obj;`	• *Access members using* `->`

Advantages

1. *Pointers provide efficient memory management by allocating memory only when required.*
2. *They support dynamic allocation and deallocation of memory during program execution.*
3. *Pointers help in implementing complex data structures such as linked lists, stacks, and trees.*
4. *When used with virtual functions, pointers enable runtime polymorphism and dynamic binding.*

Disadvantages

1. *Incorrect use of pointers can cause runtime errors and program crashes.*
2. *Dangling and wild pointers may lead to undefined behavior by accessing invalid memory locations.*
3. *Pointer-based programs are more difficult to understand, debug, and maintain compared to programs using normal variables.*

1.2 Pointers to Objects in C++

*In C++, **pointers to objects** are pointers that store the **address of an object** instead of the object itself. Just as a normal pointer can point to an integer or a float, a pointer can also point to an object created from a class. Through this pointer, the program can **access the data members and member functions of the object**.*

*An object occupies a specific location in memory. A pointer to an object simply **stores this memory address**. Instead of working with the object directly, the program uses the pointer to refer to the object. This provides flexibility and efficiency, especially when dealing with large objects or dynamic memory.*

*Conceptually, the pointer does not contain the object's data; it only knows **where the object is stored**.*

Importance of Pointers to Objects:

*In C++, an object is stored somewhere in memory. A **pointer to an object** simply stores the **address of that object**. Instead of working with the object directly, the program works with its address. This is useful for many simple reasons.*

1. To avoid copying large objects

*Some objects contain a lot of data. Copying such objects again and again wastes time and memory. Using a pointer means **only the address is passed**, not the whole object. This makes the program faster and more efficient.*

2. To create objects at runtime

Sometimes we do not know in advance how many objects are needed. In such cases, objects are created ***while the program is running****. These objects are accessed using pointers. Without pointers, dynamic object creation would not be possible.*

3. To use the same object in many places

Pointers allow ***many functions or parts of a program to use the same object****. When one part changes the object, the change is seen everywhere because all pointers refer to the same object.*

4. To support inheritance and polymorphism

Pointers to objects make it possible for a ***base class pointer to point to a derived class object****. This helps C++ decide which function to call at runtime. This feature is very important in object-oriented programming.*

5. To link objects together

In real programs, objects often need to be connected to each other (like nodes in a list or tree). Pointers help one object ***refer to another object****, making such structures possible.*

General Syntax

```
ClassName obj;
ClassName *ptr;

ptr = &obj;       // pointer stores address of object
ptr->function();  // access members using pointer
```

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

class Student {
public:
    int roll;

    void setData(int r) {
        roll = r;
    }

    void showData() {
```

```
        cout << "Roll Number: " << roll << endl;
    }
};

int main() {
    Student s1;         // normal object
    Student *ptr;       // pointer to object

    ptr = &s1;          // pointer stores address of object

    ptr->setData(101);  // access member using pointer
    ptr->showData();

    return 0;
}
```

Advantages of Pointer to Object in C++

1. ***Dynamic memory management:*** *Pointers allow objects to be created and destroyed at runtime using* `new` *and* `delete`*, providing flexible memory usage.*
2. ***Efficient passing of objects:*** *Objects can be passed to functions by reference using pointers, avoiding unnecessary copying of large objects.*
3. ***Supports runtime polymorphism:*** *Base-class pointers can point to derived-class objects, enabling dynamic binding through virtual functions.*
4. ***Handling dynamic arrays:*** *Pointers make it possible to create and manage arrays of objects whose size is not known at compile time.*
5. ***Object sharing:*** *Multiple parts of a program can access and modify the same object using pointers without creating duplicate copies.*

1.3 this Pointer

In C++, every non-static member function of a class has access to a special pointer known as the this pointer. The this pointer is an implicit pointer, meaning it is automatically provided by the compiler and does not need to be declared by the programmer.

The this pointer always holds the address of the object that calls the member function. Whenever a member function is invoked using an object, the compiler internally passes the address of that object to the function through the this pointer. As a result, the function can correctly access the data members of the calling object.

In simple terms, the this pointer helps the compiler determine which object's data members should be accessed when the same member function is used by multiple objects.

Need for the this Pointer:

The this pointer is used in C++ to identify the object that is calling a member function. Since the same member function can be used by many objects of a class, the this pointer helps the program understand which object's data should be used.

- ***To refer to the calling object***
  When a member function is called, this points to the object that made the call. This allows the function to work with the correct object's data.
- ***To avoid confusion between variables***
  Sometimes, function parameters or local variables have the same name as class data members. The this pointer clearly tells the program that we are referring to the object's data member, not the local variable.
- ***To return the current object***
  The this pointer allows a function to return the same object that called it. This is useful when multiple functions are called one after another using the same object.
- ***To pass the current object to another function***
  Using the this pointer, the calling object can be sent to another function as an argument, so that the other function can access its data.

Characteristics of the this Pointer

- *It is automatically created by the compiler*
- *It is available only inside non-static member functions*
- *It always points to the current calling object*
- *It cannot be modified to point to another object*

Simple Example of `this` Pointer

```
#include <iostream>
using namespace std;
class Student
{
   int roll;
   string name;
public:
   void setData(int roll, string name)
  {
 // using 'this' to resolve ambiguity between data members and parameters
      this->roll = roll;
      this->name = name;
   }
   void display()
  {
```

```
        cout << "Roll Number: " << roll << ", Name: " << name << endl;
    }
};
int main()
 {
    Student s1, s2;
    s1.setData(101, "Rahul");
    s2.setData(102, "Anita");
    s1.display();
    s2.display();
    return 0;
}
```

Output:

Roll Number: 101, Name: Rahul
Roll Number: 102, Name: Anita

1.4 Pointer to Derived Classes

*In C++, a **base class pointer** can point to an object of a **derived class**. This feature plays an important role in inheritance and polymorphism. When a base class pointer refers to a derived class object, it can access only the members of the base class. Runtime polymorphism is achieved when such pointers are used along with **virtual functions**.*

Rules

1. *A base class pointer can point to a derived class object.*
2. *Through a base class pointer, only base class members can be accessed.*
3. *If the function is declared as virtual, the derived class version is executed at runtime (runtime polymorphism).*
4. *If the function is non-virtual, the base class version is executed (compile-time binding).*

Example: Pointer to Derived Class

```
#include <iostream>
using namespace std;
class Base
{
public:
    void display()
    {
        cout << "This is Base class" << endl;
    }
};
class Derived : public Base
```

```
{
public:
   void display()
   {
      cout << "This is Derived class" << endl;
}

};
int main()
{
   Base *bptr;     // base class pointer
   Derived d;
   bptr = &d;      // base pointer pointing to derived object
   bptr->display(); // calls base class function
   return 0;
}
```

Output

This is Base class

Although the pointer bptr points to a derived class object, the function display() is not virtual in the base class. Therefore, compile-time binding takes place and the base class version of the function is called.

2.File Handling in C++

2.1 Files

*Files provide a way to store data **permanently** on a storage device such as a hard disk. Unlike console input and output, which exist only during program execution, files allow data to be saved and reused later.C++ provides file handling mechanisms that allow programs to **store output into files** and **read data from files** available on the disk.*

The <iostream> header supports console input and output through predefined objects like cin and cout. However, it does not support file operations. For file handling, C++ provides the <fstream> header file, which defines special classes used to read from and write to files.

The important file handling classes provided by <fstream> are:

- ***ifstream – used for reading data from files***
- ***ofstream – used for writing data to files***
- ***fstream – used for both reading and writing data***

File Streams

*In C++, all input and output operations are performed using **streams**. A stream is a sequence of bytes that flows between a program and an input/output device.*
When dealing with files:

- *An **input stream** transfers data from a file to the program.*
- *An **output stream** transfers data from the program to a file.*

*A file stream acts as an **interface between the program and the file**. File input/output operations are carried out in a manner similar to console input/output operations.*

***Ifstream** : ifstream is an input file stream class used to read data from files. It opens files in input mode by default and allows a program to retrieve data stored in files. Functions such as get(), getline(), and read() are commonly used with this class to read characters, lines, or blocks of data.*

Example

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   ifstream ifs("data.txt");
   string text;
   ifs >> text;
   cout << text;
   ifs.close();
   return 0;
}
```

Ofstream : ofstream is an output file stream class used to write data into files. It opens files in output mode by default and allows a program to store information permanently in a file. If the file does not exist, it creates a new file; if it exists, it can overwrite or append data depending on the mode used.

Example

```
#include <fstream>
using namespace std;
int main()
{
   ofstream ofs("data.txt");
   ofs << "Writing to file";
   ofs.close();
   return 0;
}
```

***Fstream :** fstream is a file stream class that supports both input and output operations. It combines the features of ifstream and ofstream, allowing a program to read from and write to the same file. This class is useful when file data needs to be modified or updated.*

Example

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   fstream fs("data.txt", ios::in | ios::out);
   fs << "Hello";
   fs.seekg(0);
   string text;
   fs >> text;
   cout << text;
   fs.close();
   return 0;
}
```

2.2 Operations on Files

File operations in C++ allow a program to store data permanently and retrieve it when required. The most common file operations are opening a file, closing a file, writing data to a file, reading data from a file, and detecting the end of a file.

Opening a File

Before performing any read or write operation, a file must be opened. Opening a file establishes a connection between the program and the file on the disk. In C++, files are opened using ifstream, ofstream, or fstream objects along with the open() function. The file can be opened in different modes such as read, write, append, or a combination of these.

Example

```
ofstream ofs;
ofs.open("data.txt", ios::out);
```

Closing a File

After completing file operations, the file should be closed to release system resources and ensure that all data is properly saved. Although files are automatically closed when the program terminates, explicitly closing files is considered good programming practice.

Example

```
ofs.close();
```

Writing Data to a File

Writing data to a file allows a program to store information permanently. In C++, this operation is performed using an ofstream or fstream object instead of cout. The data is written into the file using insertion operators (<<) or write functions.

Example

```
ofstream ofs("data.txt");
ofs << "Welcome to File Handling";
```

```
ofs.close();
```

Reading Data from a File

Reading data from a file allows a program to retrieve previously stored information. This operation is performed using an ifstream or fstream object instead of cin. Data can be read character by character, word by word, or line by line.

Example

```
ifstream ifs("data.txt");
string text;
ifs >> text;
cout << text;
ifs.close();
```

Detecting End of File

Detecting the end of a file is important to avoid reading beyond the available data. C++ provides the eof() function, which returns true when the end of the file is reached. This function is commonly used in loops while reading data from a file.

Example

```
ifstream ifs("data.txt");
while (!ifs.eof())
{
   // read data
}
ifs.close();
```

Usage

Used to determine when file reading should stop.

Program: Demonstration of All File Operations in C++

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   char text[100];

   // -------- Writing to a File --------
   ofstream ofs;
   ofs.open("example.txt");

   if (!ofs)
   {
      cout << "Error opening file for writing";
      return 0;
   }
```

```
    cout << "Enter a line to write into the file: ";
    cin.getline(text, 100);

    ofs << text << endl;   // write data to file
    ofs.close();           // close the file

    // -------- Reading from a File --------
    ifstream ifs;
    ifs.open("example.txt");

    if (!ifs)
    {
       cout << "Error opening file for reading";
       return 0;
    }

    cout << "\nReading data from file:\n";

    // Detecting End of File
    while (!ifs.eof())
    {
       ifs.getline(text, 100);
       cout << text << endl;
    }

    ifs.close();   // close the file

    cout << "\nEnd of file reached";

    return 0;
}
```

Output (Sample)

Enter a line to write into the file: File handling in C++

Reading data from file:
File handling in C++

End of file reached

2.3 File Modes in C++

File modes in C++ define the manner in which a file is accessed and manipulated by a program. When a file is opened, the file mode determines what operations are permitted on the file and how the existing contents of the file are treated. In simple terms, file modes act as rules or permissions that govern file usage.

Because files are stored on secondary memory and retain their data permanently, incorrect access can lead to data loss or corruption. File modes ensure that the program interacts with the file safely, correctly, and intentionally.

Need for File Modes

File modes are required to:

- *Specify whether a file is opened for reading, writing, or both*
- *Control whether existing data is preserved, overwritten, or extended*
- *Determine the initial position of file pointers*
- *Prevent unintended modification of file contents*

Without proper file modes, a program would not know how to treat the file, leading to unpredictable results.

Classification of File Modes

File modes can be conceptually grouped based on their purpose:

1. Read Mode

Read mode allows the program to access data already stored in a file without modifying it. The file must exist, and the program can only retrieve information. This mode ensures data safety, as no changes are permitted.

2. Write Mode

Write mode allows the program to store data in a file. If the file already contains data, this mode may remove the existing contents before writing new information. Therefore, write mode is powerful but must be used carefully to avoid data loss.

3. Append Mode

Append mode allows the program to add new data at the end of the file. Existing data remains unchanged. This mode is useful when data needs to be continuously added, such as maintaining logs or records.

4. Update Mode (Read and Write)

Update mode allows both reading and writing operations on the same file. This mode is essential for modifying specific parts of a file, as the program can read existing data, reposition the file pointer, and then write updated data at the required location.

5. Binary Mode

Binary mode treats the file as a raw sequence of bytes rather than formatted text. Data is stored and retrieved exactly as it exists, without any interpretation or transformation. This mode is required for non-text files such as images, audio, or executable data.

File Mode	*Description*	*Purpose / Usage*
ios::in	*Opens a file for reading*	*Used to read data from an existing file*
ios::out	*Opens a file for writing*	*Used to write data to a file; creates a new file if it does not exist*
ios::app	*Opens a file in append mode*	*Used to add data at the end of an existing file*
ios::ate	*Opens file and moves pointer to end*	*Allows reading and writing from any position*
ios::trunc	*Deletes existing file contents*	*Used to clear old data before writing new data*
ios::binary	*Opens file in binary mode*	*Used for non-text files like images and videos*

2.4 File Pointers in C++

*File pointers in C++ are **special internal indicators** used to keep track of the **current position within a file** during input and output operations. When a program performs file operations, it does not access the entire file at once. Instead, it reads from or writes to the file **at a particular position**, which is controlled and monitored by file pointers.*

*Since files are stored on **secondary storage devices**, such as hard disks, data access is slower and must be performed in an organized manner. File pointers enable this by remembering **where the previous read or write operation ended** and determining **where the next operation should begin**. This ensures smooth and orderly file processing.*

Role of File Pointers in File Updating

File pointers are especially important when updating files:

- *To overwrite existing data, the pointer must be moved to the correct position*
- *To append data, the pointer is moved to the end of the file*
- *To modify specific records, pointers help locate the exact byte position*

Types of File Pointers

*C++ maintains **two independent file pointers**, each designed for a specific type of operation:*

1. Get Pointer (Input Pointer)

*The get pointer is used for **reading data from a file**. It indicates the position from which the next read operation will take place. After data is read, the get pointer automatically advances by the number of bytes read, preparing the file for the next input operation.*

Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
 ifstream ifs("data.txt");
 cout << "Current position: " << ifs.tellg() << endl;
ifs.seekg(5);
   cout << "New position: " << ifs.tellg() << endl;
   ifs.close();
   return 0;
}
```

2. Put Pointer (Output Pointer)

*The put pointer is used for **writing data to a file**. It specifies the location where new data will be written. Once data is written, the put pointer moves forward, ensuring that subsequent data is written at the correct position.*

These pointers may operate independently when a file is opened for both reading and writing.

Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream ofs("data.txt");
   cout << "Current position: " << ofs.tellp() << endl;
   ofs.seekp(10);
   ofs << "Hello";
   ofs.close();
   return 0;
}
```

2.5 Updating a File in C++

Updating a file in C++ refers to the process of ***altering the data that is already stored in a file****. Unlike creating a new file, updating works on an existing file and may involve changing, replacing, or adding information while preserving the rest of the data.*

Files are kept on ***secondary storage devices*** *such as hard disks or SSDs, which means their contents remain available even after the program ends. Because this storage is permanent, any modification must be done carefully. An incorrect update can permanently overwrite useful information or make the file inconsistent. Therefore, file updating requires controlled access and proper positioning within the file.*

C++ does not allow direct modification of file contents in the same way as variables in memory. Instead, the program interacts with a file through ***streams****, which act as a connection between the program and the file. These streams manage the flow of data during reading and writing operations.*

The <fstream> library in C++ provides the necessary tools for this purpose. It supports input, output, and combined input–output operations on files. For updating a file, the program typically uses a stream that allows ***both reading and writing****, so existing data can be examined and then modified at the required location.*

updating a file in C++ is a controlled operation that involves:

- *Opening an existing file with appropriate access permissions*
- *Locating the position where changes are required*
- *Modifying the data without disturbing unrelated contents*
- *Safely closing the file to ensure changes are stored permanently*

Role of File Pointers in File Updating

File updating relies heavily on correct management of file pointers. To update existing data in a file:

1. *The file pointer is moved to the exact position where the data to be modified is stored.*
2. *New data is written at that location.*
3. *Only the intended portion of the file is altered, while the remaining data remains unchanged.*

If file pointers are not handled properly, updates may occur at incorrect positions, which can lead to ***data corruption, loss of information, or inconsistent files****.*

In C++, updating is generally done with the fstream class, which supports both input (read) and output (write).

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

int main()
{
   fstream file;

   // Step 1: Create a file and write some initial data
   file.open("sample.txt", ios::out);  // write mode
   file << "Hello World!";
   file.close();

   // Step 2: Append new data at the end of the file
   file.open("sample.txt", ios::out | ios::app);  // append mode
   file << "\nThis is appended text.";
   file.close();

   // Step 3: Modify existing content (overwrite from 6th position)
   file.open("sample.txt", ios::in | ios::out);  // read + write
   file.seekp(6);   // Move write pointer after "Hello "
   file << "C++ Users";  // Overwrites "World!" with "C++ Users"
   file.close();

   // Step 4: Read final content
   file.open("sample.txt", ios::in);
   string line;
   cout << "Final File Content:\n";
   while (getline(file, line))
{
      cout << line << endl;
```

```
    }
    file.close();
    return 0;
}
```

Output:
Hello C++ Users
This is appended text.

3.Exception Handling

3.1. Introduction to Exception Handling

*An **exception** refers to an unexpected or abnormal condition that arises during the execution of a program. Such unusual conditions may result from logical faults, invalid input data, or environmental failures, and can cause errors that lead to abnormal termination of the program.*

*In C++, the mechanism used to detect and handle such runtime errors in a structured manner is known as **exception handling**. This mechanism allows the programmer to transfer program control from the point where an error occurs to another part of the program that is designed to handle the error.*

2. Classification of Exceptions

*In general, exceptions are classified into **synchronous exceptions** and **asynchronous exceptions** based on their origin.*

2.1 Synchronous Exceptions

Synchronous exceptions occur during program execution due to errors within the program logic or improper input data. These exceptions arise as a direct result of executing specific program statements and are therefore predictable.

Examples of synchronous exceptions include:

- *Array index out of range*
- *Arithmetic overflow*
- *Arithmetic underflow*
- *Division by zero*

2.2 Asynchronous Exceptions

Asynchronous exceptions are caused by events that are external to the program and beyond the control of the programmer. These exceptions are not directly related to the execution of program instructions.

Examples of asynchronous exceptions include:

- *Keyboard interrupts*
- *Hardware malfunctions*
- *Disk failure*
- *Power failure*

3.2 Exception Handling Model

When a program encounters an abnormal condition for which it is not designed, control must be transferred to a specialized section of the program that can handle the problem effectively. This transfer of control is achieved by ***throwing an exception****.*

The C++ exception handling mechanism is based on three fundamental constructs:

- ***try***
- ***throw***
- ***catch***

These constructs together form the ***exception handling model****.*

Working of the Exception Handling Model

The execution of the exception handling model proceeds as follows:

1. *A block of code that may generate an exception is enclosed within a* `try` *block.*
2. *If an abnormal condition is detected, an exception is generated using the* `throw` *statement.*
3. *Program control is immediately transferred to the appropriate* `catch` *block.*
4. *The* `catch` *block handles the exception and prevents abnormal termination of the program.*

❖ *Try Block*

Each `try` *block must be followed immediately by one or more* `catch` *blocks. The program should attempt to catch any exception thrown by a function.*

The ***`try` block*** *is a fundamental component of the C++ exception handling mechanism. It defines a* ***protected region of code*** *in which exceptions may occur. All statements that are likely to generate runtime errors are enclosed within the* `try` *block. If an exception occurs inside this block, the normal flow of execution is interrupted and control is transferred to the appropriate* `catch` *block.*

Syntax of `try` Block

```
try
{
   // statements that may generate an exception
}
catch (exception_type)
{
```

```
    // exception handling statements
}
```

Note:

- A `try` block must always be followed by at least one `catch` block.
- Multiple `catch` blocks can follow a single `try` block.

❖ Catch Block

*The **catch block** is a key component of the C++ exception handling mechanism. It is used to handle exceptions that are thrown inside a try block. When an exception occurs, program control is transferred from the try block to the matching catch block based on the exception type. The catch block contains the statements required to handle the error gracefully and prevent abnormal program termination. Each catch block specifies the type of exception it can handle, and multiple catch blocks can be used to handle different types of exceptions.*

Syntax of catch Block

```
catch (exception_type variable)
{
    // exception handling code
}
```

❖ Throw Keyword

*The **`throw` keyword** is used in C++ to **explicitly generate an exception**. It is an integral part of the exception handling mechanism and is generally used inside a `try` block or within a function called by a `try` block. When an abnormal or error condition is detected during program execution, the `throw` statement is used to transfer control to an appropriate `catch` block.*

When a `throw` statement is executed, the normal flow of the program is immediately terminated, and the exception is passed to the nearest matching `catch` block based on the type of exception thrown. This mechanism helps in handling runtime errors efficiently and prevents abrupt termination of the program.

Syntax of throw

```
throw exception_type;
or
throw exception_object;
```

Program try, throw, and catch

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
```

```
    cout << "Enter two numbers: ";
    cin >> a >> b;
    try
    {
        if (b == 0)
        {
            throw b;   // throwing an exception
        }
        else
        {
            cout << "Result = " << a / b;
        }
    }
    catch (int e)
    {
        cout << "Error: Division by zero is not allowed";
    }
    return 0;
}
```

*In this program, the statements that may cause a runtime error are placed inside the **try block**. If the value of `b` is zero, a division by zero error may occur. To prevent this, the program uses the **throw** statement to explicitly raise an exception when `b` equals zero.*

*When the exception is thrown, the normal flow of execution is interrupted, and control is transferred to the **catch block**. The catch block catches the thrown exception and displays an appropriate error message. If no exception occurs, the division operation is performed normally.*

Output

```
Enter two numbers: 10 0
Error: Division by zero is not allowed
```