VIVEKANANDA GLOBAL UNIVERSITY, JAIPUR

(Established by Act 11/2012 of Rajasthan Govt. Covered u/s 2(f) of UGC Act, 1956)

M.Sc. Mathematics

Object Oriented Programming with C++

SEMESTER I

Author: Dr. Ankit Dhamija

Approval Director CIQA: 27th July 2023

Approval of Academic Council: 28th July 2023

For Vivekananda Global University

Registrar

Unit – 1 PROGRAMMING PARADIGM

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Understand the different paradigms for problem solving: Gain a comprehensive understanding of procedural, functional, and logic programming paradigms, and their respective strengths and use cases.
- Recognize the need for Object-Oriented Programming (OOP): Explore the reasons why OOP has become a widely adopted approach in software development, including its modular structure, code reusability, and suitability for real-world problem-solving.
- Differentiate between Object-Oriented Programming and Procedure Oriented Programming: Understand the key differences between OOP and POP, including their approaches to code organization, data management, and code reusability.
- Grasp the concept of abstraction: Learn how abstraction allows you to focus on essential features while hiding unnecessary details, enabling the creation of higher-level models and enhancing code maintainability and understandability.
- Gain an overview of Object-Oriented Programming principles: Familiarize yourself with the fundamental principles of OOP, including encapsulation, inheritance, and polymorphism, and understand their significance in software design.
- Explore the principle of encapsulation: Deepen your understanding of encapsulation as a means to bundle data and behavior within objects, promote data integrity, control access to data, and enhance code modularity and reusability.
- Understand the concept of inheritance: Discover how inheritance enables code reuse, promotes modularity, and facilitates the creation of class hierarchies, allowing objects to inherit properties and behaviors from parent classes.
- Learn about polymorphism: Explore the concept of polymorphism and its ability to treat objects of different classes as interchangeable entities, providing flexibility and adaptability in software design.

Introduction

In the ever-evolving field of computer science and software development, problem-solving lies at the core of every successful program. Over the years, different paradigms have emerged to tackle complex problems efficiently and provide scalable solutions. One such paradigm that has gained significant popularity and revolutionized the way we approach software development is Object-Oriented Programming (OOP). Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. In OOP, objects encapsulate data and behavior, allowing for the modeling of real-world entities or abstract concepts. At its core, OOP emphasizes the concept of objects interacting with each other to solve problems. Each object has its own state, defined by its attributes or properties, and behavior, defined by its methods or functions. Objects can communicate with each other through messages or method calls, enabling collaboration and modularity in software development.

To begin with, it is essential to understand the various paradigms for problem-solving. Each paradigm represents a different approach to structuring and organizing code to address specific challenges efficiently. From the structured approach of POP to the object-centric nature of OOP, these paradigms offer unique perspectives and tools to simplify software development. Next, we will explore why OOP has become a prominent paradigm in modern software development. OOP offers several advantages over POP, primarily driven by its ability to model real-world entities as objects. By encapsulating data and behavior within these objects, OOP promotes code reusability, modularity, and maintainability. This chapter will elucidate these advantages and highlight the need for OOP in contemporary software engineering. Abstraction, one of the key concepts in OOP, allows developers to focus on essential features while hiding unnecessary details. We will discuss how abstraction helps in managing complexity and enables the creation of robust and scalable software solutions. Furthermore, this ebook will provide an overview of the fundamental principles of OOP. We will explore encapsulation, a mechanism that enables bundling of data and methods together within an object, resulting in data protection and improved code organization. Additionally, we will dive into the concept of inheritance, which facilitates code reuse and hierarchical relationships between objects. Lastly, we will explore polymorphism, a powerful feature that allows objects of different types to be treated uniformly, enhancing flexibility and extensibility in software design.

1.1 Programming Paradigm

A programming paradigm is a fundamental approach or style of programming that provides a framework for designing and structuring computer programs. It encompasses a set of principles, concepts, and techniques that guide how problems are solved and how programs are written.

Different programming paradigms have their own distinct rules, methodologies, and patterns for organizing code and data. Following are some of the commonly recognized programming paradigms:

a) **Imperative Programming:** This paradigm focuses on describing the steps or instructions necessary to solve a problem. Programs in this paradigm consist of a series of statements that modify the program state. It is based on the notion of changing program state through a series of statements that specify how to manipulate variables and data structures. Imperative programming treats a program as a set of commands or instructions that are executed in order, with control flow structures like loops and conditionals guiding the execution path.

   Key features and concepts of imperative programming include:

   - State and Variables: Imperative programming involves maintaining and modifying the program state by working with variables. Variables can hold values that can be changed throughout the program's execution.
   - Assignment Statements: Imperative programs use assignment statements to update the values of variables. These statements assign new values to variables or modify their existing values.

- Control Flow: Imperative programming employs control flow structures to determine the order in which statements are executed. This includes loops (such as for, while) for repetition and conditionals (such as if-else) for decision-making.
- Procedures and Subroutines: Imperative programming encourages organizing code into reusable procedures or subroutines. These are blocks of code that can be called and executed at different points in the program, promoting code modularity and reusability.
- Mutable State: In imperative programming, program state can be modified throughout execution. This means that variables can be changed, leading to side effects that impact the program's behavior.

Languages like C, Pascal, Fortran, and early versions of BASIC are primarily imperative programming languages. Even languages that support multiple paradigms, such as Python and Java, have an imperative core. Imperative programming is well-suited for tasks that involve explicit step-by-step instructions and precise control over program state and execution flow.

While imperative programming is powerful and widely used, it can be prone to issues like mutable state management, making it more challenging to reason about complex programs. As a result, alternative paradigms like functional programming and object-oriented programming have gained popularity as they offer different approaches to problem-solving and code organization

b) **Object-Oriented Programming (OOP):** Object-oriented programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes that encapsulate data and behavior. It provides a way to structure code by modeling real-world entities or abstract concepts as objects, allowing for modularity, reusability, and easy maintenance of code. OOP revolves around the concept of objects, which are instances of classes that encapsulate data and behavior. This paradigm emphasizes modularity, encapsulation, inheritance, and polymorphism. Languages like Java, C++, and Python provide support for OOP.

c) **Functional Programming**: In functional programming, programs are built using pure mathematical functions that avoid mutable state and side effects. Functions are treated as first-class citizens and can be passed as arguments, returned as results, and stored in variables. Functional programming languages include Haskell, Lisp, and Erlang.

Key concepts and features of functional programming include:

- Pure Functions: Pure functions are functions that produce the same output for a given set of inputs and have no side effects. They do not modify external state or variables and rely solely on their inputs to produce the output. Pure functions are deterministic and easier to reason about, test, and parallelize.
- Immutability: Functional programming promotes the use of immutable data structures, where values cannot be modified after creation. Instead of changing existing data, functional programs create new data structures that represent updated or transformed versions. Immutability ensures that data remains consistent and avoids unintended side effects.
- Recursion: Recursion is a fundamental technique in functional programming, where functions can call themselves. It allows for elegant and concise solutions to repetitive or complex problems by breaking them down into smaller subproblems.
- Declarative Style: Functional programming emphasizes a declarative style of programming, focusing on "what" needs to be done rather than "how" it should be done. Programs describe the desired results or transformations, leaving the evaluation details to the programming language or runtime.
- Functional programming languages, such as Haskell, Lisp, Erlang, and parts of languages like JavaScript, Python, and Scala, provide built-in support for functional programming concepts.

d) **Procedural Programming**: Procedural programming is similar to imperative programming, but it focuses on procedures or subroutines. Programs are organized into reusable procedures that perform specific tasks. It is based on the idea of breaking down a problem into a sequence of steps or procedures that are executed in order. Procedural programming emphasizes clear, step-by-step instructions and the use of variables to manipulate data. C and Pascal are examples of languages that primarily support procedural programming.

Key features and concepts of procedural programming include:

- Procedures: Procedures, also known as functions or subroutines, are blocks of code that perform a specific task or calculation. They are reusable and modular, allowing code to be organized into smaller, self-contained units.
- Sequential Execution: Procedural programs follow a linear execution flow, where statements are executed one after another in the order they appear. Control flow structures, such as loops and conditionals, allow for conditional branching and repetition.
- Variables: Procedural programming utilizes variables to store and manipulate data. Variables can hold different types of values and can be assigned new values throughout the program's execution.
- Modularity and Reusability: Procedural programming promotes modularity by dividing code into procedures, making it easier to understand, maintain, and reuse. Procedures can be called from different parts of the program, reducing code duplication.
- Procedural Abstraction: Procedural programming allows programmers to focus on the procedures and their functionality rather than the underlying implementation details. The procedural abstraction provides a high-level view of the program's functionality.
- Data and Procedure Separation: In procedural programming, there is a separation between data and procedures. Procedures manipulate data through parameters and local variables, keeping data and procedures distinct.

e) **Logic Programming:** Logic programming is based on formal logic and rules. Programs are written in terms of logical statements and relationships. It focuses on describing a problem as a set of logical statements and relationships, rather than providing explicit instructions or procedures. In logic programming, programs consist of a collection of logical facts and rules, and the computation is performed through logical inference. The most prominent logic programming language is Prolog.

Key concepts and features of logic programming include:

- Logic Rules: Logic programming uses logical rules, typically in the form of Horn clauses, to express relationships and constraints. These rules define logical implications and allow for reasoning and inference.
- Logical Facts: Logical facts represent information or assertions about the problem domain. Facts consist of atomic statements that describe properties or relationships between objects or entities.
- Unification: Unification is a fundamental operation in logic programming. It involves matching logical patterns or queries with available facts and rules to find solutions or evaluate queries. Unification is used to bind variables and instantiate them with appropriate values.
- Backtracking: Logic programming supports backtracking, which allows the program to explore multiple solutions to a problem. If a particular rule or fact does not yield a solution, the program can backtrack and explore other possible paths or choices.
- Rule-based Reasoning: Logic programming enables rule-based reasoning, where the program can infer new knowledge or deduce conclusions based on the provided logical rules and facts. This makes logic programming useful in domains that involve logical inference, such as expert systems and theorem proving.

f) **Event-Driven Programming:** This paradigm centers around responding to events or user actions. Programs are structured to handle events asynchronously, triggering appropriate actions when specific events occur. Event-driven programming is commonly used in graphical user interfaces (GUIs) and web development. JavaScript is an example of a language that facilitates event-driven programming.

1.2 Need for OOP

Object-oriented programming (OOP) provides a structured and modular approach to software development. It offers several advantages and addresses various needs in software engineering. Here are some key reasons for the need of OOP:

- Modularity and Code Organization: OOP allows breaking down complex systems into smaller, self-contained modules called objects. Each object encapsulates its own data and behavior, providing a clear and modular structure to the code. This modularity promotes code reusability, maintainability, and ease of understanding.
- Encapsulation and Data Hiding: OOP emphasizes encapsulation, which involves bundling data and methods together within an object and hiding the internal details from the outside world. This protects the integrity and consistency of the data by preventing direct access and modification. Encapsulation ensures that the object's state is controlled and accessed only through defined interfaces, improving data security and reducing the likelihood of errors.
- Code Reusability and Maintainability: OOP encourages the reuse of existing code components. Through inheritance, objects can inherit properties and behaviors from existing classes, promoting code reuse and minimizing redundant code. This reduces development time, enhances code maintainability, and simplifies updates and bug fixes.
- Polymorphism and Flexibility: Polymorphism, a core principle of OOP, allows objects of different types to be treated uniformly through a common interface. This enables writing generic code that can operate on objects of various classes. Polymorphism increases flexibility, modifiability, and extensibility, as new classes can be added without affecting existing code.
- Abstraction and Simplification: OOP encourages the use of abstraction to model real-world entities or concepts into classes. Abstraction focuses on essential features while hiding unnecessary details, simplifying the complexity of the system. It provides a high-level view of the problem domain, making the code easier to understand, design, and maintain.

- Collaboration and Teamwork: OOP facilitates collaboration among developers working on a project. Through the use of classes and objects, different team members can work independently on different components of the system, as long as they adhere to the defined interfaces. OOP's modular and encapsulated nature allows for parallel development, promoting efficient teamwork.
- Scalability and Extensibility: OOP provides a foundation for scalable and extensible systems. New classes and objects can be added to accommodate changing requirements without major modifications to the existing codebase. This allows software systems to grow and evolve over time, adapting to new functionalities and user needs.
- Real-world Modeling: OOP aligns with real-world modeling by allowing developers to represent entities, relationships, and behaviors in software systems that mirror the real world. This promotes better understanding, analysis, and design of the problem domain, leading to more intuitive and maintainable code.

1.3 Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. It provides a way to structure and design software applications by representing real-world objects and their interactions.

In OOP, objects are the fundamental building blocks. They encapsulate data (attributes or properties) and behavior (methods or functions) into a single unit. Objects can communicate with each other by invoking methods or accessing properties, enabling interaction and collaboration.

The key concepts in object-oriented programming are:

Classes: A class is a blueprint or template for creating objects. It defines the properties and methods that objects of that class will have. For example, if we have a class called "Car," objects of this class will represent individual cars, and the class will define the common characteristics and behaviors that all cars share.

Objects: Objects are instances of classes. They are created from a class blueprint and have their own state (values of attributes) and behavior (methods to perform actions). Each object can have different values for its attributes while still following the structure defined by the class.

Encapsulation: Encapsulation is the practice of hiding the internal details of an object and exposing only the necessary information and functionality. It helps maintain data integrity and provides a clean interface for interacting with objects. Access to object attributes and methods can be controlled through access modifiers (e.g., public, private, protected) to enforce proper usage.

Inheritance: Inheritance allows the creation of new classes based on existing classes. It enables the reuse of code and the creation of class hierarchies. A subclass (derived class) inherits the attributes and methods of its superclass (base class or parent class) and can extend or modify their behavior. Inheritance promotes code reusability and supports the "is-a" relationship.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It allows for the use of a single interface to represent different types of objects, providing flexibility and extensibility in the code. Polymorphism is achieved through method overriding (redefining a method in a subclass) and method overloading (providing multiple methods with the same name but different parameters).

Abstraction: Abstraction focuses on defining the essential characteristics and behavior of an object, while hiding the implementation details. It allows developers to create abstract classes or interfaces that provide a common interface for a group of related objects. Abstraction helps manage complexity and promotes modularity and flexibility in design.

1.3.1 Abstraction

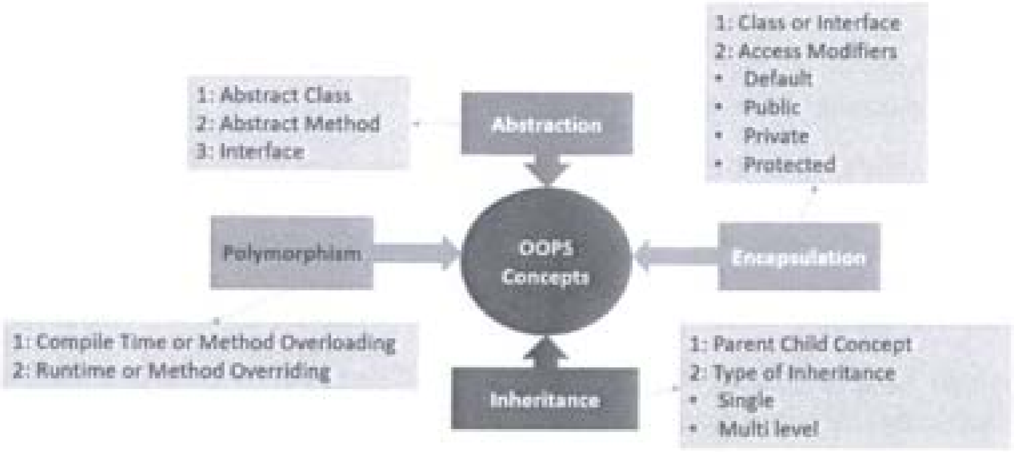
Abstraction is a fundamental concept in object-oriented programming that focuses on representing essential features and behaviors of objects while hiding unnecessary details and complexities. It allows developers to create abstract classes or interfaces that define a common interface or contract for a group of related objects.

The purpose of abstraction is to simplify the complexity of a system by providing a high-level view and hiding the implementation details. It enables programmers to focus on the essential aspects of an object or a system without getting entangled in the intricate internal workings.

In practice, abstraction involves identifying the essential properties and behaviors that define an object's functionality and ignoring the non-essential or implementation-specific details. These essential aspects become the interface or contract through which other objects can interact with the abstracted object. By using abstraction, developers can create modular and extensible systems. Abstract classes and interfaces provide a clear separation of concerns and allow for code reusability. They promote loose coupling between objects and enable flexibility in design and implementation.

Abstraction is closely related to other OOP principles like encapsulation and inheritance. It helps in achieving encapsulation by hiding internal details and providing a well-defined interface for interaction. It also supports inheritance by allowing the creation of abstract classes that can be extended by concrete subclasses.

1.4 Principles of Object Oriented Programing



OOPS Concepts

Figure 1: OOPS Concepts

The principles of Object-Oriented Programming (OOP) guide the design and implementation of software using the object-oriented paradigm. These principles help developers create modular, maintainable, and flexible code. Here are the key principles of OOP:

Encapsulation: Encapsulation is the principle of bundling data and related behaviors (methods/functions) into objects. It involves hiding the internal details of an object and providing controlled access to its data through public methods. Encapsulation ensures data integrity, promotes code reusability, and reduces code coupling.

Inheritance: Inheritance allows objects/classes to inherit properties and behaviors from parent classes. It enables code reuse by defining a hierarchy of classes, where subclasses inherit and extend the attributes and methods of their parent classes. Inheritance promotes code extensibility, modularity, and the implementation of the "is-a" relationship.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It allows for the use of a single interface to represent multiple types of objects. Polymorphism enables flexibility in code design, as different objects can respond differently to the same method call based on their specific implementation.

Abstraction: Abstraction involves focusing on essential characteristics and behaviors while hiding unnecessary details. It provides a higher-level view of objects by defining abstract classes or interfaces that specify common behavior. Abstraction allows developers to create models and design software based on general concepts and ideas, without getting into implementation specifics.

Modularity: Modularity is the principle of breaking down complex systems into smaller, self-contained modules or objects. Each module focuses on a specific functionality or aspect of the system. Modularity promotes code organization, ease of maintenance, and reusability by allowing independent development, testing, and integration of modules.

Composition: Composition is the principle of creating complex objects by combining simpler objects or components. It enables objects to be composed of other objects, creating a "has-a" relationship. Composition promotes code reuse, flexibility, and modularity by allowing objects to collaborate and delegate tasks to other objects.

Single Responsibility Principle (SRP): SRP states that a class or module should have only one reason to change. It emphasizes separating concerns and ensuring that each class or module has a single responsibility. This principle enhances code maintainability, readability, and reduces the impact of changes.

Open/Closed Principle (OCP): The OCP states that software entities (classes, modules, functions) should be open for extension but closed for modification. It encourages designing code that can be easily extended with new functionality without modifying existing code. This principle promotes code stability, modularity, and supports the concept of interfaces and abstract classes.

1.4.1 Encapsulation

Encapsulation is a fundamental principle of object-oriented programming (OOP) that promotes the bundling of data and related behaviors (methods/functions) into objects. It involves hiding the internal details of an object and providing controlled access to its data through public methods or interfaces. Encapsulation ensures data integrity, promotes code reusability, and reduces code coupling.

Following are the key aspects of encapsulation:

- **Data Hiding:** Encapsulation enables the hiding of internal data within an object, preventing direct access by external code. The object's internal state is kept private and can only be accessed or modified through designated methods or properties. This protects the integrity of the data and prevents unauthorized modifications.
- **Access Modifiers:** Access modifiers such as public, private, and protected are used to define the level of access to the data and methods within an object. Private members are only accessible within the object itself, while public members can be accessed from outside the object. Protected members are accessible within the object and its subclasses.
- **Getters and Setters:** Encapsulation involves providing controlled access to an object's data through getter and setter methods (also known as accessors and mutators). Getters retrieve the value of a private data member, while setters modify the value. By using these methods, the object can enforce validation, perform calculations, or apply additional logic before accessing or modifying the data.
- **Information Hiding:** Encapsulation allows developers to hide the implementation details of an object's methods and data. The object exposes only the essential information and behavior through a public interface. This hides the complexity and implementation details, providing a simplified view for users of the object.

- **Code Reusability:** Encapsulation promotes code reusability by encapsulating related data and behavior within an object. Objects can be created once and reused in different parts of the program without exposing their internal workings. This reduces code duplication and increases development efficiency.
- **Code Maintenance**: Encapsulation makes code maintenance easier by localizing changes within an object. Modifying the internal implementation of an object does not impact external code as long as the public interface remains unchanged. This improves code modularity, reduces the risk of introducing bugs, and enhances code maintenance and evolution.
- **Security:** Encapsulation enhances security by controlling access to an object's data. By keeping data private and providing controlled access through methods, developers can ensure that only authorized operations are performed on the data. This protects sensitive information and helps maintain data integrity.

Encapsulation plays a crucial role in achieving modular, maintainable, and secure code. It provides a way to protect internal data, control access, and define a clear interface for interacting with objects. By adhering to the principles of encapsulation, developers can design robust and flexible systems that are easier to understand, modify, and extend.

Following are some real world examples to understand abstraction

Example: Bank Account

In a bank account system, the concept of encapsulation can be applied to ensure the privacy and integrity of account data. The account object encapsulates attributes such as the account number, balance, and owner's information. These attributes are kept private and can only be accessed and modified through designated methods such as deposit(), withdraw(), and getBalance(). The internal implementation details, such as the algorithms for interest calculation or transaction logging, are hidden from external access. Encapsulation in this case protects the account's data and provides controlled access to it.

Example: Car

Consider a car object that encapsulates attributes like the make, model, year, and mileage. The car object also has methods such as startEngine(), accelerate(), and brake(). These methods internally manipulate the car's attributes, such as adjusting the speed or updating the mileage. The details of how the engine starts or how the acceleration and braking mechanisms work are hidden from external code. Only the defined methods provide access to manipulate the car's behavior and attributes.

Example: Employee

In an employee management system, encapsulation can be applied to the employee object. The employee object encapsulates attributes such as the employee ID, name, salary, and contact details. These attributes are kept private and can only be accessed or modified through specific methods like setSalary() or getContactDetails(). The internal implementation details, such as how the salary is calculated or how the contact information is stored, are hidden. Encapsulation ensures that the employee's data is protected and accessed only through the defined methods.

In these examples, encapsulation provides several benefits:

- Data Integrity: Encapsulation protects data within objects, preventing unauthorized access or modification.
- Security: By hiding implementation details, encapsulation enhances security by limiting direct access to sensitive data or critical operations.
- Modularity: Encapsulation promotes modularity by encapsulating related attributes and methods within objects, enabling independent development and maintenance.
- Code Flexibility: Encapsulation allows the internal implementation of objects to change without affecting other parts of the code that rely on the object's public interface.
- Code Reusability: Encapsulation facilitates code reuse as objects with well-defined interfaces can be easily incorporated into other systems.

1.4.2 Inheritance

Inheritance is another fundamental concept in object-oriented programming (OOP) that allows one class to inherit properties and behaviors from another class. It is a mechanism that promotes code reuse and the creation of a hierarchical relationship between classes.

Inheritance establishes an "is-a" relationship between classes, where one class (the child or derived class) inherits characteristics from another class (the parent or base class). The child class can access and use the properties and methods of the parent class, and it can also add its own specific properties and methods or override the ones inherited from the parent class.

The class that is being inherited from is called the base class or superclass, while the class that inherits from the base class is called the derived class or subclass.

Inheritance allows the derived class to inherit the following from the base class:

- Attributes (data members): The derived class inherits the attributes defined in the base class. These attributes represent the state or data associated with the objects of the class.
- Methods (member functions): The derived class inherits the methods defined in the base class. These methods define the behaviors or actions that objects of the class can perform.

Inheritance provides several benefits, including:

- Code reuse: Inheritance allows the derived class to reuse the properties and methods of the base class. This reduces code duplication and promotes efficient development by building upon existing code.
- Modularity: Inheritance supports the creation of modular and organized code. Base classes can be designed and implemented independently, focusing on specific behaviors and attributes. Derived classes can then inherit and extend this functionality as needed.

- Polymorphism: Inheritance enables polymorphism, which means that objects of the derived class can be treated as objects of the base class. This allows for writing code that can operate on objects of different classes through a common interface, promoting flexibility and extensibility.
- Hierarchical organization: Inheritance facilitates the organization of classes into a hierarchy. This hierarchy can reflect real-world relationships, making the code more intuitive and easier to understand and maintain.

The derived class can then access the inherited members using appropriate access modifiers and can override or extend the inherited methods as per its requirements.

It's important to note that inheritance should be used judiciously and follow the principle of "favor composition over inheritance" when appropriate. While inheritance offers benefits, it can also lead to tightly coupled code and limitations in terms of multiple inheritance. Therefore, careful design and consideration of the relationships between classes are crucial to effectively utilize inheritance in OOP.

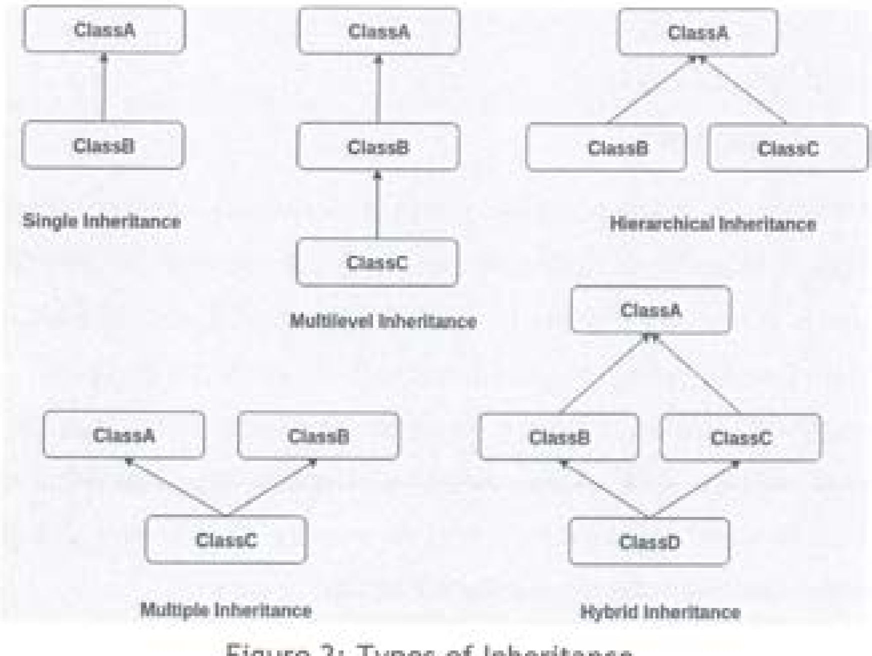


Figure 2: Types of Inheritance

In C++, there are several types of inheritance that allow classes to inherit properties and behaviors from other classes. Here are some common types of inheritance with corresponding C++ examples:

- Single Inheritance: Single inheritance involves a derived class inheriting properties and behaviors from a single base class. It forms a direct hierarchy where a derived class extends the base class.
- Multiple Inheritance: Multiple inheritance allows a derived class to inherit properties and behaviors from multiple base classes. This means that a derived class can have multiple direct base classes, combining their features into a single derived class.
- Multilevel Inheritance: Multilevel inheritance occurs when a derived class inherits from another derived class. It creates a chain or hierarchy of classes, where each derived class further extends the features of its parent class.
- Hierarchical Inheritance: Hierarchical inheritance involves multiple derived classes inheriting from a single base class. It creates a hierarchy of classes, where each derived class has its specific features while sharing the common features of the base class.
- Hybrid (Virtual) Inheritance: Hybrid inheritance combines multiple inheritance with either single or multilevel inheritance. It allows a class to inherit from multiple base classes while avoiding the issues of ambiguity that can arise due to multiple inheritance

1.4.3 Polymorphism

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables objects to exhibit different behaviors based on their specific class types while being accessed through a common interface.

Polymorphism provides a way to write code that can operate on objects of different classes without the need for explicit type checking or casting. This promotes flexibility, modularity, and extensibility in software design.

There are two main types of polymorphism:

- Compile-time Polymorphism (Static Polymorphism): Compile-time polymorphism refers to the ability of a programming language to select different functions or methods at compile-time based on the arguments provided or the types of the objects involved. This is achieved through function overloading and operator overloading.

Function overloading allows multiple functions with the same name but different parameter lists to coexist in the same scope. The appropriate function is selected based on the arguments passed during the function call.

Operator overloading allows operators such as +, -, *, /, etc., to be overloaded for different classes. This enables customized behavior for operators depending on the operands' types.

Example of function overloading in C++:

```
void add(int a, int b) {
   cout << "Sum of two integers: " << (a + b) << endl;
}
void add(double a, double b) {
   cout << "Sum of two doubles: " << (a + b) << endl;
}
int main() {
   add(5, 10);    // Calls the first add() function
   add(3.5, 2.7); // Calls the second add() function
   return 0;
}
```

- **Runtime Polymorphism (Dynamic Polymorphism):** Runtime polymorphism allows objects of different classes to be treated as objects of a common superclass. This is achieved through inheritance and virtual functions.

Inheritance establishes an "is-a" relationship between classes, where a derived class inherits properties and behaviors from a base class. The derived class can be used wherever the base class is expected.

Virtual functions are functions defined in the base class and overridden in the derived class. They allow dynamic dispatch, which means that the appropriate function to call is determined at runtime based on the actual type of the object.

Example of runtime polymorphism in C++:

```
// Base class
class Shape {
public:
   virtual void draw() {
        cout << "Drawing a shape." << endl;
    }
};

// Derived class
class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing a circle." << endl;
    }
};

int main() {
    Shape* shapePtr = new Circle();
    shapePtr->draw();  // Calls the draw() function of Circle
    delete shapePtr;
    return 0;
}
```

Polymorphism allows for code reuse, flexibility, and the ability to design systems that can easily accommodate new types of objects or behaviors. It promotes encapsulation, abstraction, and modular design, making the code more maintainable and extensible.

1.4.2 Abstraction

Abstraction is a key principle in object-oriented programming (OOP) that focuses on simplifying complex systems by hiding unnecessary details and exposing only essential features to the user. It allows us to represent real-world entities and concepts in a more understandable and manageable way.

In OOP, abstraction is achieved by creating abstract classes and interfaces that define the common behavior and characteristics of a group of related objects. These abstract entities provide a blueprint or template for concrete classes to implement. Abstract classes cannot be instantiated; they serve as a foundation for derived classes to inherit from and specialize.

Abstraction involves identifying the essential attributes and behaviors of an object or a system and disregarding irrelevant or low-level details. It allows developers to focus on the high-level concepts and interactions rather than getting bogged down by implementation specifics.

Benefits of abstraction in OOP:

- Simplification: Abstraction simplifies the complexity of a system by breaking it down into manageable parts. It removes unnecessary details and provides a higher-level view that is easier to understand and work with.
- Encapsulation: Abstraction supports encapsulation by hiding the internal implementation details of an object or a system. It allows users to interact with objects through well-defined interfaces without needing to know how the underlying functionality is implemented.
- Reusability: Abstract classes and interfaces promote code reusability. By defining common behaviors and characteristics in abstract entities, they can be inherited by multiple concrete classes, enabling the reuse of code and reducing redundancy.
- Modularity: Abstraction facilitates modular design by creating a clear separation between the interface and implementation. It allows changes to be made to the implementation without affecting the code that uses the abstract entity, promoting maintainability and extensibility.
- Flexibility: Abstraction provides flexibility in the design and evolution of software systems. By defining abstract entities that represent common concepts, it becomes easier to introduce new types of objects or behaviors by simply implementing the required functionality.

1.5 Differences between OOP and Procedure oriented programming

Object-Oriented Programming (OOP) and Procedural Programming are two distinct paradigms with different approaches to designing and structuring software. Following are the key differences between OOP and procedural programming:

Data and Function Relationship:

- Procedural Programming: In procedural programming, data and functions are separate entities. Functions operate on external data, which can be accessed and modified from anywhere in the program.
- Object-Oriented Programming: In OOP, data and functions are bundled together within objects. Objects encapsulate both data (attributes) and behavior (methods/functions) related to the object. The data and functions are tightly bound together, promoting encapsulation and data hiding.

Code Organization:

- Procedural Programming: Procedural programs are organized around procedures or functions that manipulate data. The focus is on a step-by-step sequence of instructions, where functions are called and executed sequentially.
- Object-Oriented Programming: OOP organizes code around objects, which are instances of classes. The emphasis is on the interactions and collaborations between objects. Objects encapsulate data and behavior, promoting modularity and separation of concerns.

Reusability:

- Procedural Programming: Procedural programs achieve reusability through functions, where blocks of code can be reused by calling the same function from different parts of the program.
- Object-Oriented Programming: OOP provides inherent reusability through inheritance and composition. Inheritance allows objects/classes to inherit properties and behaviors from parent classes, while composition enables objects to be composed of other objects. This promotes code reuse and modularity.

Data Access and Control:

- Procedural Programming: In procedural programming, data can be accessed and modified by any function or procedure in the program, leading to a lack of control over data integrity.

- Object-Oriented Programming: OOP promotes data encapsulation and controlled access through methods or functions. Data within objects is protected, and access to it is regulated through getter and setter methods, enhancing data integrity and security.

Code Flexibility and Extensibility:

- Procedural Programming: Procedural programs can become rigid and difficult to modify or extend as they grow larger. Changes made to one function can have ripple effects on other functions and variables.
- Object-Oriented Programming: OOP provides flexibility and extensibility through features like inheritance and polymorphism. Inheritance allows for the creation of new classes by extending existing ones, while polymorphism allows objects of different classes to be treated interchangeably. This promotes code reuse, modularity, and adaptability.

Real-World Modeling:

- Procedural Programming: Procedural programming may not provide a direct and intuitive way to model real-world entities and relationships.
- Object-Oriented Programming: OOP offers a natural way to model real-world objects and their interactions. Objects in OOP closely resemble real-world entities, allowing developers to better represent and solve real-world problems.

OOP focuses on objects, their interactions, and encapsulation of data and behavior, promoting modularity, reusability, and code organization. Procedural programming emphasizes procedures, step-by-step instructions, and separate data and function structures. Both paradigms have their strengths and are suitable for different types of applications and problem domains.

1.6 Summary

This ebook delves into different paradigms for problem solving, the need for object-oriented programming (OOP), and an overview of key OOP principles including abstraction, encapsulation, inheritance, and polymorphism.

The ebook begins by exploring different paradigms for problem solving, highlighting their characteristics and approaches. It emphasizes the shift from procedural programming to the object-oriented paradigm as a way to address the limitations of procedural programming and better model real-world problems.

The need for OOP is then discussed, emphasizing its benefits in software development. OOP's modularity, code reusability, maintainability, flexibility, and scalability are highlighted as crucial factors in tackling complex problems and accommodating changing requirements. The chapter emphasizes the significance of OOP in promoting code organization, collaboration, and better modeling of real-world entities and relationships.

Next, the ebook focuses on the differences between OOP and procedural programming. It highlights the key distinctions such as the emphasis on objects and classes in OOP, encapsulation of data and behavior within objects, and the use of inheritance and polymorphism to promote code reuse and flexibility. The chapter underscores how these differences in approach enable more efficient and maintainable code development. The concept of abstraction is then introduced, emphasizing its role in simplifying complex systems. Abstraction allows developers to focus on essential features while hiding unnecessary details. It provides a high-level view of the problem domain, making the code more intuitive and maintainable.

An overview of key OOP principles follows, starting with encapsulation. Encapsulation emphasizes bundling data and methods within objects, protecting the integrity and consistency of data by restricting direct access. The ebook highlights how encapsulation improves code security, modifiability, and reduces the likelihood of errors. The concept of inheritance is then introduced, illustrating how it enables the derived class to inherit properties and behaviors from a base class. Hierarchical relationships between classes and code reuse are discussed, showcasing how inheritance promotes modularity and extensibility in software systems. Finally, the ebook explores polymorphism, which allows objects of different types to be treated as objects of a common superclass. Polymorphism enables writing generic code that can operate on objects of various classes, fostering flexibility, modifiability, and extensibility.

1.7 Keywords

- Paradigms: Different approaches or models for problem solving.
- Object-Oriented Programming (OOP): A programming paradigm that emphasizes objects, classes, and their interactions to model real-world entities and solve complex problems.
- Procedural Programming: A programming paradigm that focuses on procedures or functions and the step-by-step execution of instructions.
- Differences: Distinctions or variations between different concepts or approaches.
- Abstraction: Simplifying complex systems by focusing on essential features and hiding unnecessary details.
- OOP Principles: Fundamental concepts or guidelines that govern object-oriented programming, including encapsulation, inheritance, and polymorphism.
- Encapsulation: Bundling data and methods within an object and hiding the internal details from the outside world.
- Inheritance: The mechanism by which a class inherits properties and behaviors from a base class, establishing a hierarchy of classes.
- Polymorphism: The ability of objects of different types to be treated as objects of a common superclass, allowing for flexibility and code reuse.
- Code Organization: Structuring and arranging code in a systematic and modular manner for better readability and maintainability.
- Modularity: Breaking down a system into smaller, self-contained modules or components.
- Reusability: The ability to reuse existing code components in different parts of a program or in different programs.
- Maintainability: Ease of making updates, bug fixes, and modifications to the codebase without introducing errors or breaking functionality.
- Flexibility: The ability of a system to adapt to changing requirements or accommodate new functionalities.
- Scalability: The capability of a system to handle increasing amounts of work or a growing user base.

- Collaboration: Working together as a team on a project, often facilitated by the modularity and encapsulation provided by OOP.
- Real-world Modeling: Creating software systems that closely mirror real-world entities, relationships, and behaviors.
- Interfaces: Defined contracts or specifications that describe the methods and behaviors that objects of a class must implement.
- Modifiability: Ease of making modifications to the codebase to add new features or modify existing functionality.
- Extensibility: The ability to extend the functionality of a system without modifying its existing codebase.

1.8 Review Questions

a) How does object-oriented programming (OOP) address the limitations of procedural programming?

b) Why is abstraction important in software development? How does it simplify complex systems?

c) Explain the concept of encapsulation and its benefits in OOP.

d) What is the purpose of inheritance in object-oriented programming? How does it promote code reuse and modularity?

e) Discuss the concept of polymorphism and its role in achieving flexibility and extensibility in OOP.

f) How does OOP promote code reusability and maintainability? Provide examples.

g) Compare and contrast OOP with procedural programming in terms of their approaches and key differences.

h) How does OOP align with real-world modeling? Explain the connection between OOP and modeling entities and relationships.

i) Discuss the need for OOP in software development. What specific challenges does it address?

j) How do the OOP principles (abstraction, encapsulation, inheritance, and polymorphism) contribute to effective and efficient problem-solving?

1.9 References

1. Shiffman, D. (2016). Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (2nd ed.). Morgan Kaufmann.
2. Eckel, B. (2006). Thinking in Java (4th ed.). Prentice Hall.
3. Deitel, P., & Deitel, H. (2017). Java: How to Program (Early Objects) (11th ed.). Pearson.
4. Meyer, B. (1997). Object-Oriented Software Construction (2nd ed.). Prentice Hall.
5. Singh, S., & Singh, S. (2019). Comparative Analysis of Object-Oriented and Procedure-Oriented Programming Paradigms. International Journal of Advanced Research in Computer Science, 10(3), 33-38.
6. Al-Mudimigh, A. S., & Ahmed, M. U. (2012). Encapsulation and Inheritance in Object-Oriented Programming: A Comparative Analysis. International Journal of Computer Science Issues, 9(5), 297-301.
7. Sharma, S., & Guleria, M. (2018). Polymorphism and Inheritance in Object-Oriented Programming: A Comparative Study. International Journal of Computer Applications, 181(23), 37-42.
8. Pandey, R., & Goyal, M. (2017). Understanding the Concept of Abstraction in Object-Oriented Programming: A Comparative Study. International Journal of Computer Applications, 169(1), 20-24.

Unit - 2 C++ BASICS

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Understand the structure of a C++ program: Learn about the essential components of a C++ program, including the main function, headers, and namespaces. Gain the ability to organize code effectively and create well-structured programs.
- Familiarize yourself with data types: Explore different data types available in C++, such as integers, floating-point numbers, characters, and booleans. Learn how to declare variables of different data types and effectively store and manipulate data.
- Master variable declaration: Gain proficiency in declaring variables in C++ and assigning values to them. Understand the rules and best practices for naming variables and managing memory efficiently.
- Comprehend expressions and operators: Learn about various operators in C++, including arithmetic, relational, logical, and assignment operators. Understand how to combine variables and values using expressions to perform calculations and make decisions.
- Understand operator precedence: Gain knowledge of operator precedence in C++ and learn how it determines the order of evaluation in complex expressions. Avoid common pitfalls and write code that produces the desired results by understanding how operators interact with each other.

Introduction

In this Ebook, we will explore the foundational concepts of C++ programming in a way that is easy to understand. We will cover topics such as the structure of a C++ program, data types, variable declaration, expressions, operators, and operator precedence, breaking them down into simple and digestible explanations. C++ is like a language of its own, with its own set of rules and vocabulary. We will guide you through these concepts, using plain language and relatable examples to help you grasp the core ideas. Whether you are new to programming or have some experience, our aim is to make these topics accessible and approachable for everyone.

We will start by examining the structure of a C++ program, explaining how it is organized and how different parts fit together. Think of it as understanding the structure of a story or a recipe. By the end of this section, you will have a clear understanding of how to structure your own programs effectively.

Data types can sometimes feel overwhelming, but fear not! We will introduce each data type step by step, describing its purpose and giving real-world examples to make it relatable. You will gain an understanding of how to use different data types to store and manipulate different kinds of information in your programs. Variable declaration might sound complex, but we will break it down into simple terms. We will show you how to create variables, assign values to them, and explain the rules for naming them. Through practical examples, you will learn how to declare variables correctly and use them to store and retrieve data. Expressions and operators are the tools that allow us to perform calculations and make decisions in C++. We will guide you through these concepts by using familiar scenarios and everyday situations. You will learn how to combine values and variables using operators, and how to write expressions that produce the desired results. To navigate the world of operators, we will explain their purpose and show you how to use them effectively. Don't worry if the different types of operators seem confusing at first - we will simplify them and provide clear examples so that you can understand their functions and apply them confidently in your code. Lastly, we will demystify operator precedence by explaining the order in which operators are evaluated in expressions. We will illustrate this concept with easy-to-understand examples and visuals, ensuring that you have a solid grasp of how to correctly evaluate expressions in your programs.

2.1 Structure of C++ Program

The structure of a C++ program consists of various components that work together to create a functional and organized codebase. Understanding the structure is essential for writing clear and efficient programs. Let's explore the key components of a typical C++ program:

a) **Preprocessor Directives:** A C++ program often begins with preprocessor directives, which provide instructions to the compiler before the actual compilation process. These directives start with a '#' symbol and are used to include header files or define constants that will be used in the program.

b) **Header Files:** Following the preprocessor directives, we include necessary header files using the "#include" directive. Header files contain definitions and declarations for libraries and functions that we want to use in our program. Commonly used header files include <iostream> for input/output operations and <cmath> for mathematical functions.

c) **Namespace Declaration:** The "namespace" keyword is used to declare the namespace(s) that we want to use in our program. Namespaces help avoid naming conflicts and provide a way to organize code. The most commonly used namespace in C++ is the "std" namespace, which includes standard library functions and objects.

d) **Main Function:** Every C++ program must have a "main" function, which serves as the entry point of the program. It has a specific format: "int main()". The main function is where the program execution starts and ends. It can also accept command-line arguments, which allow interaction with the program from the terminal.

e) **Function Definitions:** After the main function, you can define your own functions, if necessary. Functions encapsulate a specific set of instructions that can be called and reused throughout the program. They help modularize the code and make it more readable and maintainable.

f) **Variable Declarations:** In C++, variables must be declared before they can be used. Variable declarations specify the name and type of the variable. This step allocates memory to store the variable and prepares it for use in the program. Variable declarations can be done within functions or at the global level.

g) **Statements and Expressions:** Inside functions, you can write statements and expressions to perform operations and computations. Statements are individual instructions that perform specific actions, while expressions evaluate to a value. These can include assignments, calculations, conditional statements, loops, and more.

h) **Return Statement:** The "return" statement is used to exit the function and return a value (if the function has a return type other than void) back to the caller. It also serves as the end of the main function, and a return value of 0 conventionally indicates successful program execution.

Let's dive into the structure of a C++ program in more detail, using a simple example to illustrate each component. Here's a breakdown of the structure with explanations:

Lets consider this program

```
#include <iostream>

void greet();

int main() {
   int age = 25;  // Variable declaration and initialization
   float pi = 3.14159;
   std::string name = "John";
   greet();
   std::cout << "My name is " << name << "." << std::endl;
   std::cout << "I am " << age << " years old." << std::endl;
   std::cout << "The value of pi is approximately " << pi << std::endl;

   int sum = age + 5;  // Expression
   std::cout << "After adding 5, my age becomes " << sum << std::endl;

   return 0;
}

void greet() {
   std::cout << "Hello, World!" << std::endl;
}
```

Preprocessor Directive:

```
#include <iostream>
```

This line is a preprocessor directive that tells the compiler to include the iostream header file. The iostream library provides input/output stream functionality in C++. It allows us to use objects like cout and endl for console output.

Function Declaration:

```
void greet();
```

This line declares a function named greet() without providing the implementation. Function declaration informs the compiler that there is a function with a specific name, return type, and parameters. It allows functions to be called before their actual definition.

Main Function:

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

The main() function is the entry point of every C++ program. It is where program execution begins. The int before main() indicates that the function returns an integer value. In this example, we call the greet() function, and then the function returns 0 to indicate successful program execution.

Variable Declaration and Initialization:

```
int age = 25;
float pi = 3.14159;
std::string name = "John";
```

In the main() function, we declare and initialize variables. The int type is used for whole numbers, float for floating-point numbers, and std::string for strings. Here, we declare and initialize variables age, pi, and name with their respective values.

Function Definition:

```
void greet() {
   std::cout << "Hello, World!" << std::endl;
}
```

This code defines the greet() function, which was declared earlier. The function name, return type, and parameters must match the declaration. In this case, void indicates that the function does not return a value. Within the function, we use std::cout to output the string "Hello, World!" to the console.

Output Statements with Variables:

```
std::cout << "My name is " << name << "." << std::endl;

std::cout << "I am " << age << " years old." << std::endl;

std::cout << "The value of pi is approximately " << pi << std::endl;
```

These lines use the std::cout object to print output to the console. The << operator is used to concatenate variables and strings within the output statements. We display the values of name, age, and pi in the console.

```
int sum = age + 5;
```

This line demonstrates an expression where we add 5 to the age variable and assign the result to the sum variable. The + operator performs the addition, and the result is stored in sum.

2.2 Data Types

Data types define the type of data that a variable can hold. They specify the size and format of the data, as well as the range of values and the operations that can be performed on that data. Different data types are used to store different kinds of information, such as numbers, characters, or boolean values.

In C++, data types are classified into several categories, including integer types, floating-point types, character types, boolean type, and more. Each data type has its own set of properties and limitations. By choosing the appropriate data type, you can ensure efficient memory usage and perform operations accurately on the data stored in variables. Using the correct data type is important because it determines the amount of memory allocated to store the data and the range of values that can be represented. It also affects the performance and accuracy of operations performed on the data.

Following are various data types in C++:

1. **Integer Types:** int: Represents whole numbers, both positive and negative. It typically uses 4 bytes of memory.

   Example: int age = 25;

2. **short:** Represents smaller whole numbers. It typically uses 2 bytes of memory.

   Example: short quantity = 10;

3. **long:** Represents larger whole numbers. It typically uses 8 bytes of memory.

   Example: long population = 1000000;

4. **Floating-Point Types**: float: Represents decimal numbers with single precision. It typically uses 4 bytes of memory.

   Example: float pi = 3.14159;

5. **double:** Represents decimal numbers with double precision. It typically uses 8 bytes of memory.

   Example: double weight = 68.5;

6. **Character Types:** char: Represents individual characters. It typically uses 1 byte of memory.

   Example: char grade = 'A';

7. **wchar_t**: Represents wide characters. It typically uses 2 bytes of memory or more.

   Example: wchar_t unicodeChar = L'\u0394';

8. **Boolean Type**: bool: Represents boolean values, either true or false.

   Example: bool isPassed = true;

9. Enumeration Types: enum: Allows you to define a set of named constants.

   Example:

   enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

   Days today = Monday;

10. **Derived Types:** array: Represents a fixed-size collection of elements of the same type.

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

11. **pointer:** Stores the memory address of another variable.

    Example: int* p = &age;

12. **reference:** Provides an alias to an existing variable.

```
Example: int& r = age;
```

13. **structure:** Allows you to create your own composite data type.

```
Example:
struct Person {
   std::string name;
   int age;
};
Person person1 = { "John", 25 };
```

14. **class:** Similar to a structure but with additional capabilities like encapsulation and inheritance.

```
class Rectangle {
   int width;
   int height;
public:
   int area() { return width * height; }
};
Rectangle rect;
rect.width = 5;
rect.height = 10;
int area = rect.area();
```

2.3 Declaration of Variables

Declaration of variables in C++ involves specifying the type and name of a variable, allowing the compiler to allocate memory for it. Here's a detailed explanation of variable declaration in C++, along with rules and examples:

Syntax:

The general syntax for declaring a variable in C++ is:

```
data_type variable_name;
```

Rules for Variable Declaration:

- Variable names must start with a letter or underscore and can be followed by letters, digits, or underscores.
- Variable names are case-sensitive. For example, count and Count are considered different variables.
- C++ has reserved keywords that cannot be used as variable names.
- Variables should be declared before they are used in the program.

Examples:

- Integer Variable:
  int age;
  This declares a variable named age of type int, which can hold whole numbers.
- Floating-Point Variable:
  float pi;
  This declares a variable named pi of type float, which can hold decimal numbers.
- Character Variable:
  char grade;
  This declares a variable named grade of type char, which can hold single characters.
- Boolean Variable:
  bool isStudent;
  This declares a variable named isStudent of type bool, which can hold true or false values.

Variable Initialization:

Variables can be initialized at the time of declaration, assigning an initial value to them. Initialization is optional, but it's a good practice to give variables an initial value to avoid using uninitialized variables.

Example:

```
int count = 0;

float temperature = 98.6;

char symbol = 'A';
```

```
bool isValid = true;
```

Multiple Variable Declaration:

Multiple variables of the same type can be declared in a single statement, separated by commas.

Example:

```
int x, y, z;
```

Constant Variables:

Variables can be declared as constants using the const keyword. Constant variables cannot be modified after initialization.

Example:

```
const int MAX_VALUE = 100;
```

2.4 Expressions

Expressions in C++ are combinations of literals, variables, operators, and function calls that produce a value. They represent computations or operations to be performed. Expressions can be as simple as a single variable or value, or they can be complex, involving multiple operands and operators. Here's a detailed explanation of expressions in C++:

Components of Expressions:

- Literals: Fixed values such as numbers or characters. For example, 5, 'A', or "Hello".
- Variables: Named storage locations that hold values. For example, x, age, or result.
- Operators: Symbols that perform operations on operands. Examples include arithmetic operators (+, -, *, /), comparison operators (<, >, ==), logical operators (&&, ||), and assignment operators (=, +=, -=).

Function calls: Invocations of functions that return values. For example, sqrt(16), strlen("Hello").

Examples of Expressions:

```
int x = 5;
int y = 3;
int z = x + y;  // Addition expression
bool isPositive = (z > 0);  // Comparison expression
float average = (x + y) / 2.0;  // Arithmetic expression
int result = factorial(4);  // Function call expression
```

In the above examples:

The expression x + y adds the values of x and y together and assigns the result to z.

The expression z > 0 compares z with 0 and assigns the result to isPositive.

The expression (x + y) / 2.0 calculates the average of x and y and assigns the result to average.

The expression factorial(4) calls a function named factorial with the argument 4 and assigns the returned value to result.

Evaluation of Expressions:

- Expressions are evaluated by the compiler at runtime. The evaluation involves applying the operator's rules and precedence to the operands. Parentheses can be used to control the order of evaluation.

Side Effects of Expressions:

- Expressions can have side effects, such as modifying variables, changing the program's state, or invoking functions that have side effects. For example, an assignment expression (x = y) modifies the value of x, and a function call expression can perform actions beyond just returning a value.

Type Conversion in Expressions: C++ performs automatic type conversions or promotions to ensure that expressions with mixed data types can be evaluated correctly. This includes converting values of one type to another if necessary.

2.5 Operators

Operators in C++ are symbols or keywords that perform various operations on operands, such as variables, literals, or expressions. They allow you to manipulate and process data in different ways. C++ provides a wide range of operators, including arithmetic, assignment, comparison, logical, bitwise, and more.

1. **Arithmetic Operators:**
   - Addition +: Adds two operands together.
   - Subtraction -: Subtracts the second operand from the first.
   - Multiplication *: Multiplies two operands.
   - Division /: Divides the first operand by the second.
   - Modulus %: Returns the remainder of the division.
   - Increment ++: Increases the value of the operand by 1.
   - Decrement --: Decreases the value of the operand by 1.

Example:

```
int x = 5;
int y = 3;
int sum = x + y;        // Addition
int difference = x - y; // Subtraction
int product = x * y;  // Multiplication
int quotient = x / y;  // Division
int remainder = x % y;  // Modulus
x++;               // Increment
y--;               // Decrement
```

2. **Assignment Operators:**
   - Assignment =: Assigns the value on the right to the variable on the left.
   - Compound assignment operators (+=, -=, *=, /=, %=): Combine an arithmetic operation with assignment.

Example:

```
int x = 5;
x += 3;  // Equivalent to x = x + 3
```

3. **Comparison Operators:**
   - Equal to ==: Checks if two operands are equal.
   - Not equal to !=: Checks if two operands are not equal.
   - Greater than >: Checks if the first operand is greater than the second.
   - Less than <: Checks if the first operand is less than the second.

- Greater than or equal to >=: Checks if the first operand is greater than or equal to the second.
- Less than or equal to <=: Checks if the first operand is less than or equal to the second.

Example:

```
int x = 5;
int y = 3;
bool isEqual = (x == y);  // Equality comparison
bool isGreater = (x > y); // Greater than comparison
```

4. Logical Operators:

- Logical AND &&: Returns true if both operands are true.
- Logical OR ||: Returns true if at least one of the operands is true.
- Logical NOT !: Reverses the logical state of the operand.

Example:

```
bool isSunny = true;
bool isWarm = false;
bool isGoodWeather = isSunny && isWarm;  // Logical AND
bool isEitherSunnyOrWarm = isSunny || isWarm; // Logical OR
bool isNotSunny = !isSunny;  // Logical NOT
```

5. **Bitwise Operators:**

- Bitwise AND &: Performs bitwise AND operation on the operands.
- Bitwise OR |: Performs bitwise OR operation on the operands.
- Bitwise XOR ^: Performs bitwise XOR operation on the operands.
- Bitwise NOT ~: Performs bitwise NOT operation on the operand.
- Left shift <<: Shifts the bits of the first operand to the left by the number of positions specified by the second operand.
- Right shift >>: Shifts the bits of the first operand to the right by the number of positions specified by the second operand.

Example:

```
int x = 5;  // Binary: 0101
int y = 3;  // Binary: 0011
int bitwiseAnd = x & y;  // Bitwise AND: 0001 (1 in decimal)
int bitwiseOr = x | y;   // Bitwise OR: 0111 (7 in decimal)
int bitwiseXor = x ^ y;  // Bitwise XOR: 0110 (6 in decimal)
int bitwiseNot = ~x;    // Bitwise NOT: 1010 (-6 in decimal)
int leftShift = x << 2;  // Left shift: 10100 (20 in decimal)
int rightShift = x >> 1; // Right shift: 0010 (2 in decimal)
```

2.6 Operator Precedence

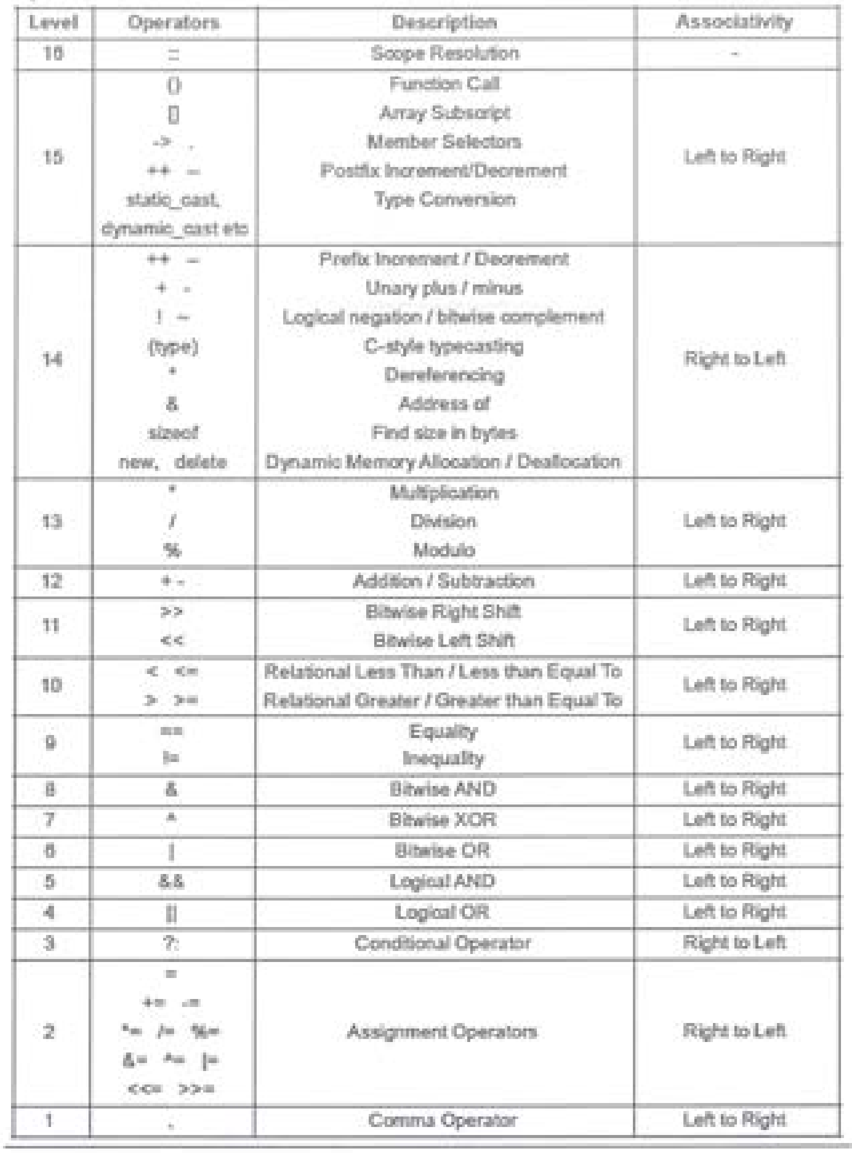
Operator precedence in C++ determines the order in which operators are evaluated when an expression contains multiple operators. It defines the grouping and sequencing of operators based on their priority levels. Operator precedence ensures that expressions are evaluated correctly, following the rules specified by the language.

Here's an explanation of operator precedence in C++:

1. **Operator Precedence Levels**: C++ assigns each operator a precedence level, which determines its priority in an expression. Operators with higher precedence are evaluated first, followed by operators with lower precedence. In case of operators having the same precedence level, the associativity of the operators comes into play.
2. **Operator Precedence Rules:**
   - Higher precedence operators are evaluated before lower precedence operators.
   - Operators with the same precedence level are evaluated from left to right, unless the operator is right-associative.
   - Parentheses ( ) can be used to override the default precedence and enforce a specific evaluation order.
3. **Common Operator Precedence Examples:**
   - Arithmetic Operators:
     - Multiplication *, Division /, and Modulus % have higher precedence than Addition + and Subtraction -.
     - For example: 2 + 3 * 4 is evaluated as 2 + (3 * 4), resulting in 14.
   - Comparison Operators:
     - Comparison operators such as <, >, <=, and >= have higher precedence than Equality operators == and !=.

For example: 2 + 3 < 4 is evaluated as (2 + 3) < 4, resulting in true.

- Logical Operators:
  - Logical NOT ! has higher precedence than Logical AND && and Logical OR ||.
  - For example: !true && false is evaluated as (!true) && false, resulting in false.
- Assignment Operators:
  - Assignment operators such as =, +=, -= have lower precedence than most other operators.
  - For example: x = 2 + 3 is evaluated as x = (2 + 3), assigning 5 to x.
- Operator Precedence and Parentheses:
  - Parentheses ( ) can be used to override the default precedence and enforce a specific evaluation order. Expressions within parentheses are evaluated first.
  - For example: (2 + 3) * 4 is evaluated as 5 * 4, resulting in 20.



Level	Operators	Description	Associativity
16	::	Scope Resolution	-
15	() [] -> . ++ -- static_cast, dynamic_cast etc	Function Call Array Subscript Member Selectors Postfix Increment/Decrement Type Conversion	Left to Right
14	++ -- + - ! ~ (type) * & sizeof new, delete	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement C-style typecasting Dereferencing Address of Find size in bytes Dynamic Memory Allocation / Deallocation	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== !=	Equality Inequality	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

2.7 Summary

The ebook begins by introducing the structure of a C++ program, explaining the necessary components and their respective roles. Readers gain a clear understanding of how to organize their code effectively and create well-structured programs.

Moving on, the book delves into data types, discussing the various options available in C++. Readers learn about fundamental types, including integers, floating-point numbers, characters, and booleans. Additionally, more advanced data types, such as arrays and structures, are explained, allowing readers to handle complex data structures efficiently.

The ebook then explores the crucial concept of variable declaration in C++. It covers the syntax for declaring variables and discusses the rules and best practices for naming variables. With this knowledge, readers can confidently create variables of different types and utilize them effectively within their programs. Next, the ebook focuses on expressions in C++. It explains how expressions are formed using variables, literals, and operators. Additionally, readers gain an understanding of the various operators available in C++ and their respective functionalities. The guide goes into detail about arithmetic, relational, logical, and assignment operators, providing examples and practical exercises to reinforce learning.Lastly, the ebook covers operator precedence, a fundamental aspect of C++ programming. It explains the rules that govern the order of evaluation when multiple operators are present in an expression. By understanding operator precedence, readers can write code that performs calculations accurately and avoids ambiguity.

2.8 Keywords

- C++ Basics: An ebook providing a comprehensive guide to fundamental concepts in C++ programming.
- Structure of a C++ program: Explains the essential components and organization of a C++ program.
- Data types: Covers different types of data in C++ such as integers, floating-point numbers, characters, and booleans.
- Declaration of variables: Discusses the syntax and best practices for declaring variables in C++.
- Expressions: Explores how expressions are formed using variables, literals, and operators in C++.
- Operators: Covers the various operators available in C++ such as arithmetic, relational, logical, and assignment operators.
- Operator precedence: Explains the rules governing the order of evaluation in expressions with multiple operators.

2.9 Review Questions

1. What is the structure of a basic C++ program? Describe the different components that make up a C++ program.
2. How are variables declared in C++? Explain the syntax and rules for declaring variables of different data types.
3. What are the fundamental data types in C++? Give examples of each data type and describe their characteristics.
4. How do you assign values to variables in C++? Provide examples of assignment statements for different data types.
5. Explain the concept of expressions in C++. What are the different types of expressions, and how are they evaluated?
6. What are operators in C++? Discuss the various types of operators and provide examples of their usage.
7. Describe the concept of operator precedence in C++. How does it determine the order in which operators are evaluated in an expression?

8. How does C++ handle arithmetic operations on variables of different data types? Explain the rules and potential implications of mixed-type arithmetic.
9. Discuss the concept of typecasting in C++. When and how would you perform explicit type conversions?

2.10 References

1. Shiffman, D. (2016). Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (2nd ed.). Morgan Kaufmann.
2. Eckel, B. (2006). Thinking in Java (4th ed.). Prentice Hall.
3. Deitel, P., & Deitel, H. (2017). Java: How to Program (Early Objects) (11th ed.). Pearson.
4. Meyer, B. (1997). Object-Oriented Software Construction (2nd ed.). Prentice Hall.

Unit - 3 Evaluation of expressions and Flow control statements

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Evaluate expressions in C++, understanding the order of operations and the use of various operators.
- Perform type conversions in C++, including implicit and explicit conversions, and comprehend their impact on program execution.
- Manipulate pointers in C++, including understanding their relationship with variables, accessing memory addresses, and modifying data directly.
- Work with arrays in C++, covering their declaration, initialization, and manipulation, including multidimensional arrays.
- Understand the intersection of pointers and arrays, utilizing pointers to efficiently access and modify array elements.
- Understand the purpose and functionality of flow control statements such as if, switch, while, for, do, break, continue, and goto.
- Learn how to use if statements to make decisions and control the flow of your code based on specific conditions.
- Master the switch statement and its ability to simplify complex branching logic by evaluating multiple cases.
- Gain proficiency in creating loops using the while statement, enabling repetitive execution until a specified condition is met.
- Discover the power of the for statement for iterating over a range of values, controlling loop progression, and streamlining code.
- Learn how to use the do statement to ensure the execution of a block of code at least once, even if the condition is initially false.

Introduction

C++ is a powerful programming language widely used in various domains, including software development, game development, and embedded systems. To become proficient in C++, it is essential to grasp the core concepts that form the building blocks of the language. This eBook focuses on key topics that are vital for any C++ programmer's skill set.

We begin by exploring the evaluation of expressions, an integral aspect of C++ programming. Understanding how expressions are computed, the order of operations, and the use of different operators will enhance your ability to write concise and effective code. Next, we delve into type conversions, an important aspect of any programming language. You will learn about implicit and explicit conversions, how they affect program execution, and techniques to handle conversions appropriately. The concept of pointers is next on our journey. Pointers allow you to work with memory addresses and manipulate data directly, offering unparalleled flexibility and control. We will guide you through the fundamentals of pointers, their relationship with variables, and their application in various programming scenarios. Arrays are another fundamental data structure in C++. We will explore their declaration, initialization, and manipulation techniques. Additionally, we will cover multidimensional arrays and efficient array manipulation using pointers. To handle textual data, we dedicate a section to strings. You will learn how to declare, initialize, and manipulate strings, as well as perform common operations like concatenation and searching. Structures provide a means to organize and store related data. We will guide you through the creation and usage of structures, including nested structures and structure arrays. We will also explore passing structures to functions, returning structures from functions, and utilizing structure pointers. Lastly, we introduce references, a powerful feature in C++ that provides an alternative to pointers for efficient variable manipulation. You will gain an understanding of references, their declaration, initialization, and usage, and discover how they offer advantages over pointers in certain scenarios.

3.1 Evaluation of expressions

Evaluation of expressions is a fundamental aspect of C++ programming. Expressions are combinations of values, variables, operators, and function calls that result in a single value. Understanding how expressions are evaluated and the order in which operations are performed is crucial for writing correct and efficient code.

Arithmetic Expressions:

Arithmetic expressions involve mathematical operations such as addition, subtraction, multiplication, and division. The evaluation follows the standard mathematical rules, considering operator precedence and associativity.

```
int a = 10;
int b = 5;
int c = 3;
int result = (a + b) * c - a / b;
```

In the above example, the expression result = (a + b) * c - a / b is evaluated as follows:

- First, the addition (a + b) is performed, resulting in 15.
- Next, the multiplication (15 * c) is evaluated, yielding 45.
- Then, the division (a / b) is calculated, resulting in 2 (integer division).
- Finally, the subtraction (45 - 2) is performed, giving the final result of 43, which is assigned to the variable result.

Relational Expressions:

Relational expressions compare values and return a Boolean result (true or false). Common relational operators include <, >, <=, >=, ==, and !=.

```
int x = 5;
int y = 10;
bool isGreater = x > y;
bool isEqual = x == y;
```

In the above example, the expression x > y evaluates to false since 5 is not greater than 10. Similarly, the expression x == y evaluates to false because x and y are not equal.

Logical Expressions:

Logical expressions combine Boolean values using logical operators such as && (logical AND), || (logical OR), and ! (logical NOT).

```
bool a = true;
bool b = false;
bool result = a && !b;
```

In the above example, the expression a && !b evaluates to true. It first applies the logical NOT operator to b, resulting in true (since !false is true). Then, it performs the logical AND operation between a and the negated b, resulting in true.

Assignment Expressions:

Assignment expressions are used to assign a value to a variable using the = operator.

```
int x = 5;
int y = 10;
y = x + 3;
```

In the above example, the expression y = x + 3 assigns the value of x + 3 (which is 8) to the variable y.

3.2 Type conversions

Type conversions, also known as type casting, in C++ allow you to convert values from one data type to another. Type conversions are essential for ensuring compatibility and performing operations between different data types. Let's explore the different types of type conversions in C++:

1. Implicit Conversions:

   Implicit conversions, also known as automatic conversions, occur automatically by the compiler when it detects a compatible type conversion. These conversions are safe and don't require any explicit operator or function calls.

   ```
   int num = 10;
   double result = num;  // Implicit conversion from int to double
   ```

   In the above example, the num variable of type int is implicitly converted to type double during the assignment. The compiler automatically performs the conversion, widening the value from an integer to a floating-point number.

2. Explicit Conversions:

Explicit conversions, also known as type casting, are performed explicitly by the programmer using casting operators. These conversions require explicit syntax to indicate the desired conversion.

1. Static Cast: The static cast operator static_cast is used for most common type conversions. It performs conversions that are known to be safe, such as converting between numeric types or derived-to-base class conversions.

   ```
   double num = 3.14;
   int result = static_cast<int>(num);  // Explicit conversion from double to int
   ```

   In the above example, the static cast is used to explicitly convert the num variable of type double to an int. This conversion truncates the decimal part, resulting in the value 3 being assigned to the result variable.

2. Dynamic Cast: The dynamic cast operator dynamic_cast is used for conversions between pointers or references to polymorphic types, such as derived-to-base class conversions. It performs runtime type checking to ensure the conversion is valid.

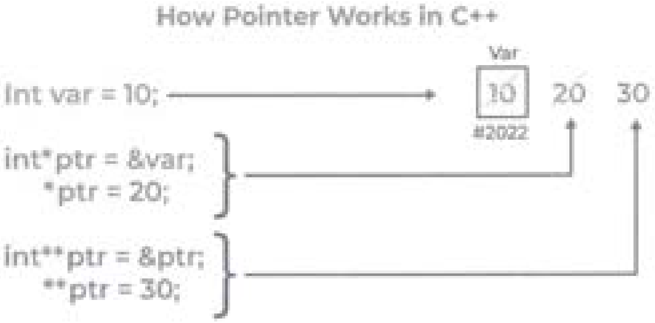
   ```
   class Base {
    // Base class definition
   };

   class Derived : public Base {
    // Derived class definition
   };

   Base* basePtr = new Derived();
   Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);  // Explicit downcast
   ```

   In the above example, the dynamic cast is used to explicitly downcast the basePtr of type Base* to a Derived*. It ensures that the conversion is valid and returns a pointer to the derived class object. If the conversion is not possible, it returns a null pointer.

3. Other Casts: C++ also provides additional cast operators for specific scenarios, such as const_cast for removing constness, reinterpret_cast for low-level reinterpretation of types, and typeid for obtaining type information at runtime.

   It's important to exercise caution when using explicit conversions, as they may lead to unintended behavior or loss of data. Only perform explicit conversions when you are certain about the safety and compatibility of the conversion. Understanding and utilizing type conversions in C++ allows you to manipulate data of different types and perform operations effectively. Be mindful of implicit and explicit conversions to ensure correct and reliable program behavior.

3.3 Pointers



C++ pointers are variables that store memory addresses. They allow you to directly access and manipulate data stored in memory. Pointers are a powerful feature of C++ and are commonly used for dynamic memory allocation, working with arrays, and interacting with functions. Let's delve into the details of C++ pointers:

1. **Declaring and Initializing Pointers:**

To declare a pointer, you use the * symbol in front of the variable name to indicate that it will store a memory address. Here's an example:

```
int* ptr; // Declaration of an integer pointer
```

To initialize a pointer, you assign it the address of a variable or allocate memory using the new operator:

```
int num = 10;
```

```
int* ptr = &num;  // Assigning the address of 'num' to 'ptr'
int* dynPtr = new int;  // Dynamically allocating memory
```

2. **Accessing and Modifying Pointed Values:** You can access the value stored at a memory address pointed to by a pointer using the dereference operator (*). This retrieves the value rather than the address itself:

```
int num = 10;
int* ptr = &num;

cout << *ptr << endl;  // Output: 10

*ptr = 20; // Modifying the value through the pointer
cout << num << endl;  // Output: 20
```

In the above example, *ptr retrieves the value stored at the memory address pointed to by ptr. Modifying the value through the pointer (*ptr = 20) also updates the original variable num.

3. **Null Pointers:** A null pointer does not point to a valid memory address. It is typically used to indicate the absence of a valid target. You can assign a null value to a pointer using the nullptr keyword or the literal 0:

```
int* nullPtr = nullptr;
int* anotherNullPtr = 0;
```

Null pointers are often used for error checking or as initial values before pointing to valid memory locations.

4. **Pointer Arithmetic:** Pointers can be incremented or decremented to navigate through memory addresses. This is particularly useful when working with arrays or accessing consecutive memory locations:

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

cout << *ptr << endl;  // Output: 1

ptr++;  // Move the pointer to the next element
```

```
cout << *ptr << endl; // Output: 2
```

5. **Memory Deallocation:** If you allocate memory dynamically using the new operator, it is essential to deallocate that memory to prevent memory leaks. Deallocate memory using the delete operator:

```
int* dynPtr = new int;
// Use the dynamically allocated memory
delete dynPtr; // Deallocate the memory
```

For arrays allocated with new[], use delete[] for deallocation:

```
int* dynArr = new int[5];
// Use the dynamically allocated array

delete[] dynArr; // Deallocate the array
```

Forgetting to deallocate dynamically allocated memory can lead to memory leaks, where memory remains allocated even after it is no longer needed.

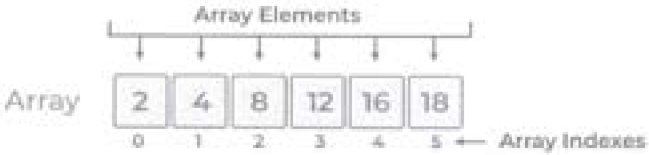
C++ pointers offer flexibility and control over memory management. However, their misuse can lead to bugs, crashes, and security vulnerabilities. Ensure proper initialization, dereference carefully, handle null pointers, and deallocate dynamically allocated memory to maintain safe and efficient code.

Points to remember

- A pointer is a variable that stores the memory address of another variable.
- It allows indirect access to the memory location where a value is stored.
- Pointers enable dynamic memory allocation and deallocation.
- They are denoted by an asterisk (*) before the variable name.
- Pointers can be used to improve program efficiency by avoiding unnecessary data copying

3.4 Arrays

Array in C



Arrays in C++ are a collection of elements of the same data type, stored in contiguous memory locations. They provide a convenient way to store and access multiple values of the same type. Here are some key points about arrays in C++:

1. **Declaring and Initializing Arrays:** To declare an array, you specify the data type of its elements, followed by the array name and the size of the array in square brackets:

```
int numbers[5]; // Declaration of an integer array with size 5
```

You can also initialize an array at the time of declaration:

```
int numbers[] = {1, 2, 3, 4, 5}; // Array initialization with values
```

2. **Accessing Array Elements:** Array elements are accessed using their index, starting from 0. You can use the square bracket notation to access or modify individual elements of the array:

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

int firstElement = numbers[0]; // Accessing the first element

numbers[2] = 10; // Modifying the third element
```

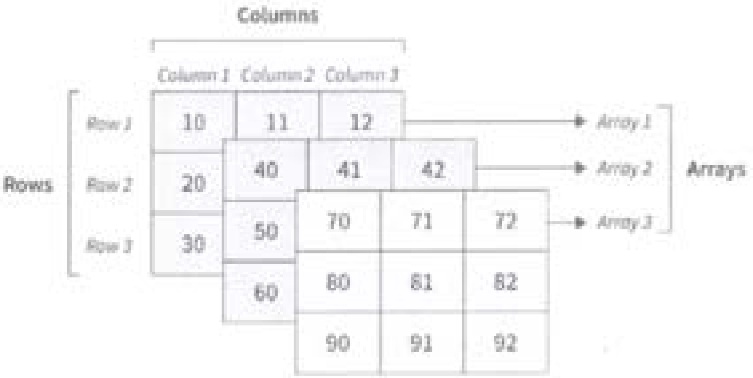
In the above example, numbers[0] retrieves the first element of the array, and numbers[2] = 10 modifies the third element.

3. **Array Size and Bounds:** The size of an array is fixed at the time of declaration and cannot be changed during runtime. It's important to ensure that array indices are within bounds to avoid accessing memory outside the array boundaries, which can result in undefined behavior or program crashes.

4. **Multidimensional Arrays:** C++ supports multidimensional arrays, such as 2D arrays, which are essentially arrays of arrays. They are useful for representing matrices, grids, or other tabular data:

```
int matrix[3][3] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
cout << matrix[1][2] << endl; // Output: 6
```

In the above example, matrix[1][2] accesses the element at the second row and third column of the 2D array.



Points to remember

- An array is a collection of elements of the same data type, stored in contiguous memory locations.
- It provides a convenient way to store and access multiple values of the same type.
- Array elements are accessed using indices, starting from 0.
- Arrays have a fixed size, which is determined at the time of declaration.

- They allow for efficient memory utilization and easy traversal of elements.

3.5 Pointers and Arrays

In C++, the array name can be treated as a pointer to its first element. Therefore, you can use a pointer to traverse an array or pass it as an argument to functions:

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

int* ptr = numbers;  // 'numbers' is equivalent to '&numbers[0]'

cout << *ptr << endl;  // Output: 1

cout << *(ptr + 2) << endl;  // Output: 3
```

Here's a detailed explanation of the relationship between arrays and pointers in C++:

1. Array Name as a Pointer: In C++, the name of an array can be treated as a pointer to its first element. When you use the array name in an expression, it is automatically converted to a pointer to the first element of the array. For example:

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

   int* ptr = numbers;
   ```

   In the above code, the array name numbers is automatically converted to a pointer to its first element. Thus, assigning numbers to the pointer ptr is valid.

2. Accessing Array Elements: To access individual elements of an array, you can use the array subscript notation or pointer arithmetic. Both methods are equivalent. For example:

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

   // Using array subscript notation
   ```

```
int firstElement = numbers[0]; // Accessing the first element

// Using pointer arithmetic

int* ptr = numbers;

int secondElement = *(ptr + 1);  // Accessing the second element
```

using either numbers[index] or *(ptr + index). The index specifies the position of the element within the array.

3. Pointer Arithmetic and Array Traversal: Pointer arithmetic allows you to traverse an array by incrementing or decrementing a pointer. When you increment a pointer, it moves to the next memory location based on the size of the data type it points to. Similarly, decrementing the pointer moves it to the previous memory location. For example:

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

int* ptr = numbers;

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

 cout << *ptr << " ";  // Accessing array elements using pointer dereference

 ptr++;  // Moving to the next element

}
```

In the above code, the ptr pointer is initially pointing to the first element of the numbers array. By incrementing ptr inside the loop, we can traverse the array and print its elements.

4. Pointers and Function Arguments: Pointers are commonly used in function parameters to pass arrays as arguments. When an array is passed to a function, it decays into a pointer to its first element. The function can then use pointer notation to access the elements of the array. For example:

```
void printArray(int* arr, int size) {
  for (int i = 0; i < size; i++) {
    cout << arr[i] << " ";
  }
}
```

```
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);
printArray(numbers, size);
```

In this example, the printArray function takes a pointer arr and the size of the array as arguments. Inside the function, the array elements are accessed using array subscript notation (arr[i]).

5. Dynamic Arrays and Pointers:

When you allocate memory for an array dynamically using the new operator, the pointer to the dynamically allocated memory is returned. This pointer can be used to access and manipulate the array elements. For example:

```
int size = 5;
int* dynamicArray = new int[size];  // Dynamically allocate memory
for (int i = 0; i < size; i++) {
  dynamicArray[i] = i + 1;  // Accessing and modifying dynamic array
```

3.6 Strings

A string is a sequence of characters that represents textual data in programming. It is a fundamental data type used to store and manipulate text or character-based information. In C++, strings are represented by the std::string class from the <string> library.

A string can contain any combination of alphabets, digits, symbols, spaces, and control characters. It is treated as a single entity and allows for various operations like concatenation, comparison, searching, and manipulation.

Strings in C++ have the following characteristics:

- Sequence of Characters: A string is a collection of characters, arranged in a specific order. Each character in the string occupies a position or index.
- Immutable: In C++, strings are immutable, meaning that once a string is created, its contents cannot be modified directly. Instead, you need to create a new string with the desired modifications.
- Dynamic Length: Strings can have a variable length, allowing you to store a flexible amount of text. They can grow or shrink dynamically based on the requirements.
- Null-Terminated: C-style strings in C++ are represented as character arrays terminated with a null character ('\0'). However, the std::string class handles null termination automatically.
- String Literals: String literals are enclosed in double quotes ("). They represent a sequence of characters and are automatically converted to std::string objects.

C++ provides a rich set of features and functions to handle strings efficiently. Here are the key aspects of working with strings in C++:

1. String Data Type:
    - C++ provides a built-in string data type called std::string in the <string> library.
    - It allows you to declare and manipulate strings easily, without worrying about memory allocation and deallocation.
2. String Declaration and Initialization:
    - Strings can be declared and initialized using various methods:
    - std::string str; - Declares an empty string.
    - std::string str = "Hello"; - Initializes a string with a specific value.

- std::string str("Hello"); - Alternative syntax for initialization.

3. String Operations:
    - Concatenation: Strings can be concatenated using the + operator or the += compound assignment operator.
    - Comparison: String comparison can be done using operators like ==, !=, <, >, <=, >=.
    - Accessing Characters: Individual characters of a string can be accessed using the subscript operator [] or the at() member function.
    - Length: The length of a string can be obtained using the length() or size() member functions.
    - Substrings: Substrings can be extracted from a string using the substr() member function.
4. Input and Output:

- Strings can be inputted and outputted using standard input/output streams (cin and cout).
- std::getline() function is used to read an entire line of text from input.

5. String Manipulation:

- C++ provides various functions in the <string> library for manipulating strings:
- std::stoi(), std::stod(), etc., for converting strings to numeric types.
- std::to_string() for converting numeric types to strings.
- std::toupper(), std::tolower(), etc., for converting case of characters.
- std::find(), std::replace(), std::substr(), etc., for searching and modifying strings.

6. String Iteration:

- Strings can be iterated using traditional loop constructs or with range-based for loops.
- Individual characters can be accessed and processed within the loop body.

7. String Memory Management:

- Memory allocation and deallocation for strings are automatically handled by the std::string class.
- You don't need to worry about memory management like you would with character arrays.

Here are some common string operations in C++

1. String Concatenation: The + operator or the += compound assignment operator can be used to concatenate strings.

```
std::string str1 = "Hello";

std::string str2 = "World";

std::string result = str1 + " " + str2; // Concatenating strings

std::cout << result << std::endl; // Output: Hello World
```

2. String Length: The length() or size() member functions can be used to get the length of a string.

```
std::string str = "Hello";

std::cout << str.length() << std::endl; // Output: 5
```

3. String Comparison:

String comparison can be done using the ==, !=, <, >, <=, >= operators.

```
std::string str1 = "Hello";

std::string str2 = "World";

if (str1 == str2) {

   std::cout << "Strings are equal" << std::endl;

} else {

   std::cout << "Strings are not equal" << std::endl;

}
```

4. Accessing Characters: Individual characters in a string can be accessed using the subscript operator [] or the at() member function.

```
std::string str = "Hello";
char firstChar = str[0]; // Accessing the first character
char lastChar = str.at(str.length() - 1); // Accessing the last character
std::cout << firstChar << std::endl; // Output: H
std::cout << lastChar << std::endl; // Output: o
```

5. Substrings: The substr() member function is used to extract substrings from a string.

```
std::string str = "Hello World";
std::string substr1 = str.substr(0, 5); // Extracting the first 5 characters
std::string substr2 = str.substr(6); // Extracting from index 6 till the end

std::cout << substr1 << std::endl; // Output: Hello
std::cout << substr2 << std::endl; // Output: World
```

6. **String Input and Output:**

Strings can be inputted and outputted using standard input/output streams (cin and cout).

```
    std::string name;
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "Hello, " << name << "!" << std::endl;
```

7. Finding Substrings: The find() member function is used to search for a substring within a string.

```
std::string str = "Hello World";

std::size_t found = str.find("World");

if (found != std::string::npos) {

   std::cout << "Substring found at index " << found << std::endl;

} else {

   std::cout << "Substring not found" << std::endl;

}
```

8. Replacing Substrings: The replace() member function is used to replace occurrences of a substring within a string.

```
std::string str = "Hello World";

str.replace(6, 5, "C++"); // Replace "World" with "C++"

std::cout << str << std::endl; // Output: Hello C++
```

9. Converting Case: The <cctype> library provides functions like toupper() and tolower() to convert the case of characters in a string.

```
#include <cctype>

std::string str = "Hello World";

for (auto& c : str) {

   if (std::islower(c)) {

      c = std::toupper(c);

   } else if (std::isupper(c)) {
```

For Vivekananda Global University, Jaipur

Registrar

```
        c = std::tolower(c);
    }
}
std::cout << str << std::endl; // Output: hELLO wORLD
```

10. Numeric to String Conversion: The std::to_string() function is used to convert numeric types to strings.

```
int num = 12345;
std::string str = std::to_string(num);


std::cout << str << std::endl; // Output: "12345"
```

11. String to Numeric Conversion: The <string> library provides functions like std::stoi() and std::stod() to convert strings to numeric types.

```
#include <string>
std::string str = "3.14";
double num = std::stod(str);
std::cout << num << std::endl; // Output: 3.14
```

3.7 Structures

A structure is a user-defined data type that allows you to group together different data items of various types into a single unit. Structures provide a way to represent a collection of related data, similar to a record in other programming languages.

To define a structure, you use the struct keyword followed by the structure name. Here's the general syntax:

```
struct StructureName {
```

For Vivekananda Global University, Jaipur

Registrar

```
  // Member variables (data items)

  // Member functions (optional)

};
```

The member variables inside a structure can be of any valid C++ data type, including primitive types like int, float, double, char, etc., as well as user-defined types and even other structures.

For example, let's create a simple structure called "Point" that represents a 2D coordinate:

```
struct Point {

  int x;

  int y;

};
```

In the above example, the Point structure has two member variables, x and y, both of type int.

Once you've defined a structure, you can create instances (also known as objects) of that structure type. You can then access the member variables of the structure using the dot operator (.). Here's an example:

```
Point p1; // Create an instance of the Point structure

p1.x = 3; // Access and modify member variables

p1.y = 5;


std::cout << "x = " << p1.x << ", y = " << p1.y << std::endl; // Print the values
```

In the above code, we create an instance p1 of the Point structure and assign values to its member variables x and y. We can then access and print the values using the dot operator (.).

C++ structures can also have member functions, similar to classes. These functions are called methods. To define a method inside a structure, you include the function declaration within the structure definition. Here's an example:

```
struct Rectangle {
  int width;
  int height;

  int calculateArea() {
    return width * height;
  }
};
```

In the above example, we define a structure called Rectangle with member variables width and height. It also includes a method called calculateArea(), which calculates and returns the area of the rectangle.

To use the structure's method, you would create an instance of the structure and then invoke the method using the dot operator (.). Here's an example:

```
Rectangle rect;
rect.width = 4;
rect.height = 5;

int area = rect.calculateArea();
std::cout << "Area = " << area << std::endl;
```

In the code above, we create an instance rect of the Rectangle structure, set its width and height member variables, and then call the calculateArea() method to calculate the area of the rectangle. The result is stored in the area variable and printed to the console.

It's important to note that structures have some differences compared to classes in C++. By default, the member variables of a structure are public, whereas in a class, they are private. However, you can explicitly specify the access level (public, private, protected) of the member variables and methods in both structures and classes using access specifiers.

3.8 Introduction to Flow control

Flow control refers to the ability to alter the order of execution of statements or blocks of code based on certain conditions or loops. It allows programmers to control the flow of program execution, making decisions, repeating code, and directing the program's behavior based on specific criteria.

Flow control statements enable programmers to determine which sections of code are executed and in what order. These statements include conditional statements (such as if, if-else, and switch), loop statements (such as while, for, and do-while), and control statements (such as break, continue, and goto).

Conditional statements, like the if statement and switch statement, evaluate a condition and execute specific blocks of code based on whether the condition is true or false. They provide the ability to make decisions and choose different paths of execution.

Loop statements, such as while, for, and do-while, allow code to be repeated multiple times. They provide the ability to iterate over a block of code as long as a specified condition is met. Loops are commonly used for tasks that require repetitive actions or processing of data.

Control statements, such as break, continue, and goto, allow programmers to modify the normal flow of execution within loops or switch statements. The break statement terminates the current loop or switch block, while the continue statement skips the remaining statements in the current iteration of a loop. The goto statement is a controversial statement that allows jumping to a labeled statement within the same function, but its usage is generally discouraged due to its potential to make code harder to understand and maintain.

By utilizing these flow control statements effectively, programmers can create programs that adapt to different scenarios, make decisions based on conditions, repeat tasks, and control the overall program flow. Flow control is crucial for writing structured, efficient, and flexible code in C++.

Flow control statements in programming languages, including C++, are used to control the flow of execution within a program. They determine the order in which statements or blocks of code are executed based on certain conditions or loops. Here are some commonly used flow control statements in C++:

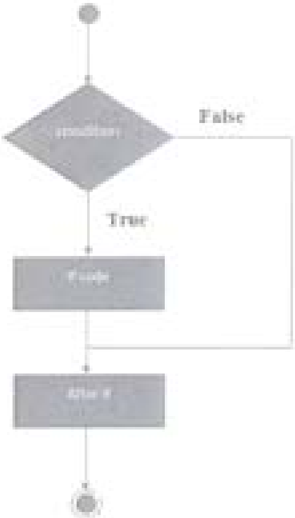
1. if Statement: The if statement allows the execution of a block of code if a specified condition is true. It is typically followed by an optional else statement to specify an alternative block of code to execute if the condition is false.
2. Switch Statement: The switch statement provides multiple branches of code execution based on the value of a variable or an expression. It allows selecting one of several code blocks to execute depending on the evaluated value.
3. while Loop: The while loop repeatedly executes a block of code as long as a specified condition remains true. It evaluates the condition before each iteration, and if the condition becomes false, the loop is terminated.
4. for Loop: The for loop provides a compact way to iterate a block of code a specific number of times. It consists of an initialization expression, a condition expression, and an increment or decrement expression. The loop continues executing as long as the condition is true.
5. do-while Loop: The do-while loop is similar to the while loop but with a different execution order. It executes a block of code at least once before evaluating the condition. If the condition is true, the loop continues; otherwise, it terminates.
6. break Statement: The break statement is used within loop or switch statements to terminate the execution of the innermost loop or switch block and continue executing the next statement after the loop or switch.
7. continue Statement: The continue statement is used within loop statements to skip the remaining statements in the current iteration and proceed to the next iteration of the loop.

8. goto Statement: The goto statement allows transferring control to a labeled statement within the same function. It is generally discouraged due to its potential to make code harder to understand and maintain.

3.9 “If” Statement

The "if" statement in C++ is used to execute a block of code if a specified condition is true. Here is the syntax of the "if" statement:

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



The "condition" in the "if" statement is an expression that evaluates to either true or false. If the condition is true, the code block enclosed within the curly braces is executed. If the condition is false, the code block is skipped, and program execution continues with the next statement after the "if" block.

Here's an example to demonstrate the usage of the "if" statement in C++:

```
#include <iostream>
int main()
{
   int num;
   std::cout << "Enter a number: ";
   std::cin >> num;

   if (num > 0)
```

For Vivekananda Global University, Jaipur

Registrar

```
    {
        std::cout << "The number is positive." << std::endl;
    }
    std::cout << "End of program." << std::endl;
    return 0;
}
```

In this example, the user is prompted to enter a number. The "if" statement checks if the entered number is greater than 0. If the condition is true, it prints "The number is positive." Otherwise, it moves on to the next statement after the "if" block. Finally, it outputs "End of program." to indicate the end of the program.

So, if the user enters a positive number, the output will be:

```
Enter a number: 7

The number is positive.

End of program.
```

If the user enters a negative number or 0, the output will be:

```
Enter a number: -5

End of program.
```

```
int number;

std::cout << "Enter a number: ";
std::cin >> number;

if (number > 0) {
    std::cout << "The number is positive." << std::endl;
}

if (number < 0) {
    std::cout << "The number is negative." << std::endl;
}

if (number == 0) {
    std::cout << "The number is zero." << std::endl;
}
```

In this example, each "if" statement is independent of the others. It checks a specific condition and prints a message if the condition is true. Therefore, multiple messages can be printed depending on the value of the input number.

3.9.1 Nested IF Statements

A nested "if" statement is an "if" statement that is placed inside another "if" statement. It allows for more complex conditional branching by providing additional levels of decision-making. The syntax of a nested "if" statement is as follows:

```
if (condition1) {
   // Code to be executed if condition1 is true

   if (condition2) {
      // Code to be executed if both condition1 and condition2 are true
   }
   // Other code within the outer "if" block
}
// Code outside the outer "if" block
```

Example:

```
int x = 10;
int y = 5;

if (x > y) {
   std::cout << "x is greater than y." << std::endl;

   if (x % 2 == 0) {
      std::cout << "x is even." << std::endl;
   } else {
      std::cout << "x is odd." << std::endl;
   }
} else {
   std::cout << "x is not greater than y." << std::endl;
}
```

In this example, the outer "if" statement checks if x is greater than y. If it is, the code within the outer "if" block is executed. Inside the outer "if" block, there is a nested "if" statement that checks if x is even or odd based on the condition x % 2 == 0. The appropriate message is printed based on the conditions.

Nested "if" statements can be extended to multiple levels based on the complexity of the decision-making process. However, it's important to maintain clarity and readability in code, so nesting should be used judiciously to avoid excessive complexity.

3.10 switch

The "switch" statement in C++ is a control statement that allows for multi-way branching based on the value of a variable or an expression. It provides a concise way to handle multiple cases without using multiple "if" statements. Here's the syntax of the "switch" statement:

```
switch (expression) {
  case value1:
    // Code to be executed if expression matches value1
    break;
  case value2:
    // Code to be executed if expression matches value2
    break;
  // Additional cases...
  default:
    // Code to be executed if none of the cases match
    break;
}
```

The expression is evaluated, and its value is compared to the values specified in the cases. If a match is found, the corresponding block of code is executed. The "break" statement is used to exit the switch block once a match is found. If no match is found, the code within the "default" block is executed (optional).

Here's an example that demonstrates the usage of the "switch" statement:

```
int day = 3;
switch (day) {
  case 1:
    std::cout << "Monday" << std::endl;
    break;
  case 2:
    std::cout << "Tuesday" << std::endl;
    break;
  case 3:
    std::cout << "Wednesday" << std::endl;
    break;
  case 4:
    std::cout << "Thursday" << std::endl;
    break;
  case 5:
    std::cout << "Friday" << std::endl;
    break;
  default:
    std::cout << "Weekend" << std::endl;
    break;
}
```

In this example, the value of the variable "day" is evaluated within the switch statement. Depending on the value, the corresponding case is executed. In this case, the output would be "Wednesday" because the value of "day" is 3.

Note that the "break" statement is important to prevent fall-through behavior, which means executing the code in subsequent cases until a break statement is encountered. If a break statement is omitted, it would continue executing code in subsequent cases even if their conditions are not met.

The "switch" statement is particularly useful when you have a limited number of discrete values to compare against and can provide a more concise and readable code structure compared to multiple "if-else" statements.

3.11 while Loop

The "while" loop in C++ is a control flow statement that repeatedly executes a block of code as long as a specified condition is true. It allows for iterative execution until the condition becomes false. Here's the syntax of the "while" loop:

```
while (condition) {

  // Code to be executed while the condition is true

  // The condition should eventually become false to exit the loop

}
```

The condition is an expression that is evaluated before each iteration of the loop. If the condition is true, the code within the loop is executed. After each iteration, the condition is evaluated again. If the condition becomes false, the loop is terminated, and the program continues with the next statement after the loop.

Here's an example that demonstrates the usage of the "while" loop:

```
int count = 1;

while (count <= 5) {

  std::cout << "Count: " << count << std::endl;

  count++;

}
```

In this example, the variable "count" is initially set to 1. The "while" loop continues executing as long as the condition "count <= 5" is true. Within each iteration, the current value of "count" is printed, and then "count" is incremented by 1. The loop will execute five times, producing the output:

```
Count: 1

Count: 2

Count: 3

Count: 4
```

```
Count: 5
```

It's important to ensure that the condition within the "while" loop eventually becomes false to prevent an infinite loop. If the condition is always true, the loop will keep executing indefinitely, causing the program to hang or become unresponsive.

You can also use control statements like "break" or "continue" within the "while" loop to alter its execution flow. The "break" statement can be used to exit the loop prematurely, while the "continue" statement can be used to skip the remaining code in the current iteration and proceed to the next iteration.

3.12 Do while

The "do-while" loop in C++ is a control flow statement that allows you to repeatedly execute a block of code while a certain condition is true. The key difference between the "do-while" loop and the "while" loop is that the "do-while" loop executes the code block at least once before checking the condition.

The syntax of the "do-while" loop in C++ is as follows:

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

Here's an example to illustrate the usage of the "do-while" loop:

```
int i = 1;

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

In this example, the "do-while" loop is used to print the numbers from 1 to 5. The loop first executes the code block, which prints the value of i and increments it by 1. Then, it checks the condition i <= 5. If the condition is true, the loop continues, and the code block is executed again. This process repeats until the condition becomes false.

The output of the above code will be: 1 2 3 4 5.

It's important to note that the condition is evaluated at the end of each iteration. This guarantees that the code block will be executed at least once, regardless of the condition. If the condition is false from the beginning, the code block will still execute once before the loop terminates.

Here's another example that demonstrates the use of the "do-while" loop to validate user input:

```
int number;

do {
  std::cout << "Enter a positive number: ";
  std::cin >> number;
} while (number <= 0);
```

In this example, the program prompts the user to enter a positive number. It continues to prompt the user until a positive number is entered. The condition number <= 0 is checked after the user input to determine whether to repeat the loop or terminate it.

The "do-while" loop is useful when you want to ensure that a block of code executes at least once, regardless of the condition.

3.13 Break

The "break" statement is used to terminate the execution of a loop or a switch statement. When the "break" statement is encountered, the control flow immediately exits the enclosing loop or switch statement, and the program continues with the next statement after the loop or switch.

The syntax for the "break" statement is as follows:

```
break;
```

Here are some examples to demonstrate the usage of the "break" statement:

- Exiting a loop:

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

  std::cout << i << " ";


  if (i == 3) {
        break; // Exit the loop when i reaches 3
    }
}

std::cout << "Loop ended." << std::endl;
```

In this example, the "for" loop iterates from 1 to 5. When the loop variable i reaches the value 3, the "break" statement is encountered, causing the loop to terminate immediately. The program then continues with the statement after the loop, which prints "Loop ended."

- Exiting a switch statement:

```
int choice = 2;
switch (choice) {
    case 1:
        std::cout << "Option 1 selected." << std::endl;
        break;
    case 2:
        std::cout << "Option 2 selected." << std::endl;
        break;
    case 3:
        std::cout << "Option 3 selected." << std::endl;
        break;
    default:
        std::cout << "Invalid option." << std::endl;
}

std::cout << "Switch statement ended." << std::endl;
```

Output:

```
Option 2 selected.
Switch statement ended.
```

In this example, the "switch" statement checks the value of the variable choice. When choice is 2, the corresponding case is executed, and the "break" statement is encountered, causing the switch statement to exit. The program then continues with the statement after the switch, which prints "Switch statement ended."

The "break" statement is essential for controlling the flow of execution in loops and switch statements, allowing you to exit prematurely based on certain conditions.

3.14 Continue

the "continue" statement is a control statement used within loops to skip the remaining iterations of the loop and continue with the next iteration. It is commonly used to skip certain iterations based on a specific condition. The "continue" statement is typically used with loops like "for" and "while".

The syntax of the "continue" statement is as follows:

```
for (initialization; condition; increment/decrement) {
  // Code before the continue statement


  if (condition_to_skip) {
    continue;
  }


  // Code after the continue statement
}
```

Here's an example that demonstrates the usage of the "continue" statement:

```
for (int i = 1; i <= 5; i++) {
  if (i == 3) {
    continue;  // Skip the current iteration when i is 3
  }
  std::cout << i << " ";
}
```

Output:

```
1 2 4 5
```

In this example, the "for" loop iterates from 1 to 5. However, when the value of i is 3, the "continue" statement is encountered, and the remaining code within the loop for that iteration is skipped. As a result, the number 3 is not printed in the output.

3.15 Goto

The "goto" statement in C++ is a control transfer statement that allows you to jump to a labeled statement within the same function. It provides an unconditional transfer of control, bypassing normal control flow constructs like loops and conditional statements. The syntax of the "goto" statement is as follows:

```
goto label;


// ...


label:
   // Statement(s) to be executed after the goto
```

The "label" is an identifier followed by a colon, and it marks a specific location in the code. The execution of the program will jump directly to the statement following the labeled position. Here's an example to illustrate the usage of "goto":

```
#include <iostream>
int main() {
   int number;
   std::cout << "Enter a positive number: ";
   std::cin >> number;
   if (number <= 0) {
      std::cout << "Invalid input. Exiting." << std::endl;
```

```
        goto end;
    }
    std::cout << "Entered number: " << number << std::endl;
        end:
    std::cout << "End of program." << std::endl;
    return 0;
}
```

In this example, the program prompts the user to enter a positive number. If the entered number is less than or equal to zero, the program jumps to the "end" label using the "goto" statement, skipping the subsequent code. The "end" label marks the end of the program, and the final message "End of program" is printed.

While the "goto" statement can be used to transfer control within a function, it is generally discouraged and considered bad practice in modern programming. The misuse of "goto" can make code difficult to understand and maintain, leading to spaghetti code and potential logical errors. In most cases, control flow constructs like loops and conditional statements provide clearer and more structured

3.16 If-Else

The "if-else" statement in C++ is used for conditional execution based on a certain condition. It provides an alternative branch of code to be executed when the condition is false. The general syntax of the "if-else" statement is as follows:

```
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

Here's an example to illustrate the usage of the "if-else" statement:

```
int number;


std::cout << "Enter a number: ";
std::cin >> number;


if (number % 2 == 0) {
   std::cout << "The number is even." << std::endl;
} else {
   std::cout << "The number is odd." << std::endl;
}
```

In this example, the program prompts the user to enter a number. If the number is divisible by 2 (i.e., the remainder is 0 when divided by 2), the condition number % 2 == 0 is true, and the code within the "if" block is executed, printing "The number is even." If the condition is false (i.e., the number is not divisible by 2), the code within the "else" block is executed, printing "The number is odd."

The "if-else" statement allows for mutually exclusive branches of code. If the condition in the "if" statement is true, only the code within the "if" block is executed, and the code within the "else" block is skipped entirely. Conversely, if the condition is false, the code within the "if" block is skipped, and only the code within the "else" block is executed.

It's important to note that the "else" block is optional. You can have an "if" statement without an "else" clause, as shown in the previous examples. In that case, if the condition is false, the program simply continues execution after the "if" block.

3.17 References

References in C++ provide a way to create an alias or an alternative name for an existing variable. They allow you to refer to the same memory location using different names. References are often used as function parameters to pass variables by reference, enabling the modification of the original variable.

1. Creating a Reference:

   To create a reference, you use the & operator after the data type when declaring the reference variable. For example:

   ```
   int x = 5;    // Declare an integer variable
   int& ref = x;  // Declare a reference to x
   ```

   In the example above, ref is a reference to the variable x. They both refer to the same memory location. Any changes made to ref will also affect x, and vice versa.

2. Using a Reference:

   References are used similarly to regular variables, but you don't need to use the & operator to access the value. For example:

   ```
   int x = 5;
   int& ref = x;
   ref = 10;   // Assigning a new value to ref also changes x
   std::cout << x << std::endl;  // Output: 10
   ```

   In the code snippet above, assigning a new value to ref also changes the value of x. When we output x, it will be 10.

3. References as Function Parameters:

   References are commonly used as function parameters to pass variables by reference, allowing the modification of the original variables. Consider the following example:

   ```
   void increment(int& num) {
   ```

```
  num++;
}

int main() {
  int x = 5;
  increment(x);
  std::cout << x << std::endl;  // Output: 6
  return 0;
}
```

In this example, the increment function takes an integer reference as a parameter. Any modifications made to num inside the function will affect the original variable x in the main function.

4. References vs. Pointers:
   - References and pointers are similar in that they both provide indirect access to variables. However, there are differences between them:
   - References cannot be reassigned to refer to a different variable once initialized, whereas pointers can be reassigned to point to different variables.
   - References cannot be null, whereas pointers can be set to a null value.
     - Pointers can perform arithmetic operations (pointer arithmetic), whereas references cannot.
     - References provide a more convenient syntax and are often preferred when you want to create an alias for a variable.

It's worth noting that references are not the same as the "reference" used in Java or C#. In C++, references are not object references or pointers to objects, but rather aliases for variables.

References in C++ are a powerful feature that allows for more efficient and readable code, especially when passing variables by reference in function calls. They provide a way to work with variables directly, avoiding the need for explicit pointer dereferencing syntax.

3.18 Summary

The ebook focuses on a range of topics including expression evaluation, type conversions, pointers, arrays, strings, structures, and references.

- Evaluation of Expressions: The ebook begins by examining the evaluation of expressions, offering a detailed explanation of arithmetic, logical, and relational operators. It delves into operator precedence and associativity, enabling readers to grasp the underlying principles of expression evaluation in C++.
- Type Conversions: The book delves into type conversions, both implicit and explicit, elucidating the rules and techniques involved in converting between different data types. By providing practical examples, readers gain a solid understanding of ensuring compatibility and managing data type conversions effectively.
- Pointers: Pointers, a powerful feature in C++, are thoroughly explored. The ebook covers pointer declaration, initialization, and dereferencing, emphasizing their role in memory manipulation and dynamic memory allocation. It delves into pointer arithmetic and illustrates how pointers enhance efficiency and flexibility in programming.
- Arrays: Arrays, fundamental data structures, are comprehensively discussed. The ebook covers array declaration, initialization, and accessing elements. Additionally, it examines array traversal, sorting, and searching techniques, equipping readers with essential skills for working effectively with arrays.
- Pointers and Arrays: This chapter explores the close relationship between pointers and arrays. It demonstrates how pointers can be used to iterate over array elements efficiently and showcases the dynamic allocation of multidimensional arrays using pointers.

- Strings: The ebook extensively covers strings, a vital component of C++ programming. It provides a thorough explanation of string manipulation, including concatenation, comparison, and searching. Readers also learn about standard library functions for efficient string handling.
- Structures: Structures are introduced as user-defined data types. The ebook explains the creation and usage of structures, including member variables and methods. It illustrates how structures can be employed as containers for related data items, demonstrating structure initialization, access, and utilization.
- References: The ebook concludes with an exploration of references, which serve as aliases for existing variables. It outlines the creation and application of references, highlighting their use as function parameters for efficient variable manipulation.

- The "if-else" statement allows for conditional execution, where a block of code is executed if a specific condition is true, and an alternative block is executed if the condition is false. It provides a way to make decisions based on certain conditions.
- Loops are used to repeat a block of code multiple times until a specified condition is met. The "while" loop repeatedly executes a block of code as long as a condition is true. The "do-while" loop is similar, but it always executes the block of code at least once before checking the condition.
- The "goto" statement provides the ability to transfer control to a labeled statement elsewhere in the program. However, the use of "goto" is generally discouraged as it can make the code harder to understand and maintain.
- The "break" statement is used to exit a loop or switch statement prematurely. When encountered, it immediately terminates the loop and continues executing the next statement after the loop. It is commonly used to exit a loop early based on certain conditions.
- The "continue" statement is used to skip the remaining code within a loop iteration and move to the next iteration. It allows you to bypass specific iterations without terminating the loop entirely.
- These control structures provide powerful tools for managing program flow and executing code selectively or repetitively based on conditions. However, it's important

to use them judiciously and ensure that the code remains readable, maintainable, and free from unnecessary complexity.

3.19 Keywords

- Evaluation of expressions: Process of calculating the value of an expression.
- Type conversions: Changing the data type of a variable to another compatible type.
- Pointers: Variables that store memory addresses, allowing manipulation of memory and dynamic memory allocation.
- Arrays: Collection of elements of the same data type, accessed using an index.
- Strings: Sequences of characters, often used to represent text.
- Structures: User-defined data types that group related data items into a single unit.
- References: Aliases for existing variables, providing an alternate name to refer to the same memory location.
- C++ Strings: Objects that represent sequences of characters in C++, providing various string manipulation operations.
- Array Declaration: Creating a fixed-size collection of elements of the same data type in C++.
- Array Initialization: Assigning initial values to the elements of an array during declaration or later.
- Array Access: Retrieving or modifying the value of an element in an array using its index.
- Pointer Declaration: Creating a variable that holds the memory address of another variable.
- Pointer Arithmetic: Performing arithmetic operations on pointers, such as addition, subtraction, and comparison.
- Dynamic Memory Allocation: Allocating memory at runtime using pointers, enabling flexible memory management.
- Structure Initialization: Assigning initial values to the member variables of a structure during declaration or later.

- Structure Access: Retrieving or modifying the value of member variables in a structure using the dot operator.
- Reference Variables: Aliases for existing variables, providing an alternative name to refer to the same value or object.
- Passing by Reference: Passing variables to functions using references, allowing modification of the original variables.
- Structure Methods: Functions defined within a structure, allowing operations and manipulations specific to the structure's data.
- Structure Nesting: Defining structures within structures, creating a hierarchy or composition of related data.
- Const References: References that provide read-only access to a variable, preventing modifications through the reference.
- Pointer to Structure: Creating a pointer that points to a structure, enabling indirect access and manipulation of structure data.
- if: Executes a block of code if a specified condition is true.
- switch: Evaluates an expression and executes a block of code based on different cases or values.
- while: Repeatedly executes a block of code as long as a specified condition is true.
- for: Executes a block of code repeatedly for a specified number of times, with a loop control variable.
- do: Executes a block of code at least once and then repeatedly as long as a specified condition is true.
- break: Terminates the execution of a loop or switch statement.
- continue: Skips the remaining code within a loop iteration and moves to the next iteration.
- goto: Transfers control to a labeled statement elsewhere in the program (usually discouraged due to its potential to create complex and hard-to-maintain code).

3.20 Review Questions

1. What is the purpose of type conversions in C++? Provide an example of an implicit and explicit type conversion.
2. How do pointers facilitate memory manipulation and dynamic memory allocation in C++? Explain with an example.
3. Describe the process of declaring and initializing an array in C++. How are array elements accessed?
4. How are strings represented and manipulated in C++? Discuss common string operations and available library functions.
5. What are structures in C++? How do they differ from classes? Provide an example of a structure with member variables and methods.
6. Explain the concept of references in C++. How are they created and used? Compare references with pointers.
7. Discuss the advantages of passing variables by reference in function parameters. Provide an example to demonstrate the impact of passing by reference.
8. How are pointers and arrays related in C++? Explain how pointers can be used to access and manipulate array elements efficiently.
9. What is the significance of dynamic memory allocation using pointers? Provide an example of allocating and deallocating memory dynamically.
10. How can structures be nested within other structures in C++? Illustrate with an example the use of nested structures and accessing their member variables.
11. What is the purpose of the "if-else" statement in C++?
12. How does a "switch" statement differ from an "if-else" statement in terms of functionality and usage?
13. What is the difference between a "while" loop and a "do-while" loop in C++?
14. Explain the concept of a loop control variable and its role in a "for" loop.
15. What is the purpose of the "break" statement in C++? Provide an example scenario where it would be useful.
16. How does the "continue" statement differ from the "break" statement? Give an example to illustrate its usage.
17. Discuss the potential drawbacks and considerations when using the "goto" statement in C++.
18. How does the flow of execution change when encountering an "if-else" statement within a loop?
19. Explain the concept of nested control structures and provide an example scenario where they might be used.
20. How do flow control statements like "if-else," loops, and break/continue contribute to the overall control flow and logic of a C++ program?

3.12 References

1. Shiffman, D. (2016). Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (2nd ed.). Morgan Kaufmann.

Unit - 4 Functions

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Understand the scope of variables: Gain a comprehensive understanding of variable lifetimes and visibility within different scopes, including global, local, and function scopes. Learn to prevent naming conflicts and write code that is easier to understand and maintain.
- Master parameter passing: Explore various methods of passing arguments to functions, such as pass-by-value, pass-by-reference, and pass-by-pointer. Understand the advantages and limitations of each method and apply them effectively in your code.
- Harness the power of default arguments: Learn how to define default values for function parameters, making your functions more flexible and allowing them to be called with varying levels of input. Develop an understanding of when and how to utilize default arguments in your programs.
- Optimize performance with inline functions: Discover the concept of inline functions and their ability to reduce function call overhead, thereby improving program efficiency. Learn to identify suitable scenarios for inline functions and make informed decisions regarding their usage.
- Grasp the principles of recursive functions: Explore the concept of recursion, where functions call themselves to solve complex problems. Develop a solid understanding of recursive algorithms and their application in solving a variety of programming challenges.
- Master pointers to functions: Gain proficiency in working with pointers to functions, which allow you to store and invoke functions dynamically. Understand the syntax and usage of function pointers, enabling you to leverage this powerful feature in advanced programming scenarios.
- Apply acquired knowledge through practical examples: Throughout the eBook, engage in hands-on coding exercises and work through real-world examples to reinforce your understanding of each topic. Apply the concepts learned to solve problems and build robust C++ programs.

- Enhance problem-solving skills: Develop a problem-solving mindset by decomposing complex tasks into smaller, manageable functions. Learn to analyze problems effectively, design efficient algorithms, and implement them using the concepts covered in this eBook.
- Foster code reusability and modularity: Understand how functions contribute to code organization and reusability. Learn to design functions that are modular, independent, and easily maintainable, thereby enhancing the overall structure and efficiency of your programs.
- Prepare for advanced C++ programming: Lay a strong foundation for further exploration of advanced C++ topics. The knowledge gained from mastering functions will serve as a stepping stone to delve deeper into other aspects of the language, such as object-oriented programming, templates, and advanced data structures..

Introduction

This eBook will take you on a journey through the intricacies of C++ functions, exploring key concepts such as scope of variables, parameter passing, default arguments, inline functions, recursive functions, and pointers to functions. Each of these topics has its significance, and understanding them will empower you to write elegant and robust code. In the first section, we will delve into the scope of variables. You will learn about the lifetime and visibility of variables within different scopes, including global, local, and function scopes. Understanding variable scope is crucial for writing maintainable code and preventing naming conflicts.

Next, we will explore parameter passing in functions. You will discover the various ways of passing arguments to functions, such as pass-by-value, pass-by-reference, and pass-by-pointer. We will discuss the pros and cons of each method and provide examples to solidify your understanding.

Default arguments are another valuable feature in C++ functions that allow you to provide default values for function parameters. We will explain how default arguments work and demonstrate their usage in practical scenarios. This knowledge will enable you to write flexible functions that can be called with varying levels of input. inline functions offer a way to improve performance by avoiding the overhead of function calls. We will explore the concept of inline functions and discuss when and how to use them effectively. You will gain insights into the trade-offs between inline functions and regular functions and learn to make informed decisions. Recursive functions are a powerful technique in programming, allowing functions to call themselves. We will dive into the world of recursive functions, understanding the underlying principles and exploring their potential applications. You will gain the skills needed to solve complex problems and implement elegant recursive algorithms.

Finally, we will unravel the concept of pointers to functions. Pointers to functions allow you to store addresses of functions and invoke them dynamically. We will walk you through the syntax and usage of function pointers, helping you understand their role in advanced programming scenarios. Throughout this eBook, we will provide clear explanations, code examples, and practical exercises to reinforce your understanding of these topics. By the end, you will have a solid grasp of functions in C++, equipping you with the knowledge to write efficient, modular, and scalable code.

4.1 Functions

Functions in C++ are a fundamental building block of code that encapsulates a sequence of statements and performs a specific task. They are essential for organizing and structuring programs, promoting code reuse, and improving overall program efficiency. In C++, functions are defined using a specific syntax and can have various characteristics and behaviors.

Here are some key aspects of functions in C++:

1. Function Declaration: A function declaration specifies the function's name, return type, and parameters (if any). It provides a blueprint for the function's implementation and allows other parts of the program to call the function.
   Syntax:

   ```
   return_type function_name(parameter_list);
   ```

Example:

```
// Function declaration
int sum(int a, int b);
```

Explanation: The function declaration specifies the name (sum), return type (int), and parameters (a and b) of the function. It informs the compiler about the existence and signature of the function.

2. Function Definition: The function definition contains the actual implementation of the function. It includes the function's body, which consists of a sequence of statements enclosed within curly braces.

Syntax:

Copy code

```
return_type function_name(parameter_list)
{
  // Function body
  // Statements to be executed
  // ...
}
```

Example:

```
// Function definition
int sum(int a, int b)
{
  int result = a + b;
  return result;
}
```

The function definition contains the implementation of the function. In this example, the function sum takes two parameters (a and b), calculates their sum, and returns the result as an int value.

3. Return Type: Every function in C++ has a return type, which specifies the type of data that the function will return after executing its statements. The return type can be any valid C++ data type, including fundamental types, user-defined types, or even void (indicating no return value).

   Syntax:

```
return_type function_name(parameter_list)
{
  // Function body
  // Statements to be executed
  // ...
  return value; // Return statement
}
```

   Example:

```
// Function definition with void return type
void greet()
{
  cout << "Hello, World!" << endl;
}
```

   The return type specifies the type of data that the function will return after executing its statements. In this example, the greet function has a return type of void, indicating that it does not return any value.

4. Parameters: Functions can accept zero or more parameters, which are variables used to pass values into the function. Parameters define the input required by the function to perform its task. Each parameter has a type, a name, and is enclosed within parentheses in the function declaration.

   Syntax:

```
return_type function_name(parameter_type parameter_name)

{

  // Function body

  // Statements to be executed

  // ...

}
```

   Example:

```
// Function definition with parameters

int multiply(int a, int b)

{

  int result = a * b;

  return result;

}
```

   Parameters allow values to be passed into functions. In this example, the multiply function takes two parameters (a and b), multiplies them, and returns the result.

5. Function Call: To execute a function and utilize its functionality, it needs to be called from other parts of the program. A function call includes the function name followed by parentheses. If the function has parameters, the values to be passed are provided within the parentheses.

Syntax:

```
function_name(argument_list);
```

Example:

```
// Function call
int x = 3, y = 5;
int sum_result = sum(x, y);
```

Explanation: To execute a function and utilize its functionality, we call the function by providing the required arguments within parentheses. In this example, the sum function is called with the arguments x and y, and the returned value is stored in the sum_result variable.

6. Function Prototypes: A function prototype is a declaration that provides the function's name, return type, and parameter list without the function's actual implementation. Prototypes are often placed in header files and allow functions to be declared before they are defined. They facilitate code organization and enable separate compilation of program modules.

   A function prototype, also known as a function declaration, is a statement that provides the necessary information about a function before its actual implementation. It serves as a forward declaration of the function, allowing other parts of the program to know about the function's existence, return type, and parameter list without needing to see its complete implementation.

   The syntax for a function prototype is similar to that of a function declaration:

```
return_type function_name(parameter_list);
```

   Here's an example of a function prototype:

```
int sum(int a, int b);
```

   In this example, the function prototype declares a function named sum that takes two int parameters and returns an int value. It provides the necessary information about the function's signature, enabling other parts of the program to use this function before its actual definition. Function prototypes are often placed in header files (.h) or at the beginning of a source code file to provide a clear interface and facilitate modular programming. By separating the function declaration from its implementation, function prototypes allow for separate compilation of program modules, reducing compilation time and improving code organization.

7. Function Libraries: C++ provides numerous pre-defined functions through standard libraries. These libraries contain a wide range of functions that perform various tasks, such as mathematical calculations, input/output operations, string manipulation, and more. To use functions from libraries, appropriate header files need to be included in the program.

4.2 Scope of Variable

The scope of a variable refers to the region of a program where the variable is visible and accessible. It defines the part of the program where the variable can be used, and it determines its lifetime and visibility to other parts of the program.

In C++, there are several types of variable scope:

- Global Scope: Variables declared outside of any function or block have global scope. They are accessible from any part of the program, including all functions and blocks. Global variables have a lifetime that spans the entire execution of the program.

  Example:

  ```
  #include <iostream>


  int globalVariable = 10; // Global variable


  void function()

  {
  ```

```
    std::cout << globalVariable << std::endl; // Accessible within functions
}

int main()
{
    std::cout << globalVariable << std::endl; // Accessible within main function
    return 0;
}
```

- Local Scope: Variables declared within a function or block have local scope. They are accessible only within the specific function or block where they are defined. Local variables have a lifetime limited to the duration of the function or block execution.

  Example:

```
#include <iostream>
void function()
{
    int localVariable = 5; // Local variable
    std::cout << localVariable << std::endl; // Accessible within function
}
int main()
{
    // std::cout << localVariable << std::endl; // Error: localVariable is not accessible outside the function
    function();
```

```
  return 0;
}
```

- Function Parameter Scope: Parameters passed to a function have a scope limited to the function's body. They behave like local variables within the function and are accessible only within that function.

```
#include <iostream>
void function(int parameter)
{
  std::cout << parameter << std::endl; // Accessible within function
}
int main()
{
  // std::cout << parameter << std::endl; // Error: parameter is not accessible outside
the function
  function(10);
  return 0;
}
```

- Block Scope: Variables declared within a block, denoted by a pair of curly braces {}, have a scope limited to that block. They are accessible only within the block where they are defined.

Example:

```
#include <iostream>
int main()
{
  {
        int blockVariable = 7; // Block variable
        std::cout << blockVariable << std::endl; // Accessible within block
    }
    // std::cout << blockVariable << std::endl; // Error: blockVariable is not accessible outside the block
    return 0;
}
```

Variable scope plays a crucial role in managing the visibility and lifetime of variables. It helps prevent naming conflicts, enhances code readability, and ensures that variables are only accessible where they are intended to be used. By understanding variable scope, you can effectively organize and manage variables in your C++ programs.

4.3 Parameter passing

Parameter passing in C++ refers to the mechanism by which values are transferred to function parameters when a function is called. It determines how arguments are passed from the calling code to the corresponding parameters in the function's definition. C++ supports different parameter passing methods, including pass-by-value, pass-by-reference, and pass-by-pointer.

Pass-by-Value:

In pass-by-value, the value of the argument is copied into the function's parameter. Any modifications made to the parameter within the function do not affect the original argument in the calling code.

Example:

```
void square(int num) {
  num = num * num; // Modify the parameter
  cout << "Square inside function: " << num << endl;
}
```

```
int main() {
   int x = 5;
   square(x); // Pass x by value
   cout << "Original value of x: " << x << endl;
   return 0;
}
```

Output:

```
Square inside function: 25
Original value of x: 5
```

In this example, the value of x is passed to the square function by value. Any changes made to the num parameter within the function do not affect the original value of x in the main function.

Pass-by-Reference:

In pass-by-reference, a reference to the argument is passed to the function's parameter. This allows the function to directly access and modify the original argument in the calling code.

Example:

```
void square(int& num) {
   num = num * num; // Modify the reference
   cout << "Square inside function: " << num << endl;
}
int main() {
   int x = 5;
   square(x); // Pass x by reference
   cout << "Modified value of x: " << x << endl;
   return 0;
}
```

Output:

```
Square inside function: 25
Modified value of x: 25
```

In this example, the reference to x is passed to the square function by reference. The changes made to the num parameter within the function directly modify the original value of x in the main function.

Pass-by-Pointer:

In pass-by-pointer, a pointer to the argument is passed to the function's parameter. The function can access and modify the value pointed to by the pointer, allowing changes to propagate to the original argument.

Example:

```
void square(int* ptr) {
   *ptr = (*ptr) * (*ptr); // Modify the value pointed to by the pointer
   cout << "Square inside function: " << *ptr << endl;
}

int main() {
   int x = 5;
   square(&x); // Pass the address of x (pointer to x)
   cout << "Modified value of x: " << x << endl;
   return 0;
}
```

Output:

```
Square inside function: 25
Modified value of x: 25
```

In this example, the address of x is passed to the square function as a pointer. The function dereferences the pointer (*ptr) to access and modify the value stored in x, resulting in changes reflected in the main function.

Parameter passing methods determine whether modifications made to the parameter within the function affect the original argument or not. Each method has its advantages and considerations, and the choice depends on the desired behavior and the requirements of the program.

4.4 Default Arguments

Default arguments allow to assign default values to function parameters. When a function is called, if an argument is not provided for a parameter with a default value, the default value is used instead. This provides flexibility by allowing some parameters to be optional when calling the function.

The syntax for defining a default argument is as follows:

```
return_type function_name(parameter_type parameter_name = default_value);
```

Here's an example that demonstrates the usage of default arguments:

```
#include <iostream>


void greet(std::string name = "Guest") {
   std::cout << "Hello, " << name << "!" << std::endl;
}
int main() {
   greet();        // Uses the default value "Guest"
   greet("Alice"); // Overrides the default value with "Alice"
   return 0;
}
```

Output:

Copy code

```
Hello, Guest!
Hello, Alice!
```

In this example, the greet function has a default argument for the name parameter, which is set to "Guest". When the function is called without providing an argument for name, it uses the default value. However, if an argument is passed, it overrides the default value.

Default arguments can be specified for one or more parameters in a function. When using default arguments, it is important to note that parameters with default values must be defined at the end of the parameter list. In other words, all parameters with default values should appear after parameters without default values.

Default arguments are particularly useful when a function has parameters that are commonly used with a specific value but occasionally need to be customized. They enhance the flexibility and usability of functions, making the code more concise and reducing the need for function overloads or separate functions to handle different cases.

4.5 Inline Functions

Inline functions in C++ are a compiler optimization feature that allows the code of a function to be inserted directly at the call site instead of performing a function call. This can improve the performance of small and frequently used functions by reducing the overhead of function call mechanisms.

To declare an inline function, the inline keyword is used before the function definition. The function can be defined within the class declaration (for member functions) or outside the class declaration (for non-member functions).

Here's an example that demonstrates the usage of inline functions:

```
#include <iostream>

// Inline function declaration

inline int square(int num) {

   return num * num;

}

int main() {

   int x = 5;

   int result = square(x); // Function call is replaced with the actual code of the function
```

```
    std::cout << "Square: " << result << std::endl;

    return 0;

}
```

Output:

```
Square: 25
```

In this example, the square function is declared as inline. When the function is called, instead of performing a function call, the code of the function square is directly inserted at the call site. This eliminates the overhead of the function call and improves performance.

It's important to note that inline functions are typically used for small functions with a few lines of code. The decision to inline a function is made by the compiler, which analyzes various factors such as function size, complexity, and the surrounding context. The inline keyword serves as a hint to the compiler, but the compiler has the final say on whether to inline the function or not.

Inline functions can be beneficial for performance-critical code or in situations where function call overhead needs to be minimized. However, it's important to balance the benefits of inlining with code size and maintainability. Inlining large or complex functions may lead to increased executable size and potentially hinder code readability.

4.6 Recursive Functions

Recursive functions in C++ are functions that call themselves either directly or indirectly. They solve complex problems by breaking them down into smaller subproblems and solving each subproblem recursively until a base case is reached. Recursive functions typically have two parts: the base case and the recursive case.

The base case specifies a condition that determines when the recursion should stop. When the base case is reached, the function stops calling itself and returns a value. The recursive case defines how the function calls itself with smaller inputs to make progress towards the base case.

Here's an example of a recursive function that calculates the factorial of a number:

```
#include <iostream>

int factorial(int n) {

  // Base case: factorial of 0 is 1

  if (n == 0) {

    return 1;

  }


  // Recursive case: factorial of n is n multiplied by factorial of (n-1)

  return n * factorial(n - 1);

}

int main() {

  int num = 5;

  int result = factorial(num);

  std::cout << "Factorial of " << num << " is: " << result << std::endl;

  return 0;

}
```

Output:

```
Factorial of 5 is: 120
```

In this example, the factorial function calculates the factorial of a number n. If the base case n == 0 is reached, the function returns 1. Otherwise, it recursively calls itself with n - 1 and multiplies the current value of n with the factorial of n - 1 to make progress towards the base case.

Example calculates the sum of integers from 1 to a given number:

```
#include <iostream>
int calculateSum(int n) {
   // Base case: if n is 1, return 1
   if (n == 1) {
      return 1;
   }

   // Recursive case: return the sum of n and the sum of integers from 1 to (n-1)
   return n + calculateSum(n - 1);
}
int main() {
   int num = 5;
   int result = calculateSum(num);
   std::cout << "Sum of integers from 1 to " << num << " is: " << result << std::endl;
   return 0;
}
```

Output:

```
Sum of integers from 1 to 5 is: 15
```

In this example, the calculateSum function recursively calculates the sum of integers from 1 to a given number n. If the base case n == 1 is reached, the function returns 1. Otherwise, it recursively calls itself with n - 1 and adds the current value of n to the sum of integers from 1 to n - 1 to make progress towards the base case.

Recursive functions are useful when the problem can be divided into smaller subproblems that are similar in nature to the original problem. They provide an elegant solution for problems that can be solved iteratively as well. However, it's important to ensure that recursive functions have well-defined base cases and are properly structured to avoid infinite recursion.

4.7 Pointers to Functions

Pointers to functions allow to store the address of a function in a variable. They provide a way to dynamically select and invoke functions at runtime, enabling greater flexibility and extensibility in your code.

To declare a pointer to a function, you need to specify the function's signature, including the return type and parameter types. The syntax for declaring a pointer to a function is as follows:

```
return_type (*function_pointer_name)(parameter_types);
```

Here's an example that demonstrates the usage of function pointers:

```
#include <iostream>
int add(int a, int b) {
  return a + b;
}
int subtract(int a, int b) {
  return a - b;
}
int main() {
  int (*operation)(int, int); // Declare a function pointer
  operation = add; // Point to the add function
  int result = operation(5, 3); // Call the function through the pointer
  std::cout << "Result of addition: " << result << std::endl;
  operation = subtract; // Point to the subtract function
```

```
    result = operation(5, 3); // Call the function through the pointer

    std::cout << "Result of subtraction: " << result << std::endl;

    return 0;

}
```

Output:

```
Result of addition: 8

Result of subtraction: 2
```

In this example, two functions add and subtract are defined. We declare a function pointer operation that can point to functions with the same signature (in this case, taking two int parameters and returning an int). We assign the address of the add function to operation and call it through the pointer, resulting in the addition of two numbers. Similarly, we assign the address of the subtract function to operation and call it through the pointer, resulting in the subtraction of two numbers.

Function pointers are particularly useful in scenarios where you need to dynamically select and invoke different functions based on certain conditions or configurations. They provide a level of indirection and runtime flexibility in function invocation, enabling you to design more generic and reusable code.

It's important to note that function pointers can be assigned nullptr to indicate that they are not pointing to any valid function. Additionally, when calling functions through function pointers, make sure that the pointer is not null to avoid crashes or undefined behavior.

4.8 Summary

In this chapter, we explored various aspects of functions in C++. We covered the scope of variables, parameter passing mechanisms, default arguments, inline functions, recursive functions, and pointers to functions. We began by discussing the scope of variables within functions. Variables declared inside a function have local scope and are accessible only within that function. We also learned about the concept of block scope, where variables declared within a block are only visible within that block. Next, we delved into parameter passing, which determines how arguments are transferred to function parameters. We explored three methods: pass-by-value, pass-by-reference, and pass-by-pointer. Pass-by-value involves making a copy of the argument, pass-by-reference uses a reference to the argument, and pass-by-pointer passes a pointer to the argument. Each method has its advantages and considerations, depending on the desired behavior and requirements of the program. We then covered default arguments, which allow us to assign default values to function parameters. If an argument is not provided when calling a function, the default value is used. This provides flexibility by making certain parameters optional and simplifying function calls in certain scenarios. Inline functions were introduced as a compiler optimization feature. An inline function's code is directly inserted at the call site instead of performing a function call. This can improve performance for small, frequently used functions by reducing the overhead of function call mechanisms.

Recursive functions were explored as a powerful technique for solving complex problems by breaking them down into smaller subproblems. Recursive functions call themselves either directly or indirectly, and they rely on base cases to determine when to stop the recursion. Recursive functions allow elegant and efficient solutions for certain problems, but proper base cases and termination conditions are crucial to avoid infinite recursion.

Lastly, we examined pointers to functions, which enable storing the address of a function in a variable. Function pointers provide flexibility and runtime selection of functions, allowing dynamic invocation of different functions based on conditions or configurations. Understanding functions and their various aspects is fundamental in C++ programming. By mastering the scope of variables, parameter passing mechanisms, default arguments, inline functions, recursive functions, and pointers to functions, programmers gain a powerful set of tools to design modular, flexible, and efficient code.

4.9 Keywords

- Functions: Modular blocks of code that perform specific tasks.

- Scope of variables: The region or portion of the code where a variable is visible and accessible.
- Parameter passing: The mechanism of transferring values to function parameters during function calls.
- Default arguments: Values assigned to function parameters if no argument is provided during function calls.
- Inline functions: Functions defined as inline, allowing the compiler to insert their code directly at the call site for performance optimization.
- Recursive functions: Functions that call themselves either directly or indirectly to solve problems by dividing them into smaller subproblems.
- Pointers to functions: Variables that store addresses of functions, allowing dynamic function selection and invocation.
- Function overloading: The ability to define multiple functions with the same name but different parameter lists, allowing different versions of a function to be called based on the arguments provided.
- Function templates: A mechanism that allows the definition of generic functions that can operate on different data types, providing code reusability.
- Function pointers: Variables that store the address of a function, allowing indirect function invocation and enabling dynamic function dispatch.
- Call by value: A parameter passing mechanism where the value of the argument is copied into the function parameter.
- Call by reference: A parameter passing mechanism where the reference to the argument is passed to the function parameter, allowing direct access and modification of the original argument.
- Call by pointer: A parameter passing mechanism where the pointer to the argument is passed to the function parameter, allowing access and modification of the value pointed to by the pointer.

4.10 Review Questions

1. What is the scope of a variable in C++? How does it relate to functions?
2. Explain the concept of parameter passing in C++. What are the different methods of parameter passing?

3. What are default arguments in C++? How are they useful in function definitions?
4. Describe the purpose and benefits of using inline functions in C++.
5. What are recursive functions? How do they work? Provide an example.
6. Explain the importance of a base case in recursive functions. Why is it necessary?
7. What is a function pointer in C++? How is it declared and used?
8. How can you pass a function as an argument to another function in C++?
9. Discuss the advantages and considerations when using default arguments in function declarations.
10. Compare and contrast pass-by-value, pass-by-reference, and pass-by-pointer in terms of parameter passing mechanisms.

4.11 References

1. Shiffman, D. (2016). Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (2nd ed.). Morgan Kaufmann.

Unit – 5 C++ Classes And Data Abstraction

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Understand the concept of a class and its role in object-oriented programming.
- Define and declare classes in C++, including data members and member functions.
- Explore the structure of a class, including access specifiers and their impact on member visibility.
- Instantiate objects from a class and initialize them using constructors.
- Manipulate class objects and access their data using member functions.
- Understand the scope and lifetime of class members.
- Grasp the concept of the "this" pointer and its role in object-oriented programming.
- Implement friend classes and functions to access private members of a class.
- Apply best practices for designing and organizing classes for efficient and maintainable code.
- Gain hands-on experience through practical examples, code snippets, and exercises.
- Develop a solid foundation in working with C++ classes and data abstraction.
- Apply the knowledge gained to build object-oriented solutions and enhance software development skills.

Introduction

In the world of programming, managing complexity is a crucial aspect of developing robust and maintainable software. Object-oriented programming, with its emphasis on modularity and reusability, offers an effective approach to tackle complexity in software design. At the heart of object-oriented programming lies the concept of classes, which enable us to model real-world entities and encapsulate their properties and behaviors into cohesive units. C++ provides robust support for object-oriented programming, allowing us to create classes that define the structure and behavior of objects. This ebook aims to demystify the intricacies of C++ classes and data abstraction, providing you with a solid foundation to harness the power of object-oriented programming. We will begin by understanding the concept of a class and its role in C++ programming. You will learn how to define a class, its members, and their visibility within the class. We will explore the various types of class members, such as data members and member functions, and understand how they contribute to the overall functionality of a class.

Once we have a solid understanding of class structure, we will dive into the creation and utilization of class objects. You will learn how to instantiate objects from a class, initialize them, and manipulate their data using member functions. We will also explore the concept of constructors and destructors, which provide a mechanism for initializing and cleaning up object resources, respectively. As we progress, we will explore the intricacies of class scope and understand how the visibility of class members is controlled. We will discuss the use of access specifiers, namely public, private, and protected, to define the accessibility of class members from within and outside the class. Furthermore, we will delve into the fascinating concept of the "this" pointer. You will discover how the "this" pointer allows objects to refer to themselves, enabling member functions to access and modify their own data. We will explore the various use cases of the "this" pointer and understand its role in object-oriented programming. Finally, we will explore the idea of friends to a class. You will learn how to declare friend classes and functions that can access private members of a class, providing controlled access to the internals of a class. We will examine the implications and best practices of using friends in your code. Throughout this ebook, we will provide practical examples and code snippets to illustrate the concepts and techniques discussed. Additionally, we will include exercises and challenges to reinforce your understanding and give you hands-on experience in working with C++ classes and data abstraction. By the end of this ebook, you will have gained a solid understanding of C++ classes and data abstraction, empowering you to design and implement robust object-oriented solutions in your own projects. So let's dive in and explore the exciting world of C++ classes!

5.1 Introduction to Object-Oriented Programming (OOP) and C++ Classes

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into reusable and modular units called objects. These objects encapsulate both data and the operations or behaviors that manipulate that data. OOP promotes concepts such as abstraction, encapsulation, inheritance, and polymorphism, enabling developers to create complex software systems with ease.

In OOP, the key principles include:

- Encapsulation: Encapsulation involves bundling data and the methods that operate on that data together into a single unit called an object. This ensures that the data is accessed and modified only through defined methods, protecting it from unauthorized access and maintaining data integrity.
- Abstraction: Abstraction focuses on representing complex real-world entities as simplified models within the program. It allows developers to create classes that define essential characteristics and behaviors while hiding the implementation details. Users of the class interact with the abstraction without needing to know its internal workings.
- Inheritance: Inheritance enables the creation of new classes based on existing classes, known as base or parent classes. The derived or child classes inherit the properties and behaviors of the base class, allowing for code reuse and promoting a hierarchical structure. It helps to model relationships and hierarchies among classes.
- Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables the use of a single interface to represent different types of objects. Polymorphism can be achieved through function overloading, where multiple functions with the same name but different parameters coexist, or through function overriding,

where derived classes provide their own implementation of a method defined in the base class.

5.1.1 C++ Class

In C++, a class is a user-defined data type that serves as a blueprint for creating objects. A class defines the structure, behavior, and properties of objects that can be instantiated from it. It combines data members (variables) and member functions (methods) into a cohesive unit, allowing for better organization and modularization of code.

Class Definition:

Class is a fundamental concept in C++ that allows you to create user-defined data types. It serves as a blueprint or template for creating objects. The syntax for defining a class in C++ is as follows:

```
class ClassName {
  // Data members (variables)
  // Member functions (methods)
  // Access specifiers
};
```

Example:

```
class Circle {
public:
  // Data members
  double radius;
  // Member function
  double calculateArea() {
    return 3.14159 * radius * radius;
  }
};
```

Why Use C++ Classes?

C++ classes offer numerous benefits in software development:

1. Abstraction and Encapsulation: Classes allow us to abstract real-world entities and model them in our programs. By encapsulating related data and behaviors within a class, we can hide the implementation details and provide a clean interface for interacting with objects. This enhances code maintainability and reduces complexity.
2. Code Reusability: With classes, we can create objects that can be reused in different parts of our program or in other projects. This saves development time and effort, as we can leverage existing class definitions to create new objects with desired properties and behaviors.
3. Modularity and Organization: Classes promote modularity by encapsulating related functionality within a single unit. This improves code organization, readability, and ease of maintenance. Changes made to one class do not affect other parts of the program, ensuring a higher degree of code separation and reducing the risk of errors.
4. Data Integrity and Security: By encapsulating data within a class, we can control its access and modification. Class members can be defined as private or protected, restricting direct access from outside the class. This ensures data integrity and enhances security by preventing unintended modifications.
5. Inheritance and Polymorphism: C++ classes support inheritance, allowing the creation of derived classes that inherit properties and behaviors from a base class. Inheritance promotes code reuse and facilitates the creation of specialized classes. Polymorphism, another key feature of OOP, enables objects of different classes to be treated uniformly, simplifying code design and promoting flexibility.

5.2 Class Structure

The structure of a class defines its members, including data members and member functions. Data members are variables that hold data associated with objects of the class, while member functions are the actions or operations that can be performed on the objects. The structure of a class can be organized as follows:

```
class ClassName {
private:
  // Private data members

public:
  // Public data members

private:
  // Private member functions

public:
  // Public member functions
};
```

Example:

```
class Rectangle {
private:
  // Private data members
  double length;
  double width;

public:
  // Public member functions
  void setDimensions(double len, double wid) {
    length = len;
    width = wid;
  }

  double calculateArea() {
    return length * width;
  }
};
```

5.3 Class Objects.

A class object, also known as an instance, is a specific occurrence or realization of a class. It represents a unique entity with its own set of data and behaviors defined by the class. Creating objects allows us to work with individual instances and manipulate their properties and methods.

Syntax for Creating a Class Object:

To create a class object in C++, you follow this syntax:

```
ClassName objectName; // Declaration of a class object
```

Once the object is created, you can access its data members and methods using the dot (.) operator:

```
objectName.dataMember; // Accessing a data member of the object
objectName.methodName(); // Calling a method of the object
```

By creating class objects, you can work with specific instances of a class, manipulate their data, and invoke their methods to perform actions based on the defined behavior of the class.

Example 1: Creating a Class Object for a Car

```
// Class definition for a Car
class Car {
public:
   string  brand;
   string model;
   int year;

   void startEngine() {
      cout << "The " << brand << " " << model << " engine is starting..." << endl;
   }
};
```

```
// Creating a Car object
Car myCar;  // Declaration of a Car object

// Assigning values to the object's data members
myCar.brand = "Toyota";
myCar.model = "Camry";
myCar.year = 2022;

// Accessing the object's data members and methods
cout << "My car is a " << myCar.brand << " " << myCar.model << " from " << myCar.year << "."
<< endl;
myCar.startEngine();
```

In this example, we define a class called Car with data members (brand, model, year) and a member function (startEngine). We then create a Car object called myCar. We assign values to its data members and access them using the dot (.) operator. Finally, we call the startEngine method on the myCar object.

Example 2: Creating a Class Object for a Bank Account

```
// Class definition for a BankAccount
class BankAccount {
public:
  string accountNumber;
  string accountHolder;
  double balance;

  void deposit(double amount) {
    balance += amount;
    cout << "Deposit of $" << amount << " successful. New balance: $" << balance << endl;
  }

  void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            cout << "Withdrawal of $" << amount << " successful. New balance: $" << balance <<
endl;
        } else {
            cout << "Insufficient funds. Cannot withdraw $" << amount << endl;
        }
    }
};

// Creating a BankAccount object
BankAccount myAccount;  // Declaration of a BankAccount object

// Assigning values to the object's data members
myAccount.accountNumber = "123456789";
myAccount.accountHolder = "John Doe";
myAccount.balance = 1000.0;

// Accessing the object's data members and methods
cout << "Account number: " << myAccount.accountNumber << endl;
cout << "Account holder: " << myAccount.accountHolder << endl;
cout << "Balance: $" << myAccount.balance << endl;
myAccount.deposit(500.0);
myAccount.withdraw(200.0);
```

In this example, we define a class called BankAccount with data members (accountNumber, accountHolder, balance) and member functions (deposit, withdraw). We create a BankAccount object called myAccount and assign values to its data members. We then access these members and call the deposit and withdraw methods on the myAccount object.

5.4 Class scope

Class scope refers to the visibility and accessibility of class members (data members and member functions) within different parts of a program. In C++, class scope is controlled by access specifiers: private, public, and protected. These specifiers determine the accessibility of class members from outside the class and in derived classes. In other words, the scope of a class refers to the visibility or accessibility of its members (data members and member functions) from various parts of the program. C++ provides three access specifiers to control the scope:

- private: Members declared as private are accessible only within the class itself. They are not visible outside the class or in derived classes.
- public: Members declared as public are accessible from anywhere in the program, including outside the class.
- protected: Members declared as protected are accessible within the class itself and in derived classes but not from outside the class.

Let's explore class scope in more detail with examples:

Example 1: Class Scope with Private Access Specifier

```
class MyClass {

private:

  int privateData;


public:

  void setPrivateData(int value) {

    privateData = value;

  }


  void printPrivateData() {
        cout << "Private data: " << privateData << endl;
    }
};


int main() {
    MyClass myObject;
    myObject.setPrivateData(42);
    myObject.printPrivateData();


    // Accessing privateData directly will result in a compilation error
    // myObject.privateData = 123;  // Compilation error


    return 0;
}
```

In this example, the MyClass has a private data member privateData and two member functions, setPrivateData and printPrivateData. The privateData member is accessible only within the class itself. In the main function, we create an object of MyClass called myObject and use the setPrivateData method to assign a value to privateData. However, attempting to access privateData directly from outside the class, as shown in the commented line, will result in a compilation error.

Example 2: Class Scope with Public Access Specifier

```
class MyClass {
public:
    int publicData;
    void printPublicData() {
        cout << "Public data: " << publicData << endl;
    }
};


int main() {
    MyClass myObject;
    myObject.publicData = 100;
    myObject.printPublicData();


    return 0;
}
```

In this example, the MyClass has a public data member publicData and a member function printPublicData. The publicData member is accessible from anywhere in the program, including outside the class. In the main function, we create an object of MyClass called myObject and directly assign a value to publicData. We can also call the printPublicData method to print the value of publicData.

By using access specifiers, class scope allows you to control the visibility and accessibility of class members. Here's a summary of the three access specifiers and their implications:

- private: Members declared as private are accessible only within the class itself. They are not visible outside the class or in derived classes. Private members are typically used for internal implementation details that should not be accessed or modified directly from outside the class.
- public: Members declared as public are accessible from anywhere in the program, including outside the class. Public members form the interface of

the class, defining the operations that can be performed on class objects. They can be accessed and modified directly by code outside the class.

- protected: Members declared as protected are accessible within the class itself and in derived classes but not from outside the class. Protected members are often used when implementing inheritance, allowing derived classes to access and modify the protected members of their base class.

5.5 this Pointer

In object-oriented programming, a class serves as a blueprint for creating objects. Each object created from a class has its own set of member variables and member functions. The "this" pointer is a special pointer available within the member functions of a class that points to the current object being operated upon.

To understand it better, let's consider an example:

```
#include <iostream>

class MyClass {
private:
  int value;

public:
  MyClass(int value) {
    this->value = value;
  }

  void printValue() {
    std::cout << "Value: " << this->value << std::endl;
  }

  void setValue(int value) {
    this->value = value;
    }
};

int main() {
    MyClass obj1(42);
    MyClass obj2(87);

    obj1.printValue();  // Output: Value: 42
    obj2.printValue();  // Output: Value: 87

    obj1.setValue(99);
    obj1.printValue();  // Output: Value: 99

    return 0;
}
```

In this example, we have a class named "MyClass" with a member variable called "value". The constructor initializes this variable using the input value passed to it. The class also has two member functions: "printValue()" and "setValue()".

Now, let's focus on the usage of the "this" pointer within the class:

- In the constructor: When creating objects, the constructor is called to initialize the member variables. In this case, we use the "this" pointer to assign the input value to the member variable "value". The line this->value = value; ensures that the correct object's "value" variable is updated.
- In the member function "printValue()": Here, we use the "this" pointer to access the member variable "value" and print its value. By using this->value, we explicitly refer to the "value" variable of the current object.
- In the member function "setValue()": Similarly, the "this" pointer is utilized to access the member variable "value" and update it with the provided value.

In the "main" function, we create two objects of the "MyClass" class: obj1 and obj2. We then call the "printValue()" function on each object, which prints the respective values of their "value" member variables.

Later, we update the "value" of obj1 using the "setValue()" function and call "printValue()" again to confirm the change.

The "this" pointer is essential in distinguishing between member variables of different objects of the same class. It allows us to access and modify the member variables of the current object within its member functions.

5.6 Friend to a class

A friend function of a class is a function that is declared outside the scope of the class, but it has the privilege of accessing all the private and protected members of that class. Unlike member functions, friend functions are not part of the class itself.

A friend function can be a regular function, a function template, a member function of another class, or even an entire class or class template. When a function or class is declared as a friend of another class, it is granted special access to the private and protected members of that class.

Let's explore some examples to illustrate the usage of friend functions:

Example 1: Friend Function

```
#include <iostream>

class MyClass {

private:

    int privateData;
public:
    MyClass(int data) {
        privateData = data;
    }

    friend void displayPrivateData(const MyClass& obj);
};

void displayPrivateData(const MyClass& obj) {
    std::cout << "Private Data: " << obj.privateData << std::endl;
}

int main() {
    MyClass obj(42);
    displayPrivateData(obj); // Output: Private Data: 42
    return 0;
}
```

In this example, we have a class named "MyClass" with a private member variable called "privateData". The function displayPrivateData() is declared as a friend of the "MyClass" class. This allows the function to access the private member variable of any instance of the class. Outside of the class, we define the function displayPrivateData() which takes an object of the "MyClass" class as a parameter. Within this function, we can access and display the private member variable privateData of the passed object.

Example 2: Friend Class

```
#include <iostream>
class MyClass {
private:
    int privateData;

public:
    MyClass(int data) {
        privateData = data;
    }

    friend class FriendClass;
};

class FriendClass {
public:
    void displayPrivateData(const MyClass& obj) {
        std::cout << "Private Data: " << obj.privateData << std::endl;
    }
};
int main() {
    MyClass obj(42);
    FriendClass friendObj;
    friendObj.displayPrivateData(obj); // Output: Private Data: 42

    return 0;

}
```

In this example, we have a class named "MyClass" with a private member variable called "privateData". The entire class "FriendClass" is declared as a friend of the "MyClass" class. As a result, the "FriendClass" has access to all the private and protected members of the "MyClass" class.

Within the "FriendClass", we define the member function displayPrivateData() which takes an object of the "MyClass" class as a parameter. Inside this member function, we can access and display the private member variable privateData of the passed object.

In the main() function, we create an object obj of the "MyClass" class and an object friendObj of the "FriendClass". We then call the member function displayPrivateData() of the friendObj, passing the obj as an argument. Since the FriendClass is a friend of the MyClass, it can access and display the private member variable privateData.

These examples demonstrate how friend functions and friend classes can be used to grant special access to private and protected members of a class to external functions or classes. Friend functions and classes can be useful in certain scenarios where controlled access to private data is required while still maintaining encapsulation and data hiding principles.

5.7 Static Class Members

A static class member is a member (variable or method) that belongs to the class itself rather than an instance of the class. It is shared by all instances of the class and can be accessed without creating an object of the class.

The "static" keyword is used to define variables that have static memory allocation. When a variable is declared as static, its memory is allocated only once and remains unchanged throughout the program's execution. Once a static variable is declared, its memory allocation is fixed and cannot be modified during runtime. This means that the variable will retain its value across different function calls or instances of a class. The static variable essentially persists in memory throughout the entire program's execution.

Declaration and Definition:

Static members are declared within the class definition but are defined outside the class using the scope resolution operator ::. The static keyword is used to declare a member as static. For example:

```
class MyClass {
public:
  static int myStaticVariable; // declaration of static variable
  static void myStaticFunction(); // declaration of static function
};


int MyClass::myStaticVariable = 0; // definition of static variable


void MyClass::myStaticFunction() { // definition of static function
  // function body
}
```

Accessing Static Members:

Static members can be accessed using the class name followed by the scope resolution operator ::. They can also be accessed through an object of the class, but it is more common to access them directly through the class name. For example:

```
int value = MyClass::myStaticVariable; // accessing static variable
```

```
MyClass::myStaticFunction(); // calling static function
```

Shared among Instances:

Unlike regular members of a class, static members are shared among all instances of the class. This means that any change made to a static member is reflected in all instances of the class. Each instance does not have its own copy of the static member.

Initialization:

Static data members need to be explicitly initialized outside the class. This is typically done in a source file (.cpp) using the scope resolution operator ::. Static member variables are initialized only once, before any objects of the class are created. For example:

```
int MyClass::myStaticVariable = 42; // initializing static variable
```

Memory Allocation:

Static members are stored in a shared memory location that is associated with the class rather than with individual objects. They are allocated memory when the program starts and exist until the program terminates. The memory for static members is not deallocated when objects of the class are destroyed.

Usage Scenarios:

- Counters and statistics: Static variables are often used to maintain counts or statistics across multiple instances of a class. For example, a class representing employees in a company can have a static variable to keep track of the total number of employees created.
- Utility functions: Static member functions can serve as utility functions that perform common operations related to the class but don't require any instance-specific data. They can be called directly using the class name without creating objects.

Example

```
class MathUtils {
```

```
public:
  static double square(double x) { // static member function
    return x * x;
  }
};
int main() {
  double number = 5.0;
  double result = MathUtils::square(number); // accessing static member function
  cout << "Square of " << number << " is " << result << endl;
  return 0;
}
```

In this example, the MathUtils class provides a utility function square that calculates the square of a given number. Since the function doesn't rely on any instance data, it can be made static and accessed using the class name.

- Shared resources: Static members can be used to represent shared resources, such as a database connection or a configuration setting, that need to be accessed by multiple instances of a class.

It's important to note that static members can only access other static members of the class. They cannot access non-static members directly because non-static members are associated with individual instances of the class and require an object to access them.

5.8 Constant member functions

A "constant member function" is a kind of function that can't change any values inside its own class. You make a function "constant" by putting the word "const" when you write the function.

So, a constant function can look at the data in its class, but it can't change any of that data. It's like having "read-only" permission.

Just like we can make functions "constant", we can also make objects "constant". A "constant object" can't be changed once it's created. And because it can't be changed, it can only use "constant" functions, which also can't change anything.
A "constant" member function in C++ can be defined in a multitude of ways, notably in three primary methods. The application of the "const" keyword varies according to the specific context and structure of the function. Below are the detailed explanations of each method:

1. **Declaration of the function within the class:**

The basic prototype or declaration of the function involves the use of the "const" keyword following the function's name and its parentheses. Here, "return_type" represents the data type of the value that the function will return. The syntax for this declaration is as follows:

return_type function_name() const;

In the following example, **getName** and **getAge** are declared as const member functions within the Person class.

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(std::string n, int a) : name(n), age(a) {}

  std::string getName() const;
  int getAge() const;
};
```

2. **Definition of the function within the class declaration:**

If the function is defined directly within the class declaration, the function's body is included immediately following its prototype. In this case, the "const" keyword is placed after the function's parentheses, just before the opening brace of the function's body. The structure is as follows:

```
return_type function_name() const
{
   //function body
}
```

Here, the getWidth and getHeight functions are defined directly within the Rectangle class declaration.

```
class Rectangle {
private:
  int width, height;
public:
  Rectangle(int w, int h) : width(w), height(h) {}

  int getWidth() const {
    return width;
  }

  int getHeight() const {
    return height;
  }
};
```

3. **Definition of the function outside the class:**

For functions defined outside the class declaration, the syntax includes the class name, followed by the scope resolution operator "::", and then the function name.

The "const" keyword is positioned after the function's parentheses. The syntax looks like this:

```
return_type class_name::function_name() const
{
      //function body
}
```

In this case, the getName and getAge functions are defined outside the Person class declaration.

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(std::string n, int a) : name(n), age(a) {}

  std::string getName() const;
  int getAge() const;
};

std::string Person::getName() const {
  return name;
}

int Person::getAge() const {
  return age;
}
```

These definitions essentially help to reinforce the principle of "const-correctness" in C++, preserving the immutability of the objects, and ensuring that a function doesn't modify the data members of its class unintentionally. A constant member function can access and return the data members of the class, but it cannot modify them, guaranteeing that the state of an object remains unchanged following a function call.

5.9 Constructors and Destructors

Constructors and destructors are specialized member functions that exhibit distinct behaviour related to the lifecycle of objects within a class. They are automatically invoked during the instantiation and termination of class objects, serving crucial roles in memory management and program stability.

Constructor:

A constructor is a unique member function within a class, bearing the same name as the class itself. It is automatically invoked whenever an object of that class is instantiated. The constructor's main purpose is to initialize the object's state by setting initial values for its member variables. Notably, constructors lack a return type - not even void.

The general syntax for a constructor in C++ is as follows:

```
class ClassName{
private:
 // Private members

public:

 // Constructor declaration
 ClassName(parameters)
 {
  // Constructor body
 }

};
```

In this syntax, the class is named ClassName, and the constructor also has the same name. A constructor can accept any number of parameters as needed. It's important to note that constructors do not have a return type or return any value.

For instance, consider the following example:

```
class Rectangle {
private:
   int width, height;
public:
   Rectangle(int w, int h) : width(w), height(h) {}  // Constructor
};
```

Here, Rectangle(int w, int h) operates as a constructor for the Rectangle class. Upon creation of a new Rectangle object, this constructor initializes the width and height variables as per the provided values:

```
Rectangle rect(10, 20);  // Instantiates a Rectangle object with width = 10 and height = 20
```

A constructor is automatically invoked by the compiler whenever an object of that class is created. Its principal role is twofold: memory allocation and data member initialization.

When an object is instantiated, the constructor function is responsible for allocating the necessary memory for that object, ensuring that it has the space to store its data and function attributes.

Regarding initialization, a constructor sets the values of the class's data members. This can be accomplished in two ways. By default, if no explicit initialization is provided by the user, the constructor assigns each data member its default value based on the data type (e.g., integers to zero, pointers to null, etc.). Alternatively, if the user provides specific values when creating an object, the constructor uses these values for initialization instead.

For example, consider the following class:

```
class Rectangle {
private:
   int length;
```

```
    int width;
public:
    Rectangle(int l, int w) : length(l), width(w) {}
};
```

Here, the constructor Rectangle(int l, int w) assigns the user-specified values l and w to the data members length and width, respectively. Thus, when we instantiate a Rectangle object as follows:

```
Rectangle rect(10, 20);  // Creates a Rectangle object with length = 10 and width = 20
```

The constructor allocates the necessary memory for the rect object and initializes its length and width data members to the values 10 and 20, respectively.

Here are some key points to note about constructors:

1. Access specifiers: A constructor can be declared as public, protected, or private. Usually, constructors are declared as public because they are used to create objects and initialize class data members.

2. Inheritance: A derived class can call the base class's constructor if it is not declared as private.

3. Virtual: A constructor cannot be declared as virtual in C++.

There are four main types of constructors in C++:

1. Default constructor: This is a constructor that does not take any parameters. It initializes data members of the class with predefined values.

```
class ClassName{
public:

 ClassName()
 {
   // Constructor body
 }

};
```

2. Parameterized constructor: This is a constructor that takes parameters and uses these parameters to initialize the class's data members.

```
class ClassName{
public:

  ClassName(int a, string b)
  {
   // Constructor body
  }

};
```

3. Copy Constructor: This type of constructor is used to create a new object as a copy of an existing object. It takes a single argument, which is a reference to an object of the same class.

```
class ClassName{
public:

  ClassName(const ClassName& obj)
  {
   // Constructor body
  }

};
```

4. Dynamic Constructor: In this type of constructor, memory for the object is allocated at runtime using the new operator.

```
class ClassName{
public:
 ClassName(int size)
 {
  // Allocate memory dynamically
  int* arr = new int[size];
 }

};
```

Destructor:

Contrarily, a destructor is another specialized member function of a class, invoked when an object of the class is about to be destroyed, typically upon going out of scope or when explicitly deleted. A destructor shares the class's name but is prefixed with a tilde (~). It cannot return a value, and it does not accept any parameters. Destructors are generally utilized to release resources that the object may have acquired during its lifecycle.

```
class Rectangle {
private:
  int *width, *height;
public:
  Rectangle(int w, int h) {
    width = new int;
    height = new int;
    *width = w;
    *height = h;
  }

  ~Rectangle() { // Destructor
    delete width;
    delete height;
  }
};
```

In this case, ~Rectangle() functions as the destructor for the Rectangle class. This destructor is called when a Rectangle object is about to be destroyed. Specifically, it frees the memory allocated to the width and height in the constructor, ensuring no memory leaks occur.

```
{
  Rectangle rect(10, 20);  // Instantiates a Rectangle object with width = 10 and height = 20
} // At this point, the rect object goes out of scope, and its destructor is invoked
```

It is essential to note that if a constructor or destructor is not explicitly provided within the class definition, C++ automatically generates a default version. These default constructors and destructors accept no parameters and perform no actions.

A destructor signifies a function that is inversed to the constructor. A constructor functions to instantiate an object of a class, while a destructor's role is to handle the deallocation of the memory assigned to the object when it is no longer needed or at the end of its lifetime. The compiler activates the destructor when the lifecycle of an object comes to an end; this may occur at the completion of a program, at the end of a function where the local objects of the function become unreachable, or in any analogous circumstances.

Contrary to constructors, destructors are not open to overloading. They neither accept arguments nor do they have a return type or value.

Structural Syntax of a Destructor

The following outlines the fundamental syntax of a destructor in C++:

```
class ClassName{
 private:
 // Private members

 public:
 // Declaration of destructor
 ~ClassName()
 {
   // Body of destructor
 }
};
```

In the above configuration, the class and the destructor share the same nomenclature, appended with a tilde (~) in the destructor. Both return type and return value are absent in the destructor. The destructor can be classified as a public or a private member based on necessity.

Pertinent Details About the Destructor

The destructor represents the terminal member function invoked for an object and is called into action by the compiler. If the programmer fails to define a destructor, the compiler takes the initiative to define it. Regardless of its positioning within the class, the compiler will trigger the destructor since it oversees the call. For enhanced readability, the destructor is often declared at the end of the class, denoting its role as the last function to be invoked.

The roles of constructors and destructors are symmetrical but opposite; constructors are activated at the creation of an object and allocate memory to it, while destructors are triggered at the object's destruction and deallocate the memory.

How Constructors and Destructors Operate During Object Creation and Destruction

The constructor represents the first class member function the compiler calls when creating an object, while the destructor serves as the last function called for an object. The compiler will autonomously define default constructors and destructors for a class object if the user hasn't explicitly declared them.

The following piece of code elucidates how constructors and destructors function:

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

class ClassName{
 // Declaration of private class data members
private:
 int a, b;

public:

 // Declaration of constructor
 ClassName(int varA, int varB)
 {
   cout<<"Constructor is invoked"<<endl;
   a = varA;
   b = varB;

   cout<<"Value of a: "<<a<<endl;
   cout<<"Value of b: "<<b<<endl;
   cout<<endl;
 }

 // Declaration of destructor
 ~ClassName()
 {
   cout<<"Destructor is invoked"<<endl;
   cout<<"Value of a: "<<a<<endl;
   cout<<"Value of b: "<<b<<endl;
 }

};
int main()
{
  // Creation of class object using parameterized constructor
  ClassName object(5,6);
```

For Vivekananda [illegible] Registrar

```
    return 0;
}
```

In this code, a class with a constructor and destructor is defined. A class object is instantiated in the main function using the parameterized constructor, and when the program concludes, the compiler triggers the destructor, printing the values of the variables.

Both constructors and destructors are unique member functions of a class, established by the C++ compiler or defined by the user. A constructor is automatically called by the compiler to allocate memory to a class object and initialize class data members during object creation. On the other hand, a destructor is summoned when an object is slated for destruction; its principal role is the deallocation of the object's memory.

The constructor and destructor share the same name as the class, with the destructor having an additional tilde (~) operator prefix. Both constructors and destructors can be assigned as public, private, or protected, although it is generally preferred to classify the constructor as public. While constructors may have parameters, destructors don't accept any.

5.10 Dynamic Creation and destruction of objects

C++ allows dynamic creation and destruction of objects, which can be extremely useful for programs that require flexible memory management. Unlike static objects that are created at compile-time, dynamic objects are created and destroyed at runtime.

The 'new' operator is used for dynamically creating objects, while 'delete' operator is used for destroying these objects.

Creation of Dynamic Objects:

The 'new' operator allocates memory for the object in the heap, constructs the object, and returns a pointer to it. Here is the basic syntax for creating a dynamic object:

```
className* pointerName = new className;
```

If the class has parameterized constructors, you can pass arguments like this:

```
className* pointerName = new className(argument(s));
```

Here, 'className' is the name of the class, and 'pointerName' is the pointer that will hold the address of the dynamic object.

Destruction of Dynamic Objects:

When a dynamic object is no longer needed, it should be destroyed to free up memory. This is done using the 'delete' operator, as shown below:

```
delete pointerName;
```

After the 'delete' operator is called, the destructor for the object is called and the memory allocated to the object is deallocated.

Here is an example of creating and destroying a dynamic object:

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

class Test {
  int data;
public:
  Test(int value) {
    data = value;
    cout << "Constructor called, data = " << data << endl;
  }
  ~Test() {
    cout << "Destructor called, data = " << data << endl;
  }
};
```

```
int main() {
    Test* ptr = new Test(5); // Constructor is called
    delete ptr; // Destructor is called
    return 0;
}
```

In this example, a dynamic object of the class 'Test' is created using the 'new' operator, and destroyed using the 'delete' operator.

Keep in mind that it is the programmer's responsibility to destroy dynamic objects when they are no longer needed. Failing to do so can result in a memory leak, where memory that is no longer being used is not returned to the system, reducing the amount of memory available for other objects.

5.11 Summary

In this ebook, we explored the concept of classes and data abstraction in C++. We began by discussing the definition and structure of a class. A class serves as a blueprint for creating objects and defines the properties and behaviors they possess.

We delved into the components of a class, including member variables and member functions. Member variables hold the data associated with an object, while member functions define the actions that can be performed on the object.vNext, we explored the concept of data abstraction, which involves hiding the internal implementation details of a class and exposing only the necessary information to the outside world. This allows for encapsulation and data security. We then examined the scope of a class, which determines the visibility and accessibility of its members. The private and protected access specifiers restrict direct access to class members, while the public access specifier allows access from outside the class.

To access the members of a class within member functions, we learned about the "this" pointer. The "this" pointer refers to the current object and provides a means to access its member variables and member functions. Furthermore, we studied the concept of friend functions and friend classes. Friend functions are external functions that are granted access to the private and protected members of a class. Similarly, friend classes have access to all the members of a class, enabling them to manipulate private data. By leveraging friend functions and friend classes, we can selectively expose certain functionalities and provide controlled access to private data while maintaining encapsulation and data abstraction principles.and maintainable C++ programs.

We've covered several fundamental aspects of classes in C++ programming in this section, which are instrumental in developing efficient, readable, and robust code.

We learned about static class members, which, unlike typical members of a class, are shared among all instances and do not require an object for access. This unique attribute makes static members ideal for maintaining counters, creating utility functions, or representing shared resources across class instances.Constant member functions offer a means to preserve data integrity by prohibiting modifications to class data, effectively providing a "read-only" access to the class's data members.

Constructors and destructors are special class member functions automatically called upon the creation and destruction of class objects. They play a crucial role in managing an object's lifecycle, with constructors facilitating initialization and destructors ensuring cleanup. Understanding their roles and functionalities helps write safer and more efficient code. Lastly, we delved into the dynamic creation and destruction of objects, a feature that provides a level of control over memory management far beyond static objects. By dynamically allocating and deallocating memory at runtime, we can create flexible and memory-efficient applications. Taken together, these elements form a significant part of the object-oriented programming paradigm, and their judicious use can greatly enhance the effectiveness of your programming efforts.

5.12 Keywords

- Class: A blueprint for creating objects that defines their properties and behaviors.
- Data Abstraction: The process of hiding internal implementation details and exposing only necessary information to the outside world.
- Member Variables: Variables defined within a class that hold data associated with objects of that class.
- Member Functions: Functions defined within a class that operate on objects of that class.
- Scope: The visibility and accessibility of class members, determined by access specifiers (private, protected, public).
- Private: An access specifier that restricts direct access to class members from outside the class.
- Protected: An access specifier that allows access to class members from derived classes and within the same class.
- Public: An access specifier that allows unrestricted access to class members from outside the class.
- this Pointer: A special pointer in C++ that refers to the current object and enables access to its member variables and member functions.
- Friend Functions: External functions that are granted access to the private and protected members of a class.
- Friend Classes: Classes that have access to all the members of another class, including private data and member functions.
- **Static Class Members:** Class members that are shared by all objects of the class rather than being specific to an individual object.
- **Constant Member Functions:** A member function that guarantees not to modify the object on which it is called.
- **Constructors:** Special functions in the class that are called automatically when an object of the class is created.
- **Destructors:** Special functions in the class that are called automatically when an object of the class is destroyed or goes out of scope.

- **Dynamic Creation and Destruction of Objects:** The process of creating and destroying objects at runtime using 'new' and 'delete' operators, respectively, which helps in managing memory more efficiently.
- **Memory Management:** The process by which a program controls and coordinates computer memory, assigning portions to variables, data structures, functions, etc., and freeing it for reuse when no longer needed.
- **Data Integrity:** The accuracy, consistency, and reliability of data stored in a database, disk, file, or an object in object-oriented programming.
- **New and Delete Operators:** In C++, 'new' operator is used to allocate memory at runtime to variables or objects, and 'delete' operator is used to free that memory when no longer needed.
- **Instance:** A specific realization of any object, being a concrete, individual object that you can manipulate in a program.

5.13 Review Questions

1. What is the purpose of the "this" pointer in C++? How does it enable access to member variables and member functions within a class?
2. Explain the concept of data abstraction in C++. How does it contribute to encapsulation and information hiding?
3. Describe the difference between private, protected, and public access specifiers in a class. How do they impact the visibility and accessibility of class members?
4. What is the role of friend functions in C++? How do they provide special access to private and protected members of a class?
5. How are friend classes different from friend functions? Explain their significance in terms of accessing and manipulating private data within a class.
6. What is a static class member in C++? How is it different from a regular class member?
7. How are static class members declared and accessed in a C++ program?
8. Explain the concept of constant member functions in C++. When would you use a constant member function?

9. What restrictions are placed on constant member functions with regards to modifying class data?
10. What are constructors and destructors in C++? How do they contribute to object lifecycle management?
11. How does the constructor differ from the destructor in terms of functionality?
12. Differentiate between default, parameterized, copy, and dynamic constructors in C++. Provide an example of each.
13. What is the purpose of the 'new' and 'delete' operators in C++? How are they used in the dynamic creation and destruction of objects?
14. How does dynamic memory allocation contribute to the efficiency of a C++ program?
15. Describe a real-world scenario where the use of static class members, constant member functions, and dynamic creation and destruction of objects would be beneficial.
16.

5.14 References

1. Shiffman, D. (2016). Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (2nd ed.). Morgan Kaufmann.

Unit – 6 Data abstraction, Overloading and Inheritance

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Understand data abstraction's role in simplifying code.
- Implement information hiding techniques for secure code.
- Understand the concept of function overloading in C++.
- Learn how to declare and define overloaded functions with different parameter lists.
- Understand the Concept of Inheritance: Explain the fundamental concept of Inheritance in object-oriented programming and how it enables code reuse and hierarchy creation.
- Identify the Different Forms of Inheritance: Recognize and differentiate between various forms of Inheritance, including single, multiple, multilevel, hierarchical, and hybrid Inheritance.
- Define Base and Derived Classes: Define and implement base (superclass) and derived (subclass) classes in code, understanding their roles in inheritance hierarchies.
- Demonstrate Code Reusability: Showcase the practical application of Inheritance by reusing attributes and methods from a base class in derived classes.

Introduction

C++ is an object-oriented programming language that offers a wealth of features to simplify code development and enhance its efficiency. Among these features are several key aspects related to classes, which serve as the core of object-oriented programming. The principles of abstraction and information hiding stand as cornerstones for constructing robust, maintainable, and adaptable software systems. Abstraction, as a fundamental concept, empowers developers to manage complexity by distilling intricate real-world entities into simplified models that capture their essential characteristics. Through this ebook, we aim to learn the core principles of function and operator overloading, inheritance and abstraction and information hiding, elucidating their significance in the development of robust software systems.

Function overloading is a fundamental concept in C++. It allows you to define multiple functions within the same scope with the same name but different parameter lists. The distinguishing factor is the number or types of arguments each function expects. This enables you to create a set of related functions that perform similar operations but on different data types or with different argument combinations.

For example, you can have multiple "add" functions, one for integers, another for doubles, and so on. This makes the code more intuitive and readable as the function names remain consistent, yet the behavior adapts based on the provided arguments

Operator Overloading:

Operator overloading extends the flexibility of C++ by allowing you to redefine the behaviors of operators like +, -, *, and more, for custom user-defined data types (classes or structures). This means you can make your custom objects work seamlessly with these operators, just like built-in types. It enhances code expressiveness and reduces the learning curve for users of your custom classes.

For instance, you can define how the + operator should behave when applied to instances of your custom class, such as for adding complex numbers or matrices.

Inheritance is a foundational concept in object-oriented programming (OOP) that lies at the heart of creating organized and reusable code. It forms the basis for defining relationships between classes, allowing for code sharing and creating class hierarchies. It is a mechanism that enables a new class (known as the derived or subclass) to inherit properties and behaviors from an existing class (known as the base or superclass). This Inheritance allows developers to model real-world relationships, promote code reusability, and create a structured class hierarchy.

The Significance of Inheritance:

Inheritance is not merely a programming construct; it is a powerful tool that has significant implications for software development:

- Code Reusability: By inheriting attributes and methods from a base class, derived classes can reuse existing code, reducing redundancy and promoting efficiency in software development.

- Hierarchy and Organization: Inheritance enables the creation of class hierarchies, mirroring the relationships between objects in the real world. This hierarchy makes code more intuitive and easier to manage.
- Extensibility: Derived classes can add new attributes and methods or modify existing ones, allowing for customization while preserving the core functionality defined in the base class.
- Maintenance: Inheritance simplifies code maintenance because changes in the base class automatically apply to all derived classes, ensuring consistency.

Defining a Class Hierarchy:At the core of Inheritance lies the concept of a class hierarchy. In this structure, base classes serve as the foundation upon which derived classes are built. The hierarchy reflects relationships between objects in the problem domain. For instance, a base class "Animal" might have derived classes like "Dog," "Cat," and "Bird."

Different Forms of Inheritance:Inheritance can take various forms, each with distinct characteristics and use cases. These forms include single Inheritance, Multiple Inheritance, multilevel Inheritance, Hierarchical Inheritance, and hybrid Inheritance. Understanding these forms is essential for making informed design choices.

Base and Derived Classes:Base classes, also known as superclasses, provide a blueprint for derived classes, often called subclasses. Base classes define common attributes and methods shared by derived classes. Derived classes inherit these attributes and methods while having the flexibility to extend or override them.

6.1 Data Abstraction

Data abstraction is a foundational concept in computer science and software engineering that involves simplifying complex systems by focusing on essential aspects while concealing unnecessary details. It revolves around creating a clear distinction between the "what" of an object's behavior and properties from the "how" it is implemented. This approach enhances code readability, reusability, and maintainability, leading to more efficient software development.

In data abstraction, complex entities are represented in a simplified manner, emphasizing their core characteristics and behaviors. This is often achieved through the use of abstract classes and interfaces in object-oriented programming languages like C++, Java, and Python.

By defining abstract classes and interfaces, developers establish a blueprint that outlines the essential methods and properties an object should have, without specifying how those methods are implemented.

There are two main components of data abstraction:

1. **Data Encapsulation:** This involves bundling the data (attributes) and the methods (functions or procedures) that operate on the data into a single unit, known as a class. The class provides a well-defined interface through which the outside world can interact with the data and functionality of the object, while keeping the internal implementation details hidden. This is also known as encapsulation and allows for better control over data access and manipulation.
2. **Data Abstraction:** This refers to the process of defining a class in a way that focuses on the essential properties and behaviors of the object while abstracting away the less relevant details. For example, if you're designing a class to represent a "Car," you might focus on attributes like "make," "model," and methods like "start_engine" and "accelerate," while ignoring low-level details about how the engine works internally.

Here are the key points that capture the essence of data abstraction:

- **Essential Characteristics:** Data abstraction focuses on capturing the fundamental properties and behaviors of objects, distilling them down to their most important aspects.
- **Interface and Implementation:** The interface of an object comprises the set of methods that define how external entities can interact with it. The implementation includes the actual code that carries out the functionality described in the interface.
- **Hiding Complexity:** Data abstraction shields users from the intricate internal workings of an object. Users interact with objects through their interfaces, without needing to understand the underlying complexities.
- **Encapsulation:** Encapsulation, which is closely related to data abstraction, involves bundling data and methods together into a cohesive unit, often referred to as a class. Access to the internal details of the class is controlled through access modifiers like "public," "private," and "protected."
- **Code Reusability:** Abstracted classes and interfaces can be reused across different parts of a program or in entirely different programs, saving development time and promoting a modular code structure.

- **Maintenance and Evolution:** Abstraction makes it easier to modify or extend a system over time. Changes can be made to the internal implementation of a class without affecting its external interface, as long as the interface remains consistent.
- **Real-World Analogy:** Think of data abstraction as similar to driving a car. You don't need to understand the intricate mechanics of the engine to operate the vehicle; you interact with the steering wheel, pedals, and dashboard, which represent the abstracted interface for controlling the car.

Data abstraction provides a powerful framework for managing complexity in software design. By creating clear boundaries between the public interface and private implementation, it empowers developers to build scalable, maintainable, and adaptable software systems.

Benefits of Data Abstraction:

- Code Reusability: Abstracted classes can be reused in different contexts, reducing redundant code and promoting a DRY (Don't Repeat Yourself) approach.
- Enhanced Security: Hiding implementation details prevents unintended access and modification, improving the security of your code.
- Code Maintenance: Changes to the internal implementation of a class don't affect external code as long as the interface remains consistent.
- Readability and Understandability: Abstracted code is easier to read and understand, especially for other developers who may interact with or maintain the codebase.
- Scalability: Data abstraction supports the creation of scalable software systems by promoting modular design.

In the context of C++, data abstraction is a fundamental concept within the realm of object-oriented programming. It involves using classes and access specifiers to achieve encapsulation and create well-defined interfaces for interacting with objects while hiding their internal implementation details. Let's explore data abstraction in C++ in more detail:

1. Classes and Objects: In C++, a class is a blueprint for creating objects. It defines both the data (attributes) and the methods (member functions) that operate on that data. Objects are instances of a class.
2. Access Specifiers: C++ provides three access specifiers that control the visibility and accessibility of class members:

- public: Members with this specifier are accessible from outside the class.

Members declared as public within a class are accessible from outside the class, meaning that they can be accessed and manipulated by code that exists outside the class definition.

```
class Rectangle {
public:
  double length;
  double width;
  double calculateArea() {
    return length * width;
  }
};
```

In this example, the length and width attributes are declared as public. This allows code outside the Rectangle class to directly access and modify these attributes:

```
int main() {
  Rectangle myRect;
myRect.length = 5.0;
myRect.width = 3.0;

  double area = myRect.calculateArea();
  // 'area' is calculated using the public attributes
    return 0;
}
```

- private: Members with this specifier are only accessible within the class itself.

  Members declared as private within a class are only accessible within the class itself. They cannot be accessed or modified directly from outside the class.

```
class BankAccount {
private:
    double balance;

public:
BankAccount(double initialBalance) : balance(initialBalance) {}

    void deposit(double amount) {
        balance += amount;
    }

    double getBalance() {
        return balance;
    }
};
```

In this example, the balance attribute is declared as private. It cannot be accessed directly from outside the class:

```
int main() {
BankAccountmyAccount(1000.0);

    // This line would result in a compilation error
    // myAccount.balance = 1500.0;

myAccount.deposit(500.0);

    double accountBalance = myAccount.getBalance();
    // 'accountBalance' is obtained through the public method


    return 0;
}
```

- protected: Members with this specifier are accessible within the class and its derived classes.

  Members declared as protected within a class are accessible within the class itself and its derived classes (classes that inherit from this class).

```
class Vehicle {
protected:
    int speed;

public:
    Vehicle(int initialSpeed) : speed(initialSpeed) {}

    void accelerate(int amount) {
        speed += amount;
    }

    int getSpeed() {
        return speed;
    }
};


class Car : public Vehicle {
public:
    Car(int initialSpeed) : Vehicle(initialSpeed) {}

    void increaseSpeed(int amount) {
        speed += amount;  // 'speed' is accessible due to protected specifier
    }
};
```

In this example, the speed attribute is declared as protected in the Vehicle class. It is accessible within the derived Car class:

```
int main() {
  Car myCar(50);
myCar.accelerate(20);  // 'accelerate' is a public method of Vehicle


  // This line would work because 'speed' is protected in Vehicle
myCar.increaseSpeed(30);
 int carSpeed = myCar.getSpeed();
  return 0;
}
```

3. Encapsulation: Encapsulation involves bundling the data and methods that operate on the data into a single unit (class). The use of access specifiers ensures that the internal implementation details of a class are hidden from the outside world, and access is restricted to the defined interfaces.
4. Abstraction: Abstraction in C++ is achieved by designing classes in a way that exposes only the relevant information and behaviors, while abstracting away the implementation details. Users of a class need to know how to use its public interface without needing to know how it's implemented internally.

6.2 Function Overloading

Function overloading is a programming concept that allows you to define multiple functions with the same name within a programming language or class, but with different parameter lists. The choice of which function to call is based on the arguments provided during the function invocation. Function overloading is primarily used in statically typed languages like C++, Java, and C#.

The Need for Function Overloading

The primary reasons for using function overloading are as follows:

1 Readability and Clarity: Overloaded functions allow you to use descriptive and intuitive names for operations that perform similar tasks. This makes your code more readable and self-explanatory.

2 Code Reusability: You can reuse the same function name for different variations of a task, reducing the need to create new function names for each variant. This promotes code reusability.

3 Consistency: It promotes a consistent naming convention for related functions, making your codebase more organized and maintainable.

Syntax and Declaration of Overloaded Functions

To declare overloaded functions, you provide multiple function definitions with the same name in the same scope or within the same class, but with different parameter lists. The syntax varies depending on the programming language, but here is a basic representation in pseudo-code:

```
int calculate(int a, int b);

double calculate(double a, double b);
```

Here, calculate is overloaded with two different parameter lists, one for integers and another for doubles.

Resolving Overloaded Functions

When you call an overloaded function, the compiler or interpreter determines which function to execute based on the arguments provided. This process is known as function resolution or function overloading resolution. It involves analyzing the number and types of arguments in the function call and selecting the most specific function that matches the argument list.

The selection process depends on the programming language's rules for function resolution. Typically, it follows these principles:

- **Exact Match:** If an exact match exists between the function's parameter list and the provided arguments, that function is called.

- **Type Promotion:** If no exact match exists, the compiler may attempt to promote the argument types to match one of the overloaded functions.
- **Type Conversion:** If promotion doesn't work, the compiler may perform type conversions if they are available and unambiguous.
- **Ambiguity Resolution:** In some cases, if the compiler cannot decide between multiple matching overloaded functions due to ambiguity, it will result in a compilation error.

Function overloading resolution ensures that the most appropriate function is called based on the arguments provided, providing flexibility while maintaining type safety.

Overloading with Different Parameter Types

Function overloading allows you to define multiple functions with the same name but different parameter types. This is particularly useful when you want to perform similar operations on different data types. Here's an example in C++:

```
#include <iostream>
// Function to add two integers
int add(int a, int b) {
return a + b;
}
// Function to concatenate two strings
std::string add(std::string a, std::string b) {
return a + b;
}
int main() {
int result1 = add(2, 3); // Calls the first add function with integers.
std::string result2 = add("Hello, ", "world!"); // Calls the second add function with strings.

std::cout<< result1 << std::endl; // Output: 5

std::cout<< result2 << std::endl; // Output: Hello, world!

return 0;

}
```

In this example, we have two add functions—one that takes two integer parameters and another that takes two string parameters. The compiler selects the appropriate function to call based on the argument types.

Overloading with Different Number of Parameters

You can also overload functions based on the number of parameters they accept. Here's an example in C++:

```
#include <iostream>

// Function to calculate the sum of two integers

int add(int a, int b) {

  return a + b;

}

// Function to calculate the sum of three integers

int add(int a, int b, int c) {

  return a + b + c;

}

int main() {

  int result2 = add(2, 3); // Calls the first add function with two integers.
```

```
    int result3 = add(2, 3, 4); // Calls the second add function with three
integers.

    std::cout<< result2 << std::endl; // Output: 5

    std::cout<< result3 << std::endl; // Output: 9

    return 0;

}
```

In this example, we have two add functions—one that takes two integer parameters and another that takes three integer parameters. The compiler selects the appropriate function to call based on the number of arguments provided.

Overloading with different numbers of parameters is helpful when you want to provide flexibility in the number of arguments a function can accept, allowing you to perform similar operations with varying levels of detail or precision.

Function overloading with different parameter types and numbers of parameters enhances the versatility and readability of your code by enabling you to use the same function name for related operations on different data types and argument counts.

Overloading Constructors

In object-oriented programming, constructors can also be overloaded. Overloading constructors allows you to create objects of a class with different initializations, providing flexibility in how objects are created. Each overloaded constructor can have a different set of parameters or parameter types.

Best Practices and Use Cases

Best Practices for Constructor Overloading

- **Default Constructor:** Provide a default constructor with no parameters to ensure that objects can be created without mandatory initialization.
- **Progressive Initialization:** Overloaded constructors should provide progressively more detailed initializations. Parameters can be added to set different attributes or properties of an object.

- **Avoid Ambiguity:** Be cautious when overloading constructors to avoid ambiguity in object creation. Ensure that each constructor's parameter list is distinct enough that the compiler can distinguish between them.

Use Cases for Constructor Overloading

- **Default Values:** Overloaded constructors can be used to provide default values for object attributes.
- **Custom Initialization:** Constructors with different parameter sets can allow objects to be initialized in various ways to accommodate different scenarios.
- **Data Validation:** Constructors can perform data validation or transformation during object creation.

Function Overloading Examples

C++ Constructor Overloading

```
#include <iostream>

class Person {

public:

  // Default constructor

  Person() : name("Unknown"), age(0) {}

  // Constructor with name parameter

  Person(const std::string& n) : name(n), age(0) {}

  // Constructor with name and age parameters

  Person(const std::string& n, int a) : name(n), age(a) {}

  // Getter methods for name and age (not shown here)

private:

  std::string name;

  int age;

};
```

```
int main() {
   Person person1; // Creates a person with default values.
   Person person2("Alice"); // Creates a person with a specified name.
   Person person3("Bob", 30); // Creates a person with a name and age.
   return 0;
}
```

Overloaded constructors allow you to create objects with different initializations, making your classes more versatile and accommodating various use cases.

6.3 Operator Overloading

The Operator overloading is a feature in many programming languages that allows you to define custom behaviors for operators when used with user-defined types (classes or structs). This enables you to extend the capabilities of operators beyond their predefined functions and make your code more intuitive and expressive.

Operators That Can Be Overloaded

The following operators can typically be overloaded in many programming languages:

- **Arithmetic Operators:** +, -, *, /, %, etc.
- **Comparison Operators:** ==, !=, <, >, <=, >=, etc.
- **Assignment Operators:** =, +=, -=, *=, /=, etc.
- **Increment/Decrement Operators:** ++, --, etc.
- **Unary Operators:** +, -, !, ~, etc.
- **Subscript Operator (for array-like access):** []
- **Function Call Operator (for making objects callable):** ()
- **Member Access Operators (for accessing class or struct members):** ->, .
- **Bitwise Operators:**&, |, ^, <<, >>, etc.

The ability to overload these operators can greatly enhance the readability and usability of your code, especially when working with custom data types.

Syntax and Implementation of Overloaded Operators

The syntax for overloading operators varies depending on the programming language. However, the general idea is to define a special method or function within a class or struct that provides the custom behavior for the operator.

Here's a general example in C++:

```
class MyClass {

public:

  int value;

  // Overload the + operator

MyClass operator+(constMyClass& other) {

MyClass result;

result.value = this->value + other.value;

    return result;

  }

};
```

In this example, the + operator is overloaded to add two MyClass objects together.

Overloading Unary Operators

Unary operators operate on a single operand. To overload a unary operator, you typically define a member function or a free function that takes no arguments (besides the implicit this pointer or the object being operated upon) and returns the result of the operation.

Here's an example in C++ overloading the unary - operator:

```
class Complex {

public:
    double real;

    double imag;

    Complex operator-() {

        Complex result;

result.real = -this->real;

result.imag = -this->imag;

        return result;

    }

};
```

Overloading Binary Operators

Binary operators operate on two operands. To overload a binary operator, you usually define a member function or a free function that takes one argument (in addition to the implicit this pointer or the object being operated upon) and returns the result of the operation.

Here's an example in C++ overloading the binary + operator:

```
class Vector {

public:

    double x;

    double y;

    Vector operator+(const Vector& other) {

        Vector result;

result.x = this->x + other.x;

result.y = this->y + other.y;
```

```
            return result;

        }

    };
```

Friend Functions in Operator Overloading

In some cases, when overloading operators, you may need access to private members of the class or struct. To achieve this, you can use friend functions, which are non-member functions that are granted access to the private members of a class or struct. Friend functions can be useful when overloading operators that require access to private data.

Operator Overloading Examples

Overloading + Operator

```
Complex operator+(const Complex& a, const Complex& b) {

   Complex result;

result.real = a.real + b.real;

result.imag = a.imag + b.imag;

return result;

}
```

Overloading [] Operator (Subscript Operator)

```
class MyArray {

public:

   int data[10];

   int& operator[](int index) {

      return data[index];
```

```
    }

};
```

Overloading << Operator (for Custom Output)

```
class Person {

public:

    std::string name;

    int age;

    friend std::ostream& operator<<(std::ostream&os, const Person& person) {

os<< "Name: " << person.name << ", Age: " <<person.age;

        return os;

    }

};
```

Best Practices and Use Cases

Best Practices for Operator Overloading

- Overload operators only when it makes sense in the context of your custom class or struct. The behavior should be intuitive and follow common expectations.
- Avoid overloading operators in ways that might lead to confusion or unexpected behavior.
- Overload operators consistently and ensure that their behavior is documented for users of your class or struct.

Use Cases for Operator Overloading

- Mathematical Operations: Overload operators to perform mathematical operations on custom numeric types or vector/matrix classes.

- Custom Container Types: Overload the [] operator for custom container classes, allowing array-like access.
- Complex Numbers: Overload arithmetic operators for custom complex number classes.
- String Concatenation: Overload the + operator for custom string classes.
- Custom Output Formatting: Overload << and >> operators for custom output formatting and input parsing.

6.4 Inheritance

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows you to create new classes based on existing classes. It's a mechanism by which a new class, often called the subclass or derived class, can inherit properties and behaviors (i.e., fields and methods) from an existing class, known as the superclass or base class. Inheritance models an "is-a" relationship, where a subclass is a specialized superclass version.

Importance of Inheritance

1. **Code Reusability:** Inheritance promotes code reusability by allowing you to reuse existing code from a superclass in a subclass. Instead of duplicating code, you can extend and modify the superclass's behavior as needed. This reduces redundancy and makes your code more efficient and maintainable.
2. **Abstraction:** Inheritance helps in creating an abstraction hierarchy. You can define common attributes and methods in a superclass, and the subclasses can provide more specific implementations or override those methods to suit their needs. This abstraction allows for a more organized and structured codebase.
3. **Polymorphism:** Inheritance is closely related to polymorphism, another key concept in OOP. Polymorphism allows objects of different classes to be treated as objects of a common superclass. This flexibility enables you to write more generic and flexible code that can work with a variety of related objects.
4. **Simplifies Maintenance:** When you need to make changes or add features to your software, having a well-designed inheritance hierarchy can make the task easier. You can make updates in the superclass, and those changes will automatically apply to all its subclasses. This reduces the risk of introducing errors and simplifies maintenance.

5. **Elimination of Redundancy:** Inheritance eliminates redundancy in your codebase. You can define common attributes and methods once in a superclass, and multiple subclasses can inherit and use them. This saves time and effort by not having to rewrite the same code in multiple places.
6. **Consistency:** Inheritance promotes a consistent design across related classes. If you make changes or bug fixes in the superclass, those changes propagate to all subclasses, ensuring they stay consistent with the updated logic.
7. **Scalability:** As your software evolves, you can easily extend the inheritance hierarchy by creating new subclasses without altering existing code. This scalability allows your codebase to grow and adapt to new requirements with minimal disruption.
8. **Improved Debugging:** Inheritance simplifies debugging because issues in shared functionality are often centralized in the superclass. Fixing a problem in one place (the superclass) can resolve issues in multiple places (subclasses).

6.5 Types of Inheritance

Inheritance allows you to create a new class (called a derived or child class) from an existing class (called a base or parent class). The derived class inherits properties and behaviors (i.e., data members and member functions) from the base class. This concept promotes code reusability and the creation of hierarchies of classes. In C++, inheritance is one of the four pillars of OOP, alongside encapsulation, polymorphism, and abstraction.

There are several types of inheritance in C++:

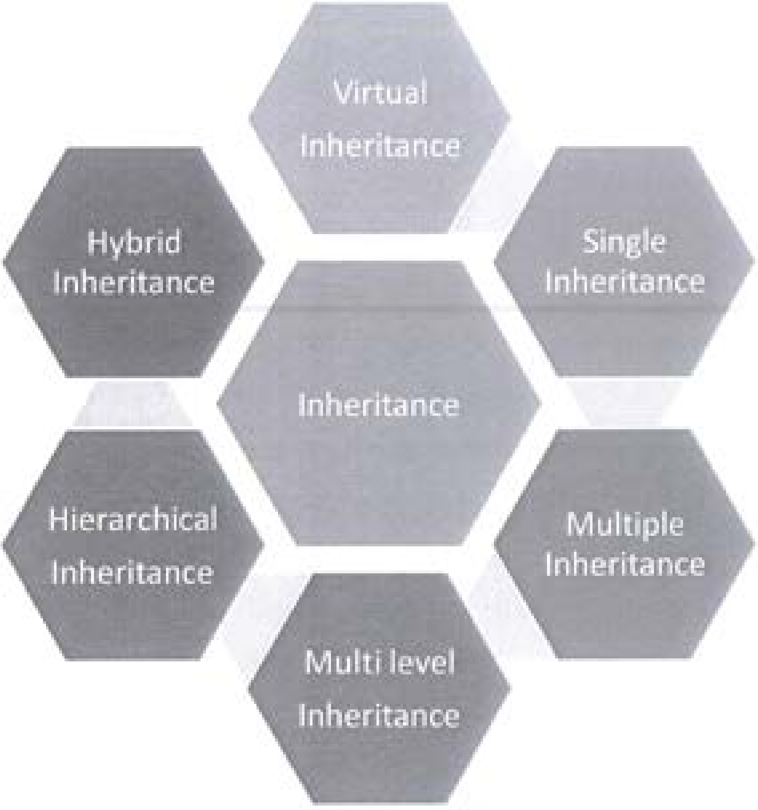


Fig – Types of Inheritance

Single Inheritance:

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

This is the simplest form of inheritance.

Example:

```
class Animal {
  // ...
};
class Dog : public Animal {
  // ...
};
```

Example

```
#include <iostream>

// Base class

class Animal {

public:

   void eat() {

      std::cout<< "Animal is eating." << std::endl;

   }

};

// Derived class inheriting from Animal

class Dog : public Animal {

public:

   void bark() {

      std::cout<< "Dog is barking." << std::endl;

   }

};

int main() {

   // Create an object of the derived class

   Dog myDog;

   // Access methods from both base and derived classes

myDog.eat();  // Call the base class method

myDog.bark(); // Call the derived class method

   return 0;

}
```

In this example:

- We define a base class Animal with a method eat.
- We create a derived class Dog using single inheritance, where Dog inherits from Animal.
- In the main function, we create an object myDog of the Dog class.
- We can access the eat method from the base class and the bark method from the derived class using the myDog object.

Multiple Inheritance:

Multiple inheritance allows a derived class to inherit from more than one base class.

It's a powerful but complex feature and can lead to the "diamond problem" where ambiguities arise.

Example:

```
        class Shape {
         // ...
        };
        class Color {
         // ...
        };
class ColoredShape : public Shape, public Color {
         // ...
        };
```

Example

```
#include <iostream>
// First base class
class Shape {
public:
    Shape(int sides) : numSides(sides) {}
    void ShowSides() {
        std::cout<< "Number of sides: " <<numSides<< std::endl;
    }
private:
    int numSides;
};
// Second base class
class Color {
public:
Color(const std::string&clr) : color(clr) {}

    void ShowColor() {
        std::cout<< "Color: " <<color<< std::endl;
    }
private:
    std::string color;
};
// Derived class inheriting from both Shape and Color
class ColoredShape: public Shape, public Color {
public:
ColoredShape(int sides, const std::string&clr) : Shape(sides), Color(clr) {}
    void ShowInfo() {
ShowSides();  // Accessing the ShowSides() method from Shape
ShowColor();  // Accessing the ShowColor() method from Color
```

```
    }
};
int main() {
ColoredShape square(4, "Red");
square.ShowInfo(); // This calls methods from both base classes
    return 0;
}
```

In this example:

- We have two base classes, Shape and Color, each with its own properties and methods.
- The ColoredShape class is derived from both Shape and Color using multiple inheritance. It inherits properties and methods from both base classes.
- In the main function, we create an instance of ColoredShape called square and initialize it with the number of sides and color.
- We then call the ShowInfo method of square, which in turn calls methods from both base classes (ShowSides from Shape and ShowColor from Color).

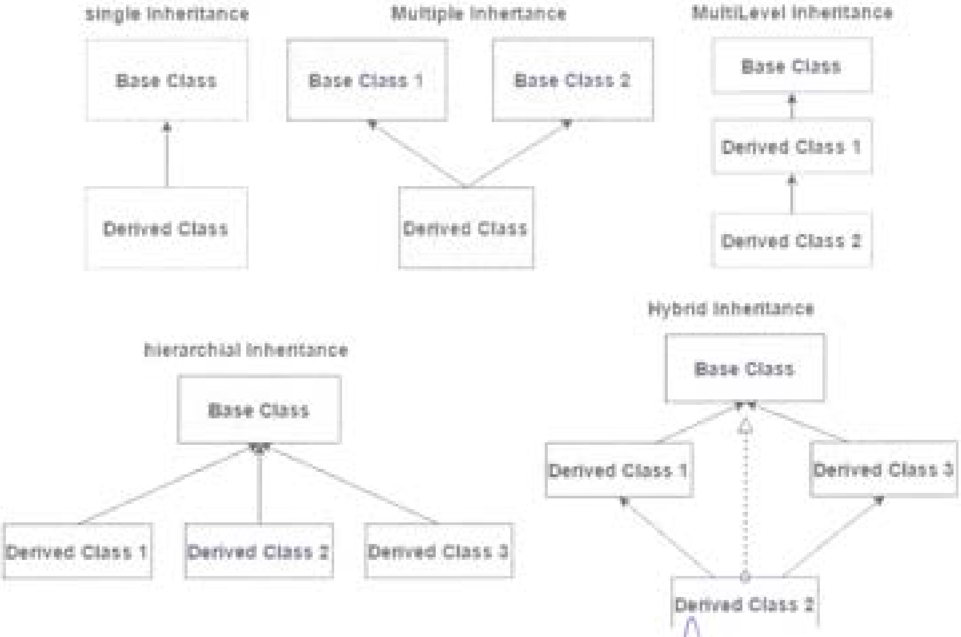


Fig – Types of Inheritance

Multilevel Inheritance:

- Multilevel inheritance involves a chain of inheritance with multiple levels of classes.
- A derived class inherits from a base class, and another class can inherit from the derived class, creating a hierarchy.

Example:

```
class Vehicle {
  // ...
};
class Car : public Vehicle {
  // ...
};
class Sedan : public Car {
  // ...
};
```

Example

```
#include <iostream>
// Base class
class Animal {
public:
    void eat() {
        std::cout<< "Animal is eating." << std::endl;
    }
};

// First level derived class
```

```
class Mammal : public Animal {
public:
  void run() {
      std::cout<< "Mammal is running." << std::endl;
  }
};
// Second level derived class
class Dog : public Mammal {
public:
  void bark() {
      std::cout<< "Dog is barking." << std::endl;
  }
};
int main() {
  Dog myDog;
      // Using functions from different levels of inheritance
myDog.eat();  // Inherited from Animal
myDog.run();  // Inherited from Mammal
myDog.bark(); // Defined in Dog
      return 0;
}
```

In this example:

- Animal is the base class, which has a method eat.
- Mammal is a derived class that inherits from Animal and adds a method run.
- Dog is another derived class that inherits from Mammal and adds a method bark.

- The main function demonstrates the use of these classes. myDog is an instance of the Dog class, and it can access methods from all levels of the inheritance hierarchy: eat from Animal, run from Mammal, and bark from Dog.

Hierarchical Inheritance:

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- This creates a hierarchy of classes with a common ancestor.

Example:

```
class Animal {
  // ...
};
class Dog : public Animal {
  // ...
};
class Cat : public Animal {
  // ...
};
```

Example

```
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
   void eat() {
cout<< "Animal is eating." <<endl;
   }
```

```
};
// Derived class 1
class Dog : public Animal {
public:
   void bark() {
cout<< "Dog is barking." <<endl;
   }
};
// Derived class 2
class Cat : public Animal {
public:
   void meow() {
cout<< "Cat is meowing." <<endl;
   }
};
int main() {
   Dog myDog;
   Cat myCat;
   // Using methods from the base class
myDog.eat();
myCat.eat();

   // Using methods from the derived classes
myDog.bark();
myCat.meow();
```

```
    return 0;

  }
```

In this example:

- Animal is the base class, which has a common behavior eat.
- Dog and Cat are derived classes that inherit from Animal.
- Both Dog and Cat have their own unique behaviors, bark and meow, respectively.
- In the main function, objects of Dog and Cat are created and used to demonstrate the inheritance hierarchy.

Hybrid Inheritance:

- Hybrid inheritance is a combination of two or more types of inheritance mentioned above.
- It often involves complex class hierarchies.

Example:

```
class A {
  // ...
};
class B : public A {
  // ...
};
class C : public A {
  // ...
};
class D : public B, public C {
  // ...

};
```

Example

```
#include <iostream>

using namespace std;

// Base class

class Animal {

public:
   void eat() {
cout<< "Animal is eating" <<endl;
   }
};
// First level of derived classes
class Mammal : public Animal {
public:
   void giveBirth() {
cout<< "Mammal gives birth" <<endl;
   }
};
class Bird : public Animal {
public:
   void layEggs() {
cout<< "Bird lays eggs" <<endl;
   }
};
// Second level of derived classes
class Bat : public Mammal, public Bird {
public:
   void fly() {
cout<< "Bat can fly" <<endl;
   }
};
int main() {
    Bat bat;

    // Accessing methods from different levels of the hierarchy

  bat.eat();       // Accessing Animal's method

  bat.giveBirth(); // Accessing Mammal's method

  bat.layEggs();  // Accessing Bird's method

  bat.fly();       // Accessing Bat's method

    return 0;

  }
```

In this example:

- Animal is the base class with a method eat.
- Mammal and Bird are derived from Animal, representing two different types of animals.
- Bat is a class derived from both Mammal and Bird, demonstrating hybrid inheritance.
- The Bat class inherits methods from both Mammal and Bird, as well as from the common base class Animal.
- In the main function, an instance of Bat is created and methods from all levels of the hierarchy are accessed.

Virtual Inheritance:

- Virtual inheritance is used to avoid ambiguity and the diamond problem in multiple inheritance.
- It ensures that only one copy of a base class is inherited, even if it appears multiple times in the inheritance hierarchy.
- Achieved by using the virtual keyword when inheriting.

Example:

```
class Animal {

 // ...

};
```

```
class Mammal : virtual public Animal {
  // ...
};
class Bird : virtual public Animal {
  // ...
};
class Bat : public Mammal, public Bird {
  // ...
};
```

Each type of inheritance has its own advantages and use cases. The choice of which type to use depends on the design of your software and the relationships between classes in your program.

6.6 Class Hierarchy

The Class Hierarchy is a crucial concept in Object-Oriented Programming (OOP) that represents the relationships between classes in a structured and organized manner. It involves the creation of a hierarchy of classes, where each class is organized into a parent-child relationship. These relationships are established through Inheritance, with a base class (also known as a superclass or parent class) and one or more derived classes (also known as subclasses or child classes).

Understanding Class Hierarchies:

- **Base Class (Superclass):** The base class is the top-level class in the hierarchy. It defines common attributes and behaviors that are shared among all classes in the hierarchy. It serves as a template for the derived classes. The base class typically contains general methods and properties that are relevant to all subclasses.
- **Derived Classes (Subclasses):** Derived classes are the classes that inherit properties and behaviors from the base class. They extend or specialize the base class's functionality by adding their unique attributes or methods. Derived classes can also override or modify the behavior of inherited methods.

- **Inheritance:** Inheritance is the mechanism that establishes the class hierarchy. It allows derived classes to inherit the base class's characteristics (fields and methods). This promotes code reuse, abstraction, and polymorphism within the hierarchy.
- **"is-a" Relationship:** Class hierarchies often represent an "is-a" relationship. This means that a derived class is a more specialized version of the base class. For example, if you have a base class called "Vehicle," derived classes like "Car" and "Bicycle" are more specific types of vehicles.

The Role of Base and Derived Classes:

Base Class (Superclass):

- Defines common attributes and methods.
- Serves as a blueprint for derived classes.
- Typically contains more general or abstract implementations.

Derived Classes (Subclasses):

- Inherit attributes and methods from the base class.
- Extend or customize the functionality of the base class.
- Can override inherited methods to provide specific implementations.
- Add their own unique attributes and methods.

6.7Base Classes (Superclasses)

Generic Base Classes, also known as Superclasses or Parent Classes, serve as the foundation for class hierarchies in Object-Oriented Programming (OOP). They define common attributes and methods that are shared among multiple derived classes. Here, we'll discuss how to define base classes, what common attributes and methods they may contain, and the concept of access modifiers in OOP.

Defining Base Classes:

To define a base class in code, you simply create a class with the attributes and methods representing the shared characteristics and behaviors of the objects it will model. Base classes often serve as templates for derived classes, so they provide a blueprint for what derived classes should inherit and override.

Common Attributes and Methods:

- **Attributes:** Base classes may have attributes representing common properties of the objects they model. For example, if you're defining a base class for shapes, common attributes might include color, size, or position.
- **Methods:** Base classes may contain methods that provide common behaviors applicable to all derived classes. For example, a base class for geometric shapes could have a method called the area that calculates the area of the shape. Derived classes would then inherit and possibly override this method to suit their specific shapes.

Access Modifiers (public, protected, private):

Access modifiers control the visibility and accessibility of class members (attributes and methods). They determine which parts of your code can access these members. Common access modifiers in many programming languages include:

- **Public:** Members marked as public are accessible from anywhere, both within and outside the class. This is the default access level in many programming languages.
- **Protected:** Members marked as protected are accessible within the class and its derived classes. They are not accessible from outside the class hierarchy.
- **Private:** Members marked as private are only accessible within the class itself. They are not accessible from derived classes or external code.

Example Base Class Implementation:

.Please note that this is a simplified example to illustrate the concept of implementing base classes and derived classes in C. In real-world scenarios, object-oriented features are better supported and more convenient in languages like C++

6.8 Derived Classes (Subclasses)

Templates In C++, creating derived classes (also known as subclasses) involves defining a new class that inherits attributes and behaviors from an existing class, known as the base class or parent class. Here's a step-by-step guide on how to create derived classes in C++:

1. **Define the Base Class (Superclass):**

First, you need to define the base class, which contains the common attributes and methods that you want to share among the derived classes. Here's an example of a simple base class called Shape:

```
        class Shape {
        protected: // Access specifier for subclass access
          std::string color;
        public:
          Shape(const std::string& _color) : color(_color) {}
          virtual double area() const = 0; // Pure virtual function
        };
```

In this example, we've defined a base class Shape with a member variable color and a pure virtual function area(). The area() function is marked as pure virtual (= 0), indicating that it must be overridden by any derived class.

2. Create the Derived Class:

To create a derived class, use the class keyword followed by the derived class name, a colon :, and the access specifier (public, protected, or private). Then, specify the base class from which you want to inherit:

```
        class Circle : public Shape {
        private:
          double radius;
public:
          Circle(const std::string& _color, double _radius) : Shape(_color),
        radius(_radius) {}
          double area() const override {
            return 3.14159 * radius * radius;
          }
```

For Vivekananda Global University, Jaipur

Registrar

```
};
```

In this example, we've defined a derived class Circle that inherits from the Shape base class. We also have a member variable radius specific to the Circle class. We override the area() function to provide a specific implementation for circles.

3. Create Instances of Derived Classes:

You can create instances of derived classes just like any other class:

```
int main() {
    Circle myCircle("Red", 5.0);
    double circleArea = myCircle.area();
    std::cout<< "The area of the circle is: " <<circleArea<< std::endl;
    return 0;
}
```

Here, we create an instance of the Circle class and use its member function area() to calculate the area of the circle.

4. Accessing Base Class Members:

In the derived class, you can access the base class members using the :: operator or through the constructor initialization list. In the Circle class constructor, we use the base class constructor to initialize the color member:

```
Circle(const std::string& _color, double _radius) : Shape(_color), radius(_radius) {}
```

This way, you initialize the base class part of the object.

By following these steps, you can create derived classes in C++ that inherit attributes and behaviors from a base class, allowing you to model more specialized objects while reusing common functionality. Additionally, polymorphism is supported through the use of virtual functions, enabling dynamic dispatch and runtime method resolution when dealing with objects of the base and derived classes.

6.9 Summary

This chapter delves into the importance of encapsulating data and functionality within a class to achieve data abstraction and information hiding, thereby promoting better code organization and maintenance. Here's a summary of the key points covered in the chapter:

Data Abstraction: Data abstraction is the process of simplifying complex reality by modeling classes that represent real-world entities. It involves focusing on the essential characteristics of an object while hiding unnecessary details. By defining classes that abstract data and operations, programmers can create higher-level structures that are easier to work with.

Access Control and Encapsulation:C++ offers access control modifiers like public, private, and protected. By designating certain members as private, the class can control which parts of its implementation are exposed to the outside world. Encapsulation ensures that the internal state and methods of a class are shielded from direct external access, promoting data integrity and reducing potential sources of bugs.

Benefits of Information Hiding: Information hiding improves code maintainability and reduces dependencies between different parts of the codebase. Changes to the internal implementation of a class won't affect the external code that relies on its interface, as long as the interface remains consistent. This separation of concerns simplifies testing, debugging, and overall software evolution.

Use Cases of Data Abstraction and ADTs: Data abstraction and ADTs are particularly useful when dealing with complex systems, where multiple components need to interact while keeping their implementations separate. Common examples include handling file systems, databases, networking protocols, and more. efforts.

Function Overloading:

Function overloading allows you to define multiple functions with the same name but different parameter lists. This enhances code readability by providing a consistent interface for related functions that perform similar tasks on different data types or with different argument combinations.

Operator Overloading:

Operator overloading extends C++ by allowing you to redefine the behaviors of operators like +, -, *, etc., for user-defined data types. It makes custom objects work seamlessly with these operators, enhancing code expressiveness and reducing the learning curve for users of your classes.

Inheritance:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create new classes based on existing classes. It enables the creation of a hierarchy of classes, where derived classes inherit properties and behaviors from base (parent) classes. Inheritance promotes code reuse and establishes an "is-a" relationship between classes.

6.10 Keywords

- Data Abstraction: Simplifying complex reality by modeling classes that represent real-world entities while hiding unnecessary details.
- Encapsulation: Restricting direct access to a class's internal state and methods, promoting data integrity and reducing external dependencies.
- Information Hiding: Concealing the implementation details of a class while providing a well-defined interface for external interaction.
- Access Control: C++ modifiers like public, private, and protected that manage the visibility and accessibility of class members.
- Modularity: Breaking down a system into smaller, manageable components that can be developed and tested independently.
- Function Overloading: Function overloading is a feature in C++ that allows multiple functions in the same scope to have the same name but with different parameters.
- Operator Overloading: Operator overloading is a feature in C++ that allows you to redefine the behavior of operators, such as +, -, *, etc., for user-defined data types.
- Inheritance:Inheritance is a fundamental concept in object-oriented programming where a class (derived or child class) inherits properties and behaviors from another class (base or parent class).
- Base Class:A base class, also known as a parent class, is the class from which other classes inherit properties and behaviors.
- Derived Class:A derived class, also known as a child class, is a class that inherits properties and behaviors from a base class.
- Single Inheritance:Single inheritance is a form of inheritance in which a derived class inherits from a single base class.

- Multiple Inheritance:Multiple inheritance is a form of inheritance in which a derived class can inherit from multiple base classes.
- Multilevel Inheritance:Multilevel inheritance is a form of inheritance where a derived class is derived from another derived class, creating a hierarchy of classes.
- Hierarchical Inheritance:Hierarchical inheritance is a form of inheritance in which multiple derived classes inherit from a single base class, creating a branching hierarchy.
- Hybrid Inheritance:Hybrid inheritance is a combination of two or more forms of inheritance, often including multiple and multilevel inheritance.
- Method Overriding:Method overriding is a feature of inheritance where a derived class provides a specific implementation for a method already defined in the base class.
- Dynamic Binding:Dynamic binding, also known as late binding or runtime polymorphism, is the process of determining at runtime which version of a method to call based on the actual object's type.
- Virtual Function:A virtual function is a member function in a base class that can be overridden by derived classes, allowing dynamic binding.

6.11 Review Questions

1. Define data abstraction and explain its significance in programming. How does it contribute to code organization and simplification? Provide an example to illustrate your explanation.
2. Describe the principles of encapsulation and information hiding in object-oriented programming. Provide a scenario where encapsulation enhances code reliability and explain how it achieves this.
3. What is function overloading, and why is it used in C++?
4. Can you have two functions with the same name and the same parameter list in C++?
5. What is operator overloading, and why is it important in C++?
6. Name some operators that can be overloaded for user-defined types.
7. What is function overloading, and why is it used in C++?
8. Name some operators that can be overloaded for user-defined types.
9. How are function templates instantiated with specific data types?
10. Define class templates and their purpose in C++.

11. Explain how class templates are instantiated with specific data types.

6.12 References

1. Shiffman, D. (2016). Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (2nd ed.). Morgan Kaufmann
2. "C++ Primer" (5th Edition) by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo.
3. Stroustrup, Bjarne. "The C++ Programming Language." 4th ed., Addison-Wesley, 2013.
4. Lippman, Stanley B., Lajoie, Josée, and Moo, Barbara E. "C++ Primer." 5th ed., Addison-Wesley, 2012.
5. Meyers, Scott. "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14." 1st ed., O'Reilly Media, 2014.
6. Stroustrup, Bjarne. "Programming: Principles and Practice Using C++." 2nd ed., Addison-Wesley, 2014.
7. Williams, Anthony. "C++ Concurrency in Action." 2nd ed., Manning Publications, 2019.

C++ LAB

Lab Manual

Practical 1: Create a user defined function (any) and use it inside the program

Exercise 1: Write a C++ program that calculates the factorial of a given number using a user-defined function.

Exercise 2: Write a C++ program that calculates the area and perimeter of a rectangle using user-defined functions.

Practical 2: Implement "call by value" & "call by reference "function call techniques by using any user defined functions.

Exercise 3: Write a C++ program that swaps the values of two integers using the "call by value" mechanism.

Exercise 4: Write a C++ program that swaps the values of two integers using the "call by reference" mechanism.

Practical 3: Implement the working of classes and objects by using any real world object.

Exercise 5: Implement a Car class in C++ to represent a real-world car object.

Exercise 6: Create a C++ class named "Rectangle" to represent a rectangle object.

Practical 4: Create any user defined class using the concept of static data and member functions.

Exercise 7: Create a user-defined class called "Employee" that represents an employee in a company.

Exercise 8: Create a user-defined class called "MathUtils" that contains static member functions for performing mathematical operations.

Practical 5: Create a Class or program implementing the concept of passing and returning object to/from member functions.

Exercise 9: Passing and Returning Objects in Member Functions.

Practical 6: To implement polymorphism through function overloading (Area of different shapes).

Exercise 10: Write a C++ program to implement polymorphism through function overloading for calculating the area of different shapes.

Practical 7: Create a user defined type Complex and do all the Complex number arithmetic and also make use of operator overloading.

Exercise 11: Write a C++ program to implement a user-defined type Complex.

Practical 8: Implement single level inheritance by using Student and Marks class.

Exercise 12: Write a C++ program to implement single-level inheritance by using the Student and Marks classes.

Practical 9: Implement multilevel inheritance by using the Stack class.

Exercise 13: Create a multilevel inheritance hierarchy using the Stack class as the base class. The derived classes should be named Numeric Stack and String Stack.

Practical 10: Implement the concept of Abstract classes and virtual functions by using Shape, Rectangle and Triangle class.

Exercise 14: In this exercise, you will implement the concept of abstract classes and virtual functions by creating the Shape, Rectangle, and Triangle classes. The Shape class will be an abstract base class, while the Rectangle and Triangle classes will inherit from it.

Practical 1: Create a user defined function (any) and use it inside the program

A user-defined function in C++ is a function that is created and defined by the user to perform a specific task. It allows the programmer to break down a complex problem into smaller, manageable parts. User-defined functions enhance code reusability, readability, and modularity. The syntax for defining a user-defined function in C++ is as follows:

```
return_type function_name(parameters) {
    // Function body
    // Statements and calculations
    // Return statement (if applicable)
}
```

Following is a breakdown of each component:

return_type: This specifies the type of value that the function will return after executing its task. It can be any valid C++ data type or void if the function does not return any value.

function_name: This is the name given to the function. It should be unique and meaningful, describing the task the function performs.

parameters: These are optional and represent the input values that the function may require to perform its task. Parameters are enclosed in parentheses and separated by commas. Each parameter consists of a type followed by its name, which will be used within the function body to refer to the corresponding input values.

function_body: This is the block of code that defines the task performed by the function. It consists of statements, calculations, control structures, and other function calls.

return statement: If the function has a return type other than void, it must include a return statement that specifies the value to be returned. The return statement also terminates the function execution and transfers control back to the calling code.

To use a user-defined function, you need to call it from within your program. The function call includes the function name followed by parentheses, which may contain arguments (actual values) that will be passed to the function's parameters.

Exercise 1: Write a C++ program that calculates the factorial of a given number using a user-defined function.

The factorial of a number 'n' is defined as the product of all positive integers less than or equal to 'n'. The program should prompt the user to enter a positive integer and display the factorial of that number.

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

```
// Function to calculate the factorial of a number
int factorial(int n) {
    int fact = 1;
for (int i = 1; i <= n; i++) {
fact *= i;
    }
    return fact;
}
int main() {
    int num;
    cout << "Enter a positive integer: ";
    cin >> num;

    // Check if the entered number is positive
    if (num < 0) {
        cout << "Error: Invalid input. Please enter a positive integer." << endl;
        return 0;
    }
    // Call the factorial function and display the result
    int result = factorial(num);
    cout << "The factorial of " << num << " is: " << result << endl;
    return 0;
}
```

In this lab exercise, the program prompts the user to enter a positive integer. The program then checks if the entered number is positive. If it is, it calls the factorial function, which calculates the factorial of the given number using a loop. The calculated factorial is then displayed to the user. If the entered number is not positive, an error message is displayed.

Exercise 2:Write a C++ program that calculates the area and perimeter of a rectangle using user-defined functions.

The program should prompt the user to enter the length and width of the rectangle, and then display the calculated area and perimeter.

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

// Function to calculate the area of a rectangle
double calculateArea(double length, double width) {
    return length * width;
}

// Function to calculate the perimeter of a rectangle
double calculatePerimeter(double length, double width) {
    return 2 * (length + width);
}

int main() {
    double length, width;
    cout << "Enter the length of the rectangle: ";
    cin >> length;
    cout << "Enter the width of the rectangle: ";
    cin >> width;

    // Check if the entered values are valid
    if (length <= 0 || width <= 0) {
        cout << "Error: Invalid input. Length and width should be positive values." << endl;
        return 0;
    }

    // Call the functions to calculate the area and perimeter
    double area = calculateArea(length, width);
    double perimeter = calculatePerimeter(length, width);

    cout << "Area of the rectangle: " << area << endl;
    cout << "Perimeter of the rectangle: " << perimeter << endl;

    return 0;
}
```

In this lab exercise, the program prompts the user to enter the length and width of a rectangle. The program then checks if the entered values are valid (i.e., positive). If either of the values is not positive, an error message is displayed. Otherwise, the program calls the calculateArea and calculatePerimeter functions to compute the area and perimeter of the rectangle, respectively. The calculated values are then displayed to the user.

Practical 2 : Implement "call by value" & "call by reference " function call techniques by using any user defined functions.

"Call by value" and "call by reference" are two different mechanisms used to pass arguments to functions in programming languages like C++. Let's explore each mechanism in more detail:

Call by Value:

In the "call by value" mechanism, the function receives a copy of the argument's value. Any modifications made to the parameter within the function do not affect the original argument in the calling code.

example to illustrate "call by value":

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

```
void increment(int num) {
   num++;
   cout << "Inside function - Value of num: " << num << endl;
}

int main() {
   int num = 5;
   cout << "Before function call - Value of num: " << num << endl;
   increment(num);
   cout << "After function call - Value of num: " << num << endl;

   return 0;
}
```

In this example, the function increment takes an integer argument num by value. Inside the function, the num parameter is incremented. However, the increment operation only modifies the local copy of num within the function, and the original num variable in the main function remains unchanged. Therefore, the output of the program will be:

```
Before function call - Value of num: 5

Inside function - Value of num: 6

After function call - Value of num: 5
```

Call by Reference:

In the "call by reference" mechanism, the function receives a reference to the argument. Any modifications made to the parameter within the function directly affect the original argument in the calling code.

example to illustrate "call by reference":

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

void increment(int& num) {
   num++;
   cout << "Inside function - Value of num: " << num << endl;
}

int main() {
   int num = 5;
   cout << "Before function call - Value of num: " << num << endl;
   increment(num);
   cout << "After function call - Value of num: " << num << endl;
```

```
    return 0;
}
```

In this example, the function increment takes an integer argument num by reference using the & symbol. Any changes made to num within the function directly modify the original num variable in the main function.

Exercise 3:Write a C++ program that swaps the values of two integers using the "call by value" mechanism.

The program should prompt the user to enter two integers, call a function to swap the values, and then display the updated values.

Solution (Call by Value):

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

void swapValues(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int num1, num2;
    cout << "Enter two integers: ";
    cin >> num1 >> num2;

    cout << "Before swapping - num1: " << num1 << ", num2: " << num2 << endl;

    swapValues(num1, num2);

    cout << "After swapping - num1: " << num1 << ", num2: " << num2 << endl;

    return 0;
}
```

Exercise 4:Write a C++ program that swaps the values of two integers using the "call by reference" mechanism.

The program should prompt the user to enter two integers, call a function to swap the values using call by reference, and then display the updated values.

Solution (Call by Reference):

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

void swapValues(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int num1, num2;
    cout << "Enter two integers: ";
    cin >> num1 >> num2;

    cout << "Before swapping - num1: " << num1 << ", num2: " << num2 << endl;

    swapValues(num1, num2);

    cout << "After swapping - num1: " << num1 << ", num2: " << num2 << endl;

    return 0;
}
```

Practical3: Implement the working of classes and objects by using any real world object.

Exercise 5:Implement a Car class in C++ to represent a real-world car object.

The Car class should have the following member variables:

- brand (string): to store the brand of the car.
- model (string): to store the model of the car.
- year (integer): to store the manufacturing year of the car.

The Car class should also have the following member functions:

A constructor that takes parameters to initialize the member variables.

displayDetails(): a member function that displays the details of the car, including the brand, model, and manufacturing year.

Write a C++ program that creates an object of the Car class with the following details:

- Brand: "Toyota"
- Model: "Camry"
- Year: 2021

Display the details of the car using the displayDetails() member function.

Solution:

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

class Car {
 private:
    string brand;
    string model;
    int year;

 public:
    // Constructor
    Car(string carBrand, string carModel, int carYear) {
       brand = carBrand;
       model = carModel;
       year = carYear;
    }

    // Member function to display car details
    void displayDetails() {
       cout << "Car Details:" << endl;
       cout << "Brand: " << brand << endl;
       cout << "Model: " << model << endl;
       cout << "Year: " << year << endl;
    }
};

int main() {
   // Creating an object of the Car class
   Car myCar("Toyota", "Camry", 2021);

   // Calling the displayDetails() member function
   myCar.displayDetails();

   return 0;
}
```

Exercise 6:Create a C++ class named "Rectangle" to represent a rectangle object.

The Rectangle class should have the following member variables:

- length (float): to store the length of the rectangle.
- width (float): to store the width of the rectangle.
- The Rectangle class should also have the following member functions:

A constructor that takes parameters to initialize the length and width of the rectangle.

calculateArea(): a member function that calculates and returns the area of the rectangle.

calculatePerimeter(): a member function that calculates and returns the perimeter of the rectangle.

displayDetails(): a member function that displays the details of the rectangle, including its length, width, area, and perimeter.

Write a C++ program that creates an object of the Rectangle class with the following details:

- Length: 5.3
- Width: 2.7

Display the details of the rectangle using the displayDetails() member function

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

class Rectangle {
private:
    float length;
    float width;

public:
    // Constructor
    Rectangle(float rectLength, float rectWidth) {
        length = rectLength;
        width = rectWidth;
    }

    // Member function to calculate the area of the rectangle
    float calculateArea() {
        return length * width;
    }

    // Member function to calculate the perimeter of the rectangle
    float calculatePerimeter() {
        return 2 * (length + width);
    }

    // Member function to display rectangle details
    void displayDetails() {
        cout << "Rectangle Details:" << endl;
        cout << "Length: " << length << endl;
        cout << "Width: " << width << endl;
        cout << "Area: " << calculateArea() << endl;
        cout << "Perimeter: " << calculatePerimeter() << endl;
    }
};

int main() {
   // Creating an object of the Rectangle class
   Rectangle myRectangle(5.3, 2.7);

   // Calling the displayDetails() member function
   myRectangle.displayDetails();

   return 0;
}
```

Practical 4: Create any user defined class using the concept of static data and member functions.

Exercise 7: Create a user-defined class called "Employee" that represents an employee in a company.
The Employee class should have the following attributes:

static data member "companyName" that stores the name of the company (same for all employees)

instance data member "name" that stores the name of the employee

instance data member "salary" that stores the salary of the employee

The Employee class should also have the following member functions:

A static member function called "changeCompanyName" that allows changing the company name.

A member function called "displayDetails" that displays the name, salary, and company name of an employee.

Write the necessary code to implement the Employee class and demonstrate its usage by creating two employee objects and performing the following tasks:

Set the company name for both employees.

Set the name and salary for each employee.

Display the details of both employees.

```
#include <iostream>
#include <string>
```

```
class Employee {
    static std::string companyName;
    std::string name;
    float salary;

public:
    static void changeCompanyName(const std::string& newName) {
        companyName = newName;
    }

    Employee(const std::string& empName, float empSalary) {
        name = empName;
        salary = empSalary;
    }

    void displayDetails() {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Salary: " << salary << std::endl;
        std::cout << "Company Name: " << companyName << std::endl;
    }
};

std::string Employee::companyName = "";

int main() {
    Employee::changeCompanyName("ABC Corp");

    Employee employee1("John Doe", 5000);
    Employee employee2("Jane Smith", 6000);

    employee1.displayDetails();
    std::cout << std::endl;
    employee2.displayDetails();
    return 0;
}
```

Exercise 8: Create a user-defined class called "MathUtils" that contains static member functions for performing mathematical operations.

The MathUtils class should have the following static member functions:

"square" that takes a number as input and returns its square.

"cube" that takes a number as input and returns its cube.

"factorial" that takes a positive integer as input and returns its factorial.

Write the necessary code to implement the MathUtils class and demonstrate its usage by performing the following tasks:

- Calculate the square of a given number.
- Calculate the cube of a given number.
- Calculate the factorial of a given positive integer.

```
#include <iostream>

class MathUtils {
public:
    static int square(int number) {
        return number * number;
    }

    static int cube(int number) {
        return number * number * number;
    }

    static int factorial(int number) {
        if (number == 0)
            return 1;
        else
            return number * factorial(number - 1);
    }
};

int main() {
    int number = 5;
    int squareResult = MathUtils::square(number);
    std::cout << "Square of " << number << " is " << squareResult << std::endl;

    number = 3;
    int cubeResult = MathUtils::cube(number);
    std::cout << "Cube of " << number << " is " << cubeResult << std::endl;

    number = 4;
    int factorialResult = MathUtils::factorial(number);
    std::cout << "Factorial of " << number << " is " << factorialResult << std::endl;

    return 0;
}
```

Practical 5: Create a Class or program implementing the concept of passing and returning object to/from member functions.

Exercise 9: Passing and Returning Objects in Member Functions.

Objective:To create a class or program that demonstrates the concept of passing and returning objects to/from member functions.

Instructions:

Create a class called "Rectangle" with the following private attributes:

- length (integer)
- width (integer)

Implement the following public member functions in the Rectangle class:

- setDimensions(int l, int w): Sets the length and width of the rectangle based on the given parameters.
- calculateArea(): Calculates and returns the area of the rectangle.
- calculatePerimeter(): Calculates and returns the perimeter of the rectangle.
- compareRectangles(Rectangle r): Compares the area of the current rectangle with another rectangle (r) passed as a parameter. It should return 1 if the current rectangle has a greater area, -1 if the passed rectangle has a greater area, and 0 if both rectangles have the same area.

In the main function, create two Rectangle objects and perform the following tasks:

- Set the dimensions of the first rectangle to length=5 and width=7 using the setDimensions() function.
- Set the dimensions of the second rectangle to length=4 and width=9 using the setDimensions() function.
- Calculate and display the area and perimeter of both rectangles using the appropriate member functions.

Compare the areas of the two rectangles using the compareRectangles() function and display the result.

```
#include <iostream>

using namespace std;

class Rectangle {
private:
   int length;
   int width;

public:
   void setDimensions(int l, int w) {
      length = l;
      width = w;
   }

   int calculateArea() {
      return length * width;
   }

   int calculatePerimeter() {
      return 2 * (length + width);
```

```
    }

    int compareRectangles(Rectangle r) {
        int area1 = calculateArea();
        int area2 = r.calculateArea();

        if (area1 > area2) {
            return 1;
        } else if (area1 < area2) {
            return -1;
        } else {
            return 0;
        }
    }
};

int main() {
    Rectangle rect1, rect2;

    rect1.setDimensions(5, 7);
    rect2.setDimensions(4, 9);

    cout << "Rectangle 1:" << endl;
    cout << "Area: " << rect1.calculateArea() << endl;
    cout << "Perimeter: " << rect1.calculatePerimeter() << endl;

    cout << "Rectangle 2:" << endl;
    cout << "Area: " << rect2.calculateArea() << endl;
    cout << "Perimeter: " << rect2.calculatePerimeter() << endl;

    int comparisonResult = rect1.compareRectangles(rect2);

    if (comparisonResult == 1) {
        cout << "Rectangle 1 has a greater area than Rectangle 2." << endl;
    } else if (comparisonResult == -1) {
        cout << "Rectangle 2 has a greater area than Rectangle 1." << endl;
    } else {
        cout << "Both rectangles have the same area." << endl;
    }

    return 0;
}
```

Practical 6: To implement polymorphism through function overloading (Area of different shapes).

Exercise 10: Write a C++ program to implement polymorphism through function

overloading for calculating the area of different shapes.

The program should include the following shapes: rectangle, triangle, and circle. The program should provide a menu-driven interface to the user, allowing them to choose a shape and enter the required dimensions for calculating the area. Implement the area calculation using function overloading.

Solution:

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

// Function to calculate the area of a rectangle
float area(float length, float breadth) {
   return length * breadth;
}

// Function to calculate the area of a triangle
float area(float base, float height) {
   return 0.5 * base * height;
}

// Function to calculate the area of a circle
float area(float radius) {
   return 3.14 * radius * radius;
}

int main() {
   int choice;

   do {
      cout << "Menu:\n";
      cout << "1. Calculate the area of a rectangle\n";
      cout << "2. Calculate the area of a triangle\n";
      cout << "3. Calculate the area of a circle\n";
      cout << "4. Exit\n";
      cout << "Enter your choice: ";
      cin >> choice;

      switch (choice) {
         case 1:
            float length, breadth;
            cout << "Enter length and breadth: ";
            cin >> length >> breadth;
            cout << "Area of the rectangle: " << area(length, breadth) << endl;
            break;

        case 2:
            float base, height;
            cout << "Enter base and height: ";
            cin >> base >> height;
            cout << "Area of the triangle: " << area(base, height) << endl;
            break;

        case 3:
            float radius;
            cout << "Enter radius: ";
            cin >> radius;
            cout << "Area of the circle: " << area(radius) << endl;
            break;

        case 4:
            cout << "Exiting the program.\n";
            break;

        default:
            cout << "Invalid choice. Please try again.\n";
            break;
      }

      cout << endl;

   } while (choice != 4);

   return 0;
}
```

Practical 7:Create a user defined type Complex and do all the Complex number arithmetic and also make use of operator overloading.

Exercise 11: Write a C++ program to implement a user-defined type Complex.

The Complex type should have the following functionalities:

- Accept and display the real and imaginary parts of a complex number.
- Perform addition, subtraction, multiplication, and division of two complex numbers using operator overloading.
- Display the result of each arithmetic operation.

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

class Complex {
private:
    double real;
    double imaginary;

public:
    Complex(double r = 0, double i = 0) {
        real = r;
        imaginary = i;
    }

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

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

    Complex operator-(Complex const &obj) {
        Complex result;
        result.real = real - obj.real;
        result.imaginary = imaginary - obj.imaginary;
        return result;
    }
```

```
  Complex operator*(Complex const &obj) {
    Complex result;
    result.real = (real * obj.real) - (imaginary * obj.imaginary);
    result.imaginary = (real * obj.imaginary) + (imaginary * obj.real);
    return result;
  }

  Complex operator/(Complex const &obj) {
    Complex result;
    double denominator = (obj.real * obj.real) + (obj.imaginary * obj.imaginary);
    result.real = ((real * obj.real) + (imaginary * obj.imaginary)) / denominator;
    result.imaginary = ((imaginary * obj.real) - (real * obj.imaginary)) / denominator;
    return result;
  }
};

int main() {
  Complex num1(2.5, 3.7);
  Complex num2(1.6, 2.8);

  Complex sum = num1 + num2;
  Complex difference = num1 - num2;
  Complex product = num1 * num2;
  Complex quotient = num1 / num2;

  cout << "Complex Numbers: " << endl;
  cout << "Number 1: ";
  num1.display();
  cout << "Number 2: ";
  num2.display();
```

```
    cout << "Arithmetic Operations: " << endl;
    cout << "Sum: ";
    sum.display();
    cout << "Difference: ";
    difference.display();
    cout << "Product: ";
    product.display();
    cout << "Quotient: ";
    quotient.display();

    return 0;
}
```

Practical 8: Implement single level inheritance by using Student and Marks class.

Exercise 12: Write a C++ program to implement single-level inheritance by using the Student and Marks classes.

The Student class should have data members for the student's name and roll number, along with member functions to input and display the student's details. The Marks class should inherit from the Student class and have data members for the marks obtained in three subjects, along with member functions to input and display the marks. Implement appropriate constructors and destructor in both classes.

Solution:

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

class Student {
   string name;
   int rollNumber;

public:
   Student() {
        name = "";
        rollNumber = 0;
    }

    void input() {
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter roll number: ";
        cin >> rollNumber;
    }

    void display() {
        cout << "Name: " << name << endl;
        cout << "Roll Number: " << rollNumber << endl;
    }
};

class Marks : public Student {
    int subject1;
    int subject2;
    int subject3;

public:
    Marks() {
        subject1 = 0;
        subject2 = 0;
        subject3 = 0;
    }

    void inputMarks() {
        cout << "Enter marks for subject 1: ";
        cin >> subject1;
        cout << "Enter marks for subject 2: ";
        cin >> subject2;
        cout << "Enter marks for subject 3: ";
        cin >> subject3;
    }

    void displayMarks() {
        cout << "Marks for subject 1: " << subject1 << endl;
        cout << "Marks for subject 2: " << subject2 << endl;
        cout << "Marks for subject 3: " << subject3 << endl;
    }
};

int main() {
    Marks studentMarks;

    studentMarks.input();        // Input student details
    studentMarks.inputMarks();   // Input marks
    cout << endl;
    studentMarks.display();      // Display student details
    studentMarks.displayMarks(); // Display marks
    return 0;
}
```

Practical 9: Implement multilevel inheritance by using the Stack class.

Exercise 13: Create a multilevel inheritance hierarchy using the Stack class as the base class. The derived classes should be named NumericStack and StringStack.
The Stack class should have the following functionalities:

- A constructor that initializes an empty stack.
- A method named push() that accepts an item and adds it to the top of the stack.
- A method named pop() that removes and returns the top item from the stack.
- A method named is_empty() that checks if the stack is empty.
- A method named display() that prints the contents of the stack.

The NumericStack class should inherit from the Stack class and additionally have:

- A method named get_average() that calculates and returns the average of all the numeric elements in the stack.

The StringStack class should inherit from the Stack class and additionally have:

- A method named count_vowels() that counts and returns the number of vowels in all the string elements in the stack.

Write a program that demonstrates the use of these classes. Your program should perform the following actions:

1. Create a NumericStack object.
2. Push 5 numeric values (e.g., 10, 20, 30, 40, 50) into the NumericStack.
3. Display the contents of the NumericStack.
4. Calculate and display the average of the numeric elements in the NumericStack.
5. Create a StringStack object.
6. Push 3 string values (e.g., "hello", "world", "openai") into the StringStack.
7. Display the contents of the StringStack.
8. Count and display the number of vowels in the string elements in the StringStack

```
#include <iostream>
#include <string>
using namespace std;
class Stack {
protected:
  static const int MAX_SIZE = 100;
  int top;
  int arr[MAX_SIZE];
public:
  Stack() {
    top = -1;
  }

  void push(int item) {
    if (top == MAX_SIZE - 1) {
            cout << "Stack Overflow!" << endl;
            return;
        }
        arr[++top] = item;
    }
    int pop() {
        if (top == -1) {
            cout << "Stack Underflow!" << endl;
            return -1;
        }
        return arr[top--];
    }

    bool is_empty() {
        return top == -1;
    }
    void display() {
        cout << "Stack Contents: ";
for (int i = top; i >= 0; i--) {
cout << arr[i] << " ";
        }
        cout << endl;
    }
};
class NumericStack : public Stack {
public:
    double get_average() {
        if (top == -1) {
            cout << "Stack is empty!" << endl;
            return 0.0;
        }
        double sum = 0.0;
        for (int i = 0; i <= top; i++) {
            sum += arr[i];
        }
        return sum / (top + 1);
    }
};

class StringStack : public Stack {
public:
    int count_vowels() {
        if (top == -1) {
            cout << "Stack is empty!" << endl;
            return 0;
        }
        int vowelCount = 0;
        string str;
for (int i = 0; i <= top; i++) {
str = to_string(arr[i]);
            for (char ch : str) {
                if (tolower(ch) == 'a' || tolower(ch) == 'e' || tolower(ch) == 'i' ||
                    tolower(ch) == 'o' || tolower(ch) == 'u') {
                    vowelCount++;
                }
            }
        }
        return vowelCount;
    }
};

int main() {
    NumericStack numStack;
    numStack.push(10);
    numStack.push(20);
    numStack.push(30);
    numStack.push(40);
    numStack.push(50);

    cout << "Numeric Stack:" << endl;
    numStack.display();
    cout << "Average: " << numStack.get_average() << endl;

    StringStack strStack;
    strStack.push(104);
    strStack.push(101);
    strStack.push(108);
    strStack.push(108);
    strStack.push(111);
    strStack.push(119);
    strStack.push(111);
    strStack.push(114);
    strStack.push(108);
    strStack.push(100);

    cout << "String Stack:" << endl;
    strStack.display();
    cout << "Vowel Count: " << strStack.count_vowels() << endl;

    return 0;
}
```

For Vivekananda Global Univers

Registrar

Practical 10: Implement the concept of Abstract classes and virtual functions by using

Shape, Rectangle and Triangle class.

Exercise 14: In this exercise, you will implement the concept of abstract classes and virtual functions by creating the Shape, Rectangle, and Triangle classes. The Shape class will be an abstract base class, while the Rectangle and Triangle classes will inherit from it.

Your task is to complete the implementation of the classes and ensure that the appropriate functions are marked as abstract using pure virtual functions. You will also override the virtual functions in the derived classes to calculate the area of the shapes.

To implement the classes and their functions. Follow the steps below:

1. Create an abstract base class called Shape.
2. Inside the Shape class, declare a pure virtual function called getArea(). This function will be used to calculate the area of different shapes and should return a double.
3. Create a derived class called Rectangle that inherits from Shape.
4. Inside the Rectangle class, declare private member variables for the length and width of the rectangle.
5. Implement a constructor for the Rectangle class that initializes the length and width.
6. Override the getArea() function in the Rectangle class to calculate and return the area of the rectangle using the formula length * width.
7. Create another derived class called Triangle that also inherits from Shape.
8. Inside the Triangle class, declare private member variables for the base and height of the triangle.
9. Implement a constructor for the Triangle class that initializes the base and height.
10. Override the getArea() function in the Triangle class to calculate and return the area of the triangle using the formula 0.5 * base * height.

```
#include <iostream>

class Shape {
public:
   virtual double getArea() const = 0; // Pure virtual function
};

class Rectangle : public Shape {
private:
   double length;
   double width;

public:
   Rectangle(double length, double width) : length(length), width(width) {}

   double getArea() const override {
      return length * width;
   }
};

class Triangle : public Shape {
private:
   double base;
   double height;
public:
    Triangle(double base, double height) : base(base), height(height) {}

    double getArea() const override {
        return 0.5 * base * height;
    }
};

int main() {
    Shape* shape1 = new Rectangle(5.0, 3.0); // Create a Rectangle object
    double area1 = shape1->getArea(); // Call the getArea() function
    std::cout << "Area of Rectangle: " << area1 << std::endl;

    Shape* shape2 = new Triangle(4.0, 6.0); // Create a Triangle object
    double area2 = shape2->getArea(); // Call the getArea() function
    std::cout << "Area of Triangle: " << area2 << std::endl;

    delete shape1;
    delete shape2;

    return 0;
}
```

For Vivekananda Globa

Registrar