Syllabus :
Object-Oriented Programming Concepts: Introduction, comparison between procedural programming paradigm and object-oriented programming paradigm, basic concepts of object oriented programming — concepts of an object and a class, interface and implementation of a class, operations on objects, relationship among objects, abstraction, encapsulation, data hiding, inheritance, overloading, polymorphism, messaging.
Classes and Objects: Specifying a class, creating class objects, accessing class members, access specifiers, static members, use of const keyword, friends of a class, empty classes, nested classes, local classes, abstract classes, container classes, bit fields and classes.

Section 1: Object-Oriented Programming Concepts

1.1 Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm that uses classes and objects for designing and implementing software. It is based on the principles of encapsulation, inheritance, abstraction and polymorphism. It emphasizes the encapsulation of data (variable) and behavior (function) within objects, promoting reusability, modularity, and maintainability.

A. Benefits of OOP

1. **Improved code organization:** OOP organizes code around objects and classes, making it easier to understand and maintain.
2. **Modularity:** OOP promotes the development of modular components, allowing for easier troubleshooting and updates.
3. **Reusability:** Objects can be reused across different parts of a program or in different programs, leading to faster development and fewer errors.

B. Procedural vs. Object-Oriented Programming

Aspect	Procedural Programming	Object-Oriented Programming
Primary Focus	Focuses on functions and procedure.	Focuses on classes and objects that encapsulate data and behavior.
Data Handling	Data is typically global and can be accessed by any part of the program.	Emphasizes encapsulation, data hiding, and object-oriented design principles. Data hiding is achieved through access modifiers.
Organization	Programs are structured around functions.	Programs are structured around objects and classes, promoting modularity and code reusability.

Code Reusability	Code reuse is limited to functions.	Encourages the reuse of classes and objects through inheritance and composition.
Maintainabillity	Managing complexity becomes challenging as the program grows in size.	Provides better organization and easier maintenance through modularization and encapsulation.
Encapsulation	Not a primary concern; data and functions are loosely connected.	Encourages bundling data and methods into a single unit (class), promoting data hiding and abstraction.
Inheritance	Not supported or limited (e.g., through library functions).	Supports inheritance, allowing new classes to inherit properties and behaviors from existing classes, promoting code reusability.
Polymorphism	Achieved through functions with the same name but different parameters.	Achieved through method overriding and dynamic dispatch, allowing objects to be treated as instances of their parent class.
Abstraction	Limited; focuses more on procedural logic.	Promotes abstraction by simplifying complex systems through modeling classes based on essential features.
Example Languages	C, Pascal, BASIC	Java, C++, Python

1.2 Basic Concept of OOP

A. Classes and Objects:

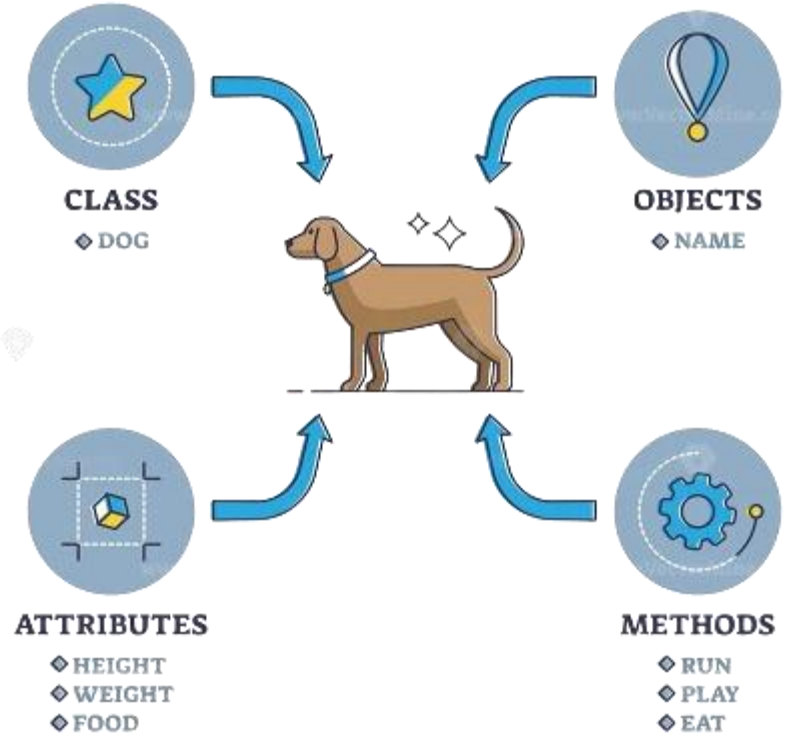
Classes:

- A class is a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.
- A class encapsulates data members (attributes) and member functions (methods) that operate on the data.
- Classes facilitate code reusability and promote modularity by organizing related data and functions into a single unit.
- Classes act as user-defined data types.
- In C++, a class is declared using the class keyword, followed by the class name and a pair of curly braces containing class members.
- **Syntax:**

```
class MyClass {
    // Class members (attributes and methods)
};
```

Objects:

- Objects are instances of classes. They represent specific instances of the class, each with its own unique state.
- An object encapsulates data and behaviour, and provides an interface (through its methods) to interact with that data.
- Multiple objects can be created from the same class, each with its own independent state.
- Object attributes can be initialized using the dot (.) operator.



B. Attributes and Methods:

Attributes:

- Attributes are the data members or variables associated with a class.
- They represent the state of an object and define its characteristics or properties.
- Attributes are declared within the class and are usually private to enforce encapsulation, but they can also be public or protected based on the desired access level.
- Data members can be accessed using the dot (.) operator.

Methods:

- Methods are functions associated with a class.
- They define the behavior or actions that objects of the class can perform.
- Member functions are called using the dot (.) operator
- Methods operate on the object's data (attributes) and can manipulate its state.
- Methods can be public, private, or protected, determining their accessibility from outside the class.

Example:

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

// Class declaration
class Student {
public:
    // Data members
    int studentID;
    string name;

    // Member function to display student information
    void displayInfo() {
        cout << "Student ID: " << studentID << ", Name: " << name << endl;
    }
};

int main() {
    // Creating objects of class Student
    Student student1, student2;

    // Initializing object attributes
    student1.studentID = 101;
    student1.name = "Deepak Modi";

    student2.studentID = 102;
    student2.name = "Sumit Modi";

    // Accessing member functions to display student information
    cout << "Student 1: ";
    student1.displayInfo();

    cout << "Student 2: ";
    student2.displayInfo();

    return 0;
}
```

C. Defining Classes:

Defining a Class without a Constructor

- When a class is defined without a constructor, the compiler generates a default constructor implicitly. This default constructor initializes the data members of the

class with default values (zero for numeric types, empty string for string types, etc.).

- However, if any constructor is explicitly defined within the class, the compiler does not generate the default constructor.

Example:

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

// Class declaration
class Person {
public:
    // Data members
    string name;
    int age;

    // Member function to display person's information
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    // Creating an object of class Person
    Person p1;

    // Initializing object attributes
    p1.name = "Sanjay";
    p1.age = 30;

    // Accessing member function to display person's information
    p1.displayInfo();

    return 0;
}
```

Defining a Class with a Constructor

- A constructor is a special member function of a class that is automatically called when an object of that class is created.
- It is used to initialize object properties.
- It has the same name as the class. It has no return type.

Example:

```
#include <iostream>
```

```cpp
#include <string>
using namespace std;

// Class declaration
class Person {
public:
    // Data members
    string name;
    int age;

    // Constructor declaration
    Person(string n, int a) {
        name = n;
        age = a;
    }

    // Member function to display person's information
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    // Creating an object of class Person with constructor
    Person p1("Sanjay", 30);

    // Accessing member function to display person's information
    p1.displayInfo();

    return 0;
}
```

Defining a Class with this Keyword

- The **"this"** keyword in C++ is a pointer that refers to the current instance of a class. It is used within class methods to refer to the current object on which the method is being invoked.
- ***this->member*** is used to access the members (attributes or methods) of the current object.
- It resolves the scope and ensures that the member being accessed belongs to the current object.

Example:

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

```
class MyClass {
private:
    int value;

public:
    // Constructor with parameter
    MyClass(int value) {
        // Using 'this' to distinguish between member and parameter
        this->value = value;
    }

    // Member function to display value
    void displayValue() {
        // Accessing 'value' using 'this'
        cout << "Value: " << this->value << endl;
    }
};

int main() {
    // Creating object of MyClass
    MyClass obj(10);

    // Calling member function
    obj.displayValue();

    return 0;
}
```

D. Constructor and its Types

Constructors are special member functions that are automatically called when an object is created.

Default constructors (with no parameters)

- A default constructor is a constructor that does not take any parameters. It is called implicitly when an object of the class is created without any arguments.
- The default constructor initializes the data members with default values.
- If no constructor is explicitly defined within the class, the compiler generates a default constructor automatically.

Parameterized constructors (with parameters)

- A parameterized constructor is a constructor that takes one or more parameters. It allows for initializing object attributes with specified values at the time of object creation.
- The parameterized constructor takes the arguments and initializes the data members with the provided values.

Example:

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

// Class declaration
class Car {
private:
    string brand;
    string model;
    int year;

public:
    // Default constructor
    Car() {
        brand = "";
        model = "";
        year = 0;
    }

    // Parameterized constructor
    Car(string brand, string model, int year) {
        this->brand = brand;
        this->model = model;
        this->year = year;
    }

    // Member function to display car information
    void displayInfo() {
        cout << "Brand: " << brand << ", Model: " << model << ", Year: "
<< year << endl;
    }
};

int main() {
    // Creating objects of Car class using default and parameterized
constructors
    Car car1;                           // Using default constructor
    Car car2("Toyota", "Camry", 2022);  // Using parameterized constructor

    // Displaying information about the cars
    cout << "Car 1: ";
    car1.displayInfo();

    cout << "Car 2: ";
```

```
    car2.displayInfo();

    return 0;
}
```

E. OOP Principles

Object-Oriented Programming (OOP) is built upon several fundamental principles that help in designing and implementing effective software solutions. These principles are as follows:

1. Encapsulation

- Definition: Encapsulation is the wrapping of data (attributes) and methods (functions) into a single unit or class.
- Benefits:
    - Data Hiding: Encapsulation hides the internal state of an object from the outside world, allowing controlled access to the object's data through well-defined interfaces.
    - Modularity: Encapsulation promotes modularity by grouping related data and functions together, making it easier to manage and maintain complex systems.

2. Abstraction

- Definition: Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object.
- Benefits:
    - Simplification: Abstraction simplifies the system by focusing on what an object does rather than how it does it, making it easier to understand and use.
    - Managing Complexity: Abstraction helps in managing complexity by breaking down a system into manageable parts and hiding unnecessary details from the user.

3. Inheritance

- Definition: Inheritance is a mechanism by which a class (derived class) can inherit properties and behaviors from another class (base class).
- Benefits:
    - Code Reusability: Inheritance facilitates code reuse by allowing derived classes to inherit common functionality from a base class, avoiding redundancy and promoting modularity.
    - Hierarchical Classification: Inheritance establishes a hierarchical relationship between classes, enabling the creation of specialized classes that inherit and extend the functionality of their base classes.

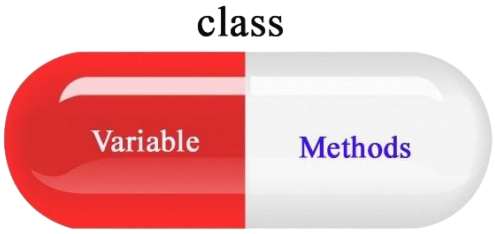
4. Polymorphism

- Definition: Polymorphism is the ability of objects of different classes to be treated as objects of a common superclass. It allows for writing code that can work with objects of various types without needing to know their specific class.
- Benefits:
    - Flexibility: Polymorphism enables writing flexible and extensible code that can accommodate changes and variations in object types without modifying existing code.
    - Code Reusability: Polymorphism promotes code reuse by allowing methods to be defined in terms of their abstract behavior, rather than specific implementations, facilitating easier maintenance and enhancement of code.

1.3 Encapsulation

A. Definition

Encapsulation is the bundling of data (attributes/variables) and methods (functions) that operate on the data into a single unit or class. It hides the internal state of an object from outside interference, allowing access to the data only through well-defined interfaces.



B. Benefits

1. Data Hiding: Encapsulation prevents direct access to the internal state of an object from outside the class, enforcing data hiding. This protects the integrity of the data and prevents unintended modifications.
2. Modularity: Encapsulation promotes modularity by organizing related data and methods into a single unit or class. This improves code organization and makes it easier to understand and maintain.
3. Access Control: Encapsulation allows for controlling access to the data members of a class through access specifiers (public, private, protected), enabling selective exposure of data and behavior.
4. Code Reusability: Encapsulation facilitates code reuse by encapsulating data and behavior within a class, allowing objects of the class to be reused in different parts of the program or in other programs.

Example:

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

// Class definition with encapsulation
```

```
class Employee {
private:
    string name;
    int employeeID;
    double salary;
public:
    // Constructor to initialize employee attributes
    Employee(string n, int id, double s) {
        name = n;
        employeeID = id;
        salary = s;
    }

    // Getter methods to access private attributes
    string getName() {
        return name;
    }

    int getEmployeeID() {
        return employeeID;
    }

    double getSalary() {
        return salary;
    }

    // Setter method to modify salary (demonstrating access control)
    void setSalary(double s) {
        if (s >= 0) {
            salary = s;
        }
    }
};

int main() {
    // Creating an object of class Employee
    Employee emp1("John Doe", 101, 50000.0);

    // Accessing and modifying private attributes using getter and setter methods
    cout << "Employee Name: " << emp1.getName() << endl;
    cout << "Employee ID: " << emp1.getEmployeeID() << endl;
    cout << "Employee Salary: $" << emp1.getSalary() << endl;

    emp1.setSalary(55000.0); // Modifying salary
```

```
    cout << "Updated Employee Salary: $" << emp1.getSalary() << endl;

    return 0;
}
```

C. Access specifiers (public, private, protected) and their role in encapsulation:

Access specifiers determine the accessibility of class members (attributes and methods) from outside the class:

1. **Public:** Members declared as public are accessible from outside the class. They form the interface of the class and can be accessed by any part of the program.
2. **Private:** Members declared as private are only accessible from within the class. They are hidden from outside access, enforcing encapsulation and data hiding.
3. **Protected:** Members declared as protected are accessible from within the class and its derived classes. They provide a level of access that is intermediate between public and private, allowing derived classes to access them while still maintaining encapsulation.

Access specifiers play a crucial role in encapsulation by controlling the visibility of class members, thereby enforcing data hiding and encapsulation principles.

Getter and Setter Methods

Getter and setter methods are a part of encapsulation and are used to access and modify the private data members of a class. Getter methods allow you to retrieve the values of private data members, and setter methods allow you to set or modify the values of private data members.

D. Explain how data hiding is achieved through encapsulation:

- Data hiding is a fundamental concept in object-oriented programming (OOP) that involves restricting direct access to an object's internal details, specifically its data members (attributes).
- The goal of data hiding is to ensure that the internal state of an object remains private and can only be accessed or modified through well-defined interfaces (methods or functions).
- Data hiding is achieved through encapsulation by declaring the data members of a class as private, preventing direct access from outside the class.
- Access to the private data members is provided through public member functions (getters and setters), which act as controlled interfaces for accessing and modifying the data.
- This encapsulation of data within the class hides the internal state of an object, protecting it from unintended modifications and ensuring data integrity.

1.4 Abstraction

A. Definition:

Abstraction is the process of hiding complex implementation details and showing only the essential features of an object. It allows for focusing on what an object does rather than

how it does it, thereby simplifying the complexity of the system and enhancing clarity and maintainability.

B. Explain abstract class and interface:

- Abstract Class: A class that contains at least one pure virtual function. It cannot be instantiated and serves as a blueprint for derived classes to implement common behavior while allowing specific implementations for their own unique features.
- Interface: A class containing only pure virtual functions, representing a contract for classes that implement it. It defines a set of methods that must be implemented by any class that inherits from it, ensuring consistency in behavior while allowing flexibility in implementation.

1.5 Inheritance

A. Definition

- Inheritance is a mechanism by which a new class (derived class) can inherit properties and behaviors from an existing class (base class).
- The derived class inherits the attributes and methods of the base class, enabling code reusability and establishing a hierarchical relationship between classes.
- The class that is being inherited from is known as the "parent class" or "base class" or "superclass", and the class that inherits from the base class is known as the “child class” or "derived class" or "subclass".

- The child class will inherit all the public and protected properties (attributes) and methods from the parent class. In addition, it can have its own properties and methods,this is called inheritance.
- Example: A Car class may inherit from a Vehicle class. Here, Car is a specific type of Vehicle.

B. Benefits

1. Code Reusability: Inheritance allows classes to inherit attributes and methods from other classes, reducing redundancy and promoting code reuse.

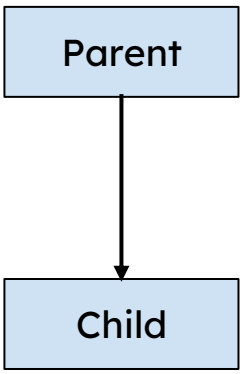
2. Extensibility: Inheritance enables the addition of new features to a class without modifying its existing structure, thus facilitating the extension of functionality.
3. Relationship Representation: Inheritance models real-world relationships making the code more intuitive and reflective of real-world scenarios.
4. Specialization: Inheritance allows for the creation of specialized classes (derived classes) that inherit common features from a more general class (base class) and add their own unique functionalities.
5. Transitivity: If class B inherits from class A, and class C inherits from class B, then class C automatically inherits properties and behaviors from both class A and class B.
6. Hierarchical Organization: Inheritance enables the creation of a hierarchical organization of classes, where classes can be organized into a hierarchy based on their relationships.
7. Encapsulation: Inheritance encourages encapsulation, as it promotes the creation of well-defined, modular classes with clear boundaries.
8. Polymorphism: Inheritance supports polymorphism, which allows objects of derived classes to be treated as objects of their base class.

C. Types of inheritance

1. Single Inheritance:

- In single inheritance, a derived class inherits from only one base class.
- It forms a simple one-to-one relationship between classes.
- It is like Father -> Child relationship.
- Example:

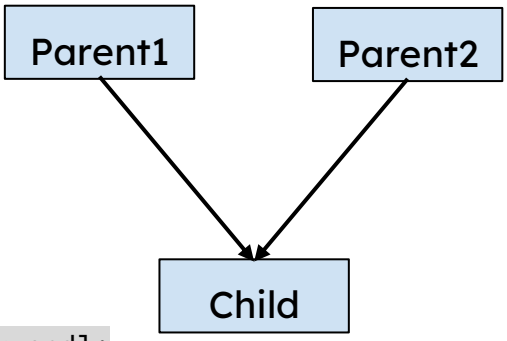
```
#include <iostream>
using namespace std;

// Base class (Parent)
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Derived class (Child) inheriting from Parent
class Child : public Parent {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

int main() {
    Child obj;
```



```
    obj.childMethod(); // Output: Method of Child Class
    obj.parentMethod(); // Output: Method of Parent Class
    return 0;
}
```

2. Multiple Inheritance:

- In multiple inheritance, a derived class inherits from multiple base classes.
- It allows a class to inherit attributes and behaviors from more than one parent class.
- It is like Mother & Father -> Child relationship.
- Example:

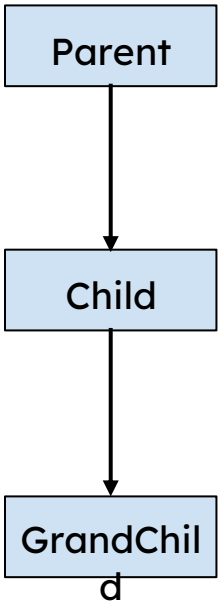



```
#include <iostream>
using namespace std;

// Parent1 class
class Parent1 {
public:
    void parent1Method() {
        cout << "Method of Parent1 Class" << endl;
    }
};

// Parent2 class
class Parent2 {
public:
    void parent2Method() {
        cout << "Method of Parent2 Class" << endl;
    }
};

// Child class inheriting from both Parent1 and Parent2
class Child : public Parent1, public Parent2 {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

int main() {
    Child obj;
    obj.childMethod(); // Output: Method of Child Class
    obj.parent1Method(); // Output: Method of Parent1 Class
    obj.parent2Method(); // Output: Method of Parent2 Class
    return 0;
}
```

3. Multilevel Inheritance:

- In multilevel inheritance, a derived class becomes the base class for another derived class, forming a chain of inheritance.
- It allows for creating a hierarchy of classes with each level adding additional features.
- It is like Father -> Child -> Grandchild relationship.
- Example:

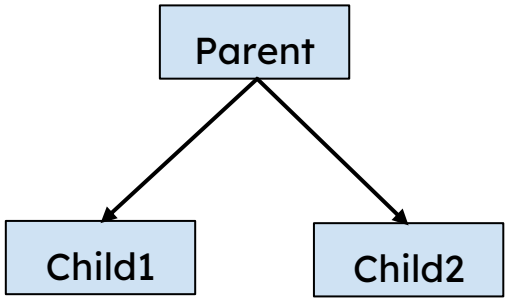
```
#include <iostream>
using namespace std;

// Parent class
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Child class inheriting from Parent
class Child : public Parent {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

// GrandChild class inheriting from Child
class GrandChild : public Child {
public:
    void grandChildMethod() {
        cout << "Method of Grand Child Class" << endl;
    }
};

int main() {
    GrandChild obj;
    obj.grandChildMethod(); // Output: Method of Grand Child Class
    obj.childMethod(); // Output: Method of Child Class
    obj.parentMethod(); // Output: Method of Parent Class
    return 0;
}
```



4. Hierarchical Inheritance:

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- It allows for creating a hierarchy of classes with a common base class.

- It is like Father -> Son & Daughter relationship.
- Example:

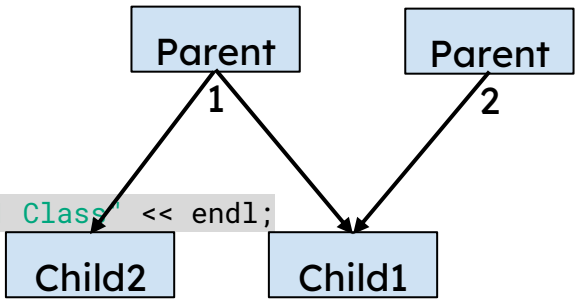



```
#include <iostream>
using namespace std;

// Parent class
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Child1 class inheriting from Parent
class Child1 : public Parent {
public:
    void child1Method() {
        cout << "Method of Child1 Class" << endl;
    }
};

// Child2 class inheriting from Parent
class Child2 : public Parent {
public:
    void child2Method() {
        cout << "Method of Child2 Class" << endl;
    }
};

int main() {
    Child1 obj1;
    obj1.child1Method(); // Output: Method of Child1 Class
    obj1.parentMethod(); // Output: Method of Parent Class

    Child2 obj2;
    obj2.child2Method(); // Output: Method of Child2 Class
    obj2.parentMethod(); // Output: Method of Parent Class

    return 0;
}
```

5. Hybrid Inheritance:

- Hybrid inheritance is a combination of multiple types of inheritance, such as single, multiple, multilevel or hierarchical inheritance.

- It allows for creating complex class hierarchies by combining features of different types of inheritance.
- Example:



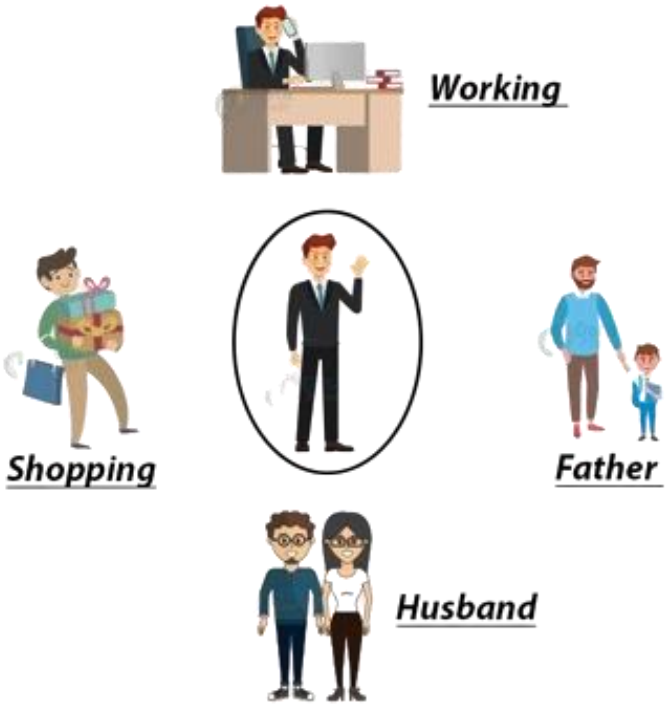

```
#include <iostream>
using namespace std;

// Parent1 class
class Parent1 {
public:
    void parent1Method() {
        cout << "Method of Parent1 Class" << endl;
    }
};

// Parent2 class
class Parent2 {
public:
    void parent2Method() {
        cout << "Method of Parent2 Class" << endl;
    }
};

// Child1 class inheriting from both Parent1 and Parent2
class Child1 : public Parent1, public Parent2 {
public:
    void child1Method() {
        cout << "Method of Child1 Class" << endl;
    }
};

// Child2 class inheriting from Parent1
class Child2 : public Parent1 {
public:
    void child2Method() {
        cout << "Method of Child2 Class" << endl;
    }
};

void main() {
    Child1 obj1;
    obj1.child1Method(); // Output: Method of Child1 Class
    obj1.parent1Method(); // Output: Method of Parent1 Class
    obj1.parent2Method(); // Output: Method of Parent2 Class

    Child2 obj2;
```

```
    obj2.child2Method(); // Output: Method of Child2 Class
    obj2.parent1Method(); // Output: Method of Parent1 Class

}
```

1.6 Polymorphism

A. Definition

- Poly means 'many' and morph means 'forms'.
- Polymorphism refers to the ability of objects to take on different forms or behaviors based on their context.
- In OOP, polymorphism refers to the ability of objects of different classes to be treated as objects of a common superclass.
- It allows a single interface (method or function) to represent multiple implementations.
- It allows a single function or operator to exhibit different behaviors based on the context in which it is called.
- Example: A man acts a father, husband, son, employee and many more.



B. Benefits

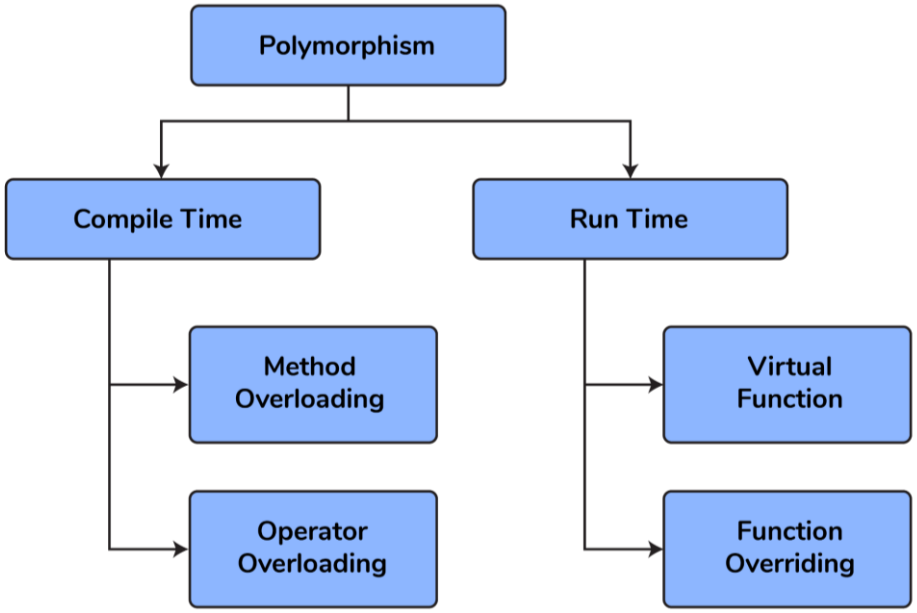
1. Code Reusability: Polymorphism allows the same code to be reused with different objects, reducing duplication and improving maintainability.
2. Flexibility: Polymorphism enables the development of flexible systems that can accommodate changes and variations in object behavior without modifying existing code.

3. Extensibility: New classes can be added to the system without affecting the existing codebase, as long as they adhere to the common interface provided by the base class.
4. Encapsulation: Polymorphism promotes encapsulation by abstracting away the implementation details of objects and focusing on their behavior through a common interface.

C. Types of Polymorphism

Polymorphism can be of two types:

1. **Compile Time Polymorphism (Early / Static Binding)**
    - Achieved through method overloading and operator overloading.
    - Decisions about method calls are made at compile time.
    - Determined by the number and types of arguments and return type.
2. **Run Time Polymorphism (Late / Dynamic Binding)**
    - Achieved through virtual functions or method overriding and dynamic diispatch.
    - Decisions about method calls are made at runtime.
    - Facilitated by pointers or references to base class objects.



Compile-time and Run-time

- **Compile Time (Early/Static):**
    - This is the phase when the **source code** you've written is being **converted into executable code** by a compiler.
    - During compile time, the compiler checks for **syntax errors** (like missing semicolons or mismatched brackets) and **semantic errors** (such as using a variable that hasn't been declared).
    - The compiler will not create an executable file until all such errors are resolved.
- **Run Time (Late/Dynamic):**
    - Run time refers to the period when the **executable code is actually running** on your computer.

- It's the phase where the program interacts with inputs, performs calculations, and may encounter **runtime errors**. These errors occur during execution and can include issues like division by zero or accessing an array out of bound.

1. Compile-time Polymorphism (Static Polymorphism):

- Polymorphism that is resolved at compile time. It is achieved through method overloading and operator overloading, where the compiler selects the appropriate function or operator based on the arguments and context at compile time.
- Also known as **static or early binding**, this type of polymorphism is achieved through method overloading and operator overloading.

Method Overloading:

- Definition: Method overloading is a form of compile-time polymorphism where multiple methods in the same class have the same name but differ in the number or type of their parameters.
- Methods can be overloaded by changing the number or type of arguments.
- It provides flexibility and clarity in code by allowing multiple functions with similar functionality to be grouped under the same name.
- Example:

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

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

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

    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(5, 7) << endl;         // Output: 12
    cout << calc.add(5, 7, 3) << endl;      // Output: 15
    cout << calc.add(3.5, 2.5) << endl;     // Output: 6
    return 0;
}
```

Operator Overloading:

- Definition: Operator overloading is a form of compile-time polymorphism where operators are overloaded to work with user-defined data types.
- It allows defining custom behavior for operators based on the data types involved.
- Example:

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

class Complex {
private:
    int real, imag;
public:
    // Constructor to initialize real and imaginary parts, default
values are 0
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

    // Overloading the + operator to add two complex numbers
    Complex operator+(Complex const& obj) {
        // Adding real parts and imaginary parts separately
        return Complex(real + obj.real, imag + obj.imag);
    }

    // Function to print the complex number in the format "real +
imagi"
    void print() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    // Creating two complex numbers
    Complex c1(10, 5), c2(2, 4);

    // Adding two complex numbers using overloaded + operator
    Complex c3 = c1 + c2;

    // Printing the result
    cout << "Result of addition: ";
    c3.print();

    return 0;
}
```

Output:

```
Result of addition: 12 + 9i
```

2. Run-time Polymorphism (Dynamic Polymorphism):

- Polymorphism that is resolved at runtime. It is achieved through method overriding and virtual functions, where the appropriate function to call is determined dynamically based on the actual object type at runtime.
- Also known as **dynamic or late binding**, this occurs during program execution.
- Achieved through virtual functions (using the virtual keyword).
- Allows a base class pointer to invoke derived class methods.

Virtual Functions:

- Definition: Virtual functions are used in run-time polymorphism to enable dynamic method binding. They are member functions which are declared in the base class with the virtual keyword and can be overridden in derived classes.
- They enable dynamic binding of function calls, allowing the correct function to be called at runtime based on the type of object.
- Example:

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

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

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

int main() {
    Shape* shape = new Circle();
    shape->draw();  // Output: Drawing Circle

    return 0;
}
```

Method Overriding:

- Definition: Method overriding is a form of run-time polymorphism where a method in a base class is redefined in a derived class. The method in the derived class must have the same signature (name and parameters) as the one in the base class.
- It allows derived classes to provide a specific implementation of a method defined in the base class, promoting flexibility and extensibility.

- Example:

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

class Animal {
public:
    virtual void sound() {
        cout << "Animal makes a sound" << endl;
    }
};

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

int main() {
    Animal* animal = new Dog();
    animal->sound();  // Output: Dog barks

    return 0;
}
```

Section 2: Other Concepts of OOP

2.1 Messaging

Messaging in object-oriented programming (OOP) refers to the process of communication between objects in a system. It involves sending messages from one object to another to request actions, retrieve information, or trigger behaviors. In C++, messaging is primarily achieved through method calls, where an object invokes a method of another object to interact with it.

Method Calls as Messages:

- In C++, objects communicate with each other by invoking methods of other objects.
- When an object sends a message to another object, it invokes a method of the target object to perform a specific action or retrieve information.
- Method calls are the primary means of messaging in C++.

Sender and Receiver Objects:

- In a message-passing scenario, the object that sends the message is called the sender, and the object that receives the message is called the receiver.

- Sender objects typically invoke methods of receiver objects to communicate with them.

Passing Parameters:

- Messages can include parameters that provide additional information to the receiver object.
- Parameters are passed as arguments to the method being invoked.

Return Values:

- In many cases, when an object sends a message to another object, it expects a response or a return value.
- The return value of a method call can be used by the sender object for further processing.

Encapsulation and Information Hiding:

- Messaging promotes encapsulation and information hiding principles in OOP.
- Objects encapsulate their data and behaviors, exposing only the necessary methods for communication.
- Encapsulation ensures that the internal state of an object is protected and accessed only through well-defined interfaces.

Example:

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

// Sender class
class Sender {
public:
    // Method to send a message
    void sendMessage(int data, Receiver& receiver) {
        // Invoking receiver's method and passing data as a parameter
        int result = receiver.receiveMessage(data);
        cout << "Received result from receiver: " << result << endl;
    }
};

// Receiver class
class Receiver {
public:
    // Method to receive a message and process data
    int receiveMessage(int data) {
        cout << "Received data from sender: " << data << endl;
        // Processing data and returning result
        return data * 2;
    }
```

```
};

int main() {
    Sender sender;
    Receiver receiver;

    // Sender object sends a message to the receiver object
    sender.sendMessage(5, receiver);

    return 0;
}
```

2.2 Access Specifiers

Access specifiers in C++ are keywords used to define the accessibility of class members (attributes and methods) from outside the class. They play a crucial role in enforcing encapsulation, which is one of the fundamental principles of object-oriented programming (OOP). In C++, there are three main access specifiers: public, private, and protected.

1. Public Access Specifier:

- Members declared as public are accessible from outside the class through objects of that class.
- Public members form the interface of the class, providing access to its functionality.
- Public members can be accessed and modified freely from any part of the program.

Example:

```
class MyClass {
public:
    int publicVar;           // Public attribute
    void publicMethod();     // Public method
};
```

2. Private Access Specifier:

- Members declared as private are accessible only within the class itself.
- They cannot be accessed directly from outside the class or by derived classes.
- Private members are used to hide implementation details and enforce data encapsulation.

Example:

```
class MyClass {
private:
    int privateVar;          // Private attribute
    void privateMethod();    // Private method
};
```

3. Protected Access Specifier:

- Members declared as protected are accessible within the class itself and by derived classes.
- They are not accessible from outside the class hierarchy.
- Protected members enable inheritance and code reuse by allowing derived classes to access certain aspects of the base class's implementation.

Example:

```
class MyBaseClass {
protected:
    int protectedVar;       // Protected attribute
    void protectedMethod(); // Protected method
};
```

Access Specifier Usage:

- Access specifiers define the level of encapsulation and control the visibility of class members.
- Public members provide an external interface to the class, allowing users to interact with its functionality.
- Private members are hidden from external access, ensuring data integrity and preventing unauthorized modification.
- Protected members enable inheritance and allow derived classes to access base class functionality.

Example:

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

// Class declaration
class MyClass {
public:
    int publicVar;    // Public attribute
    void publicMethod() {
        cout << "Public method called" << endl;
    }

private:
    int privateVar;   // Private attribute
    void privateMethod() {
        cout << "Private method called" << endl;
    }

protected:
    int protectedVar; // Protected attribute
    void protectedMethod() {
        cout << "Protected method called" << endl;
```

```
    }

public:
    // Constructor to initialize attributes
    MyClass() {
        publicVar = 0;
        privateVar = 0;
        protectedVar = 0;
    }
};

int main() {
    // Create an object of MyClass
    MyClass obj;

    // Accessing public members
    obj.publicVar = 10;
    obj.publicMethod();

    // Accessing private members (will cause compilation error)
    // obj.privateVar = 20;
    // obj.privateMethod();

    // Accessing protected members (will cause compilation error)
    // obj.protectedVar = 30;
    // obj.protectedMethod();

    return 0;
}
```

2.3 Getter and Setter

Getter and setter methods, also known as accessor and mutator methods, are class member functions used to access and modify the private data members of a class. They play a crucial role in implementing encapsulation and data abstraction in object-oriented programming. Getter methods allow us to retrieve the values of private data members, while setter methods enable us to set or modify the values of private data members.

Getter Methods:

- Getter methods are used to retrieve the values of private data members of a class.
- They are defined as public member functions within the class.
- Getter methods provide controlled access to the private data members, ensuring encapsulation and information hiding.
- They typically do not modify the state of the object and are declared as const member functions to indicate that they do not alter the object's internal state.

- Getter methods typically have a return type corresponding to the type of the data member being accessed.
- They provide a read-only view of the data, allowing users to retrieve information without directly accessing the private data members.

```
class MyClass {
private:
    int privateVar;

public:
    // Getter method to retrieve the value of privateVar
    int getPrivateVar() const {
        return privateVar;
    }
};
```

Setter Methods:

- Setter methods are used to set or modify the values of private data members of a class.
- They are defined as public member functions within the class.
- Setter methods provide controlled access to modify the private data members, ensuring encapsulation and data integrity.
- They typically take parameters corresponding to the new values to be assigned to the private data members.
- Setter methods may include validation checks or additional logic to ensure the validity of the new values being assigned.
- They are responsible for maintaining the internal consistency of the object's state.

```
class MyClass {
private:
    int privateVar;

public:
    // Setter method to set the value of privateVar
    void setPrivateVar(int newValue) {
        privateVar = newValue;
    }
};
```

Usage:

- Getter and setter methods allow controlled access to private data members, preventing direct manipulation from outside the class.
- They provide an interface for interacting with the class's internal state, hiding implementation details and promoting abstraction.
- Getter and setter methods facilitate encapsulation by encapsulating the internal representation of data and exposing only essential functionality to the user.

```
MyClass obj;
```

```
obj.setPrivateVar(10);       // Using setter method to set privateVar
int value = obj.getPrivateVar(); // Using getter method to retrieve
privateVar
```

Benefits:

- Encapsulation: Getter and setter methods encapsulate the access to private data members, preventing direct modification and ensuring data integrity.
- Abstraction: They provide a simplified interface for interacting with class objects, hiding implementation details and promoting abstraction.
- Data Validation: Setter methods enable validation checks to ensure the validity of new values being assigned to private data members.
- Flexibility: Getter and setter methods provide flexibility to modify the internal representation of data without affecting external code that interacts with the class.

2.4 Static Members (Static Keyword)

The static keyword in C++ is used to define class-level members that are shared among all instances of the class. These class-level members can include static variables and static methods. Static members belong to the class itself rather than individual objects, and they are shared among all instances of the class.
The static keyword can also be used in local variables and functions, but here we'll focus on its usage with class members.

1. Static Variables (Static Data Members):

- Static variables are shared among all instances of the class, rather than each instance having its own copy.
- They are declared with the static keyword inside the class definition.
- Static variables are initialized once, before any object of the class is created, and they retain their values throughout the program's execution.
- They are typically used for maintaining class-wide data or for counting the number of objects created from the class.

Example:

```
class MyClass {
public:
    static int staticVar; // Declaration of static variable
};

int MyClass::staticVar = 0; // Initialization of static variable

// Usage
MyClass obj1;
MyClass obj2;
obj1.staticVar = 10; // Accessing static variable using object
cout << obj2.staticVar; // Output: 10
```

2. Static Methods (Static Member Functions):

- Static methods belong to the class rather than to any specific object.
- They can be called using the class name without needing an object.
- Static methods cannot access non-static members of the class directly, as they do not have access to any specific object's state.
- They are typically used for performing operations that do not depend on the state of any particular object.

Example:

```
class MyClass {
public:
    static void staticMethod() {
        cout << "Static method called" << endl;
    }
};

// Usage
MyClass::staticMethod(); // Calling static method using class name
```

Benefits of Static Members:

- Memory Efficiency: Static variables are shared among all instances of the class, reducing memory usage as compared to non-static variables that have a separate copy for each instance.
- Global Access: Static methods can be called using the class name without needing an instance of the class, allowing for global access to certain functionalities.
- Class-wide Operations: Static members are useful for operations that are common to all instances of the class, such as maintaining global state or performing utility functions.
- Initialization: Static variables are initialized once before any object of the class is created, providing predictable behavior and avoiding initialization issues.

Scope Resolution Operator:

- The scope resolution operator is used to access static members (static variables and static methods) of a class. Static members belong to the class itself rather than individual objects, and they are shared among all instances of the class.

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

class MyClass {
public:
    static int count; // Static variable declaration
    static void displayCount() { // Static method definition
        cout << "Count: " << count << endl;
    }
};
```

```
int MyClass::count = 0; // Static variable definition

int main() {
    MyClass::count = 5; // Accessing and modifying static variable
    MyClass::displayCount(); // Accessing static method
    return 0;
}
```

- The scope resolution operator can be used to access global variables and functions from within a local scope. This is particularly useful when there's a local variable or function with the same name as a global one.

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

int x = 10; // Global variable

int main() {
    int x = 20; // Local variable
    cout << "Local variable x: " << x << endl;        // Output: Local
variable x: 20
    cout << "Global variable x: " << ::x << endl;     // Output: Global
variable x: 10
    return 0;
}
```

2.5 Constant Members (Const Keyword)

The const keyword in C++ is used to declare constants or to specify that an object or function is immutable, meaning its value cannot be changed after initialization.
When applied to member variables and member functions of a class, const ensures that these members cannot be modified after initialization. This feature enhances code safety, readability, and maintainability.

1. Constant Data Members:

- When applied to member variables of a class, const makes them constant data members.
- Constant data members must be initialized during object creation and cannot be modified thereafter.
- They provide a way to enforce immutability and prevent unintentional modification of critical data within a class.

2. Constant Member Functions:

- When applied to member functions of a class, const indicates that these functions do not modify the state of the object.
- Constant member functions can be called on const and non-const objects, but they cannot modify non-mutable data members.

- They ensure that the object's state remains unchanged when invoking certain operations on const objects.

Example:

```
class Circle {
public:
    const double PI = 3.141592653589793; // Constant data member
    double getArea() const { // Constant member function
        return PI * radius * radius;
    }
private:
    double radius;
};

// Usage
const Circle circle; // Constant object declaration
double area = circle.getArea(); // Invoking constant member function
on const object
```

Benefits:

- Safety: Constant members help prevent accidental modification of critical data and ensure code safety.
- Readability: Clearly defining constant members and constant member functions enhances code readability and understandability.
- Maintainability: Immutable data members and non-modifying member functions simplify code maintenance by reducing the risk of unintended changes and side effects.

2.6 Friends of a Class

In C++, the concept of "friendship" allows a class to grant access to its private and protected members to another class or function. When a class or function is declared as a friend of another class, it can access the private and protected members of that class as if it were a member of that class itself. This feature is often used to allow certain external functions or classes to operate on private data members of a class without violating encapsulation.

1. Declaring Friend Functions:

- In C++, a friend function is a function that is not a member of a class but has the ability to access the class's private and protected members.
- It is declared within the class using the friend keyword.
- Friend functions are typically used when an external function needs to access the internal state of a class in a controlled manner.

```
class MyClass {
private:
    int privateVar;
```

```
public:
    friend void friendFunction(MyClass obj); // Declaration of friend function
};
```

2. Declaring Friend Classes:

- A friend class is declared using the friend keyword within the class definition.
- Friend classes have access to the private and protected members of the class to which they are declared as friends.
- They are granted access to the member functions and variables of the class as if they were part of the class itself.

```
class MyClass {
private:
    int privateVar;

public:
    friend class FriendClass; // Declaration of friend class
};
```

3. Accessing Private Members:

- Friend functions and friend classes can access private and protected members of the class to which they are declared as friends.
- They are allowed to read and modify the private members of the class without using member functions or access specifiers.

```
class MyClass {
private:
    int privateVar;

public:
    friend void friendFunction(MyClass obj) {
        obj.privateVar = 10; // Accessing private member
    }
};
```

Example:

Here's an example demonstrating the usage of friend function to access private members of a class:

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

class MyClass {
private:
    int privateVar;

public:
```

```
    MyClass(int value) {
        privateVar = value;
    }

    friend void friendFunction(MyClass obj);
};

void friendFunction(MyClass obj) {
    cout << "Value of privateVar: " << obj.privateVar << endl;
}

int main() {
    MyClass obj(5);
    friendFunction(obj); // Output: Value of privateVar: 5
    return 0;
}
```

Merits of Friend Functions:

1. Access to Private Members: Friend functions can access private and protected members of the class, which allows for greater flexibility in function implementation.
2. Separation of Concerns: Friend functions can be used to separate the functionality that requires access to the internals of a class without making those functions members of the class.
3. Operator Overloading: Friend functions are often used for operator overloading where the left-hand operand is not an object of the class.

Demerits of Friend Functions:

1. Breaks Encapsulation: Friend functions violate the encapsulation principle by allowing external functions to access private and protected members of a class.
2. Maintenance Complexity: With increased access to a class's internals, the use of friend functions can lead to more complex and harder-to-maintain code.
3. Tight Coupling: Friend functions can create tight coupling between the class and the external function, making changes to the class more impactful on external code.

2.7 Empty Classes

- An empty class is defined as a class that contains no data members or member functions, or it contains only inline or default-constructed members.
- Size: An empty class has a size of at least one byte in C++ to ensure that distinct objects of the class have unique addresses in memory.
- **Syntax:**

```
class EmptyClass {
    // No member variables or member functions
};
```

Example:

```
// Empty class declaration
class Marker {};

int main() {
    Marker marker; // Creating an object of the empty class
    return 0;
}
```

Purpose and Usage:

- Marker Classes: Empty classes are often used as marker classes to convey specific information about the type or purpose of objects or functions.
- Compile-Time Checks: They can be utilized to trigger certain behaviors or compile-time checks when used as template parameters or function arguments.
- Polymorphic Behavior: Empty classes can be used as base classes to enable polymorphic behavior through inheritance, allowing derived classes to override member functions.

2.8 Nested Classes

In C++, a class can be defined within another class, known as a nested class. Nested classes have access to the private members of the enclosing class and can be used to encapsulate related functionality within the scope of the enclosing class.

Example:

```
class Outer {
private:
    int privateVar;

public:
    // Nested class declaration
    class Inner {
    public:
        void innerMethod() {
            // Accessing private member of the outer class
            cout << "Accessing privateVar from inner class: " <<
privateVar << endl;
        }
    };
};
```

Purpose and Usage:

- Encapsulation: Nested classes help in encapsulating related functionality within the scope of the enclosing class, reducing namespace pollution and improving code organization.

- Access to Enclosing Class Members: Nested classes have access to the private members of the enclosing class, allowing them to manipulate the state of the enclosing object.
- Information Hiding: They can be used to hide implementation details and provide a cleaner interface to the users of the enclosing class.

2.9 Local Classes

Local classes in C++ are classes that are defined within the scope of a function or a block. Local classes have limited local scope and are accessible only within the function or block in which they are defined. Local classes can be used to encapsulate functionality that is only relevant within a specific scope.

Example:

```
void myFunction() {
    // Local class definition
    class LocalClass {
    public:
        void localMethod() {
            cout << "Inside local method" << endl;
        }
    };

    LocalClass obj;
    obj.localMethod(); // Accessing method of local class
}
```

Purpose and Usage:

- Limited Scope: Local classes have a limited scope and are only accessible within the block or function where they are defined, reducing namespace pollution.
- Encapsulation: They can encapsulate functionality that is only relevant within a specific scope, improving code organization and modularity.
- Information Hiding: Local classes can hide implementation details and provide a cleaner interface to the surrounding code.

2.10 Abstract Classes

Abstract classes in C++ are classes that cannot be instantiated on their own and typically contain at least one pure virtual functions. They serve as base classes or blueprints for other derived classes and define an interface that derived classes must implement.
Pure virtual functions are declared with the virtual keyword and have no implementation (no function body).

Syntax:

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

```
};
```

Example:

```
class Shape {
public:
    virtual double area() const = 0; // Pure virtual function
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14159 * radius * radius;
    }
};
```

Purpose and Usage:

- Defining Interfaces: Abstract classes define interfaces for derived classes, specifying a set of methods that must be implemented.
- Enforcing Polymorphism: They provide a way to achieve runtime polymorphism through dynamic binding and function overriding.
- Providing Frameworks: Abstract classes can be used to define frameworks or templates for similar types of objects, providing a common structure and behavior.f

2.11 Container Classes

Container classes in C++ are classes that are used to store and manage collections of other objects, known as elements, in an organized manner. They provide various methods for accessing, inserting, removing, and manipulating elements within the container. Standard template library (STL) provides a variety of container classes, such as vectors, lists, sets, and maps.

Example of Container Classes:

1. Vector: A dynamic array that can resize itself automatically.
2. List: A doubly linked list that allows for efficient insertion and deletion operations.
3. Stack: A Last-In-First-Out (LIFO) data structure that allows for push and pop operations.
4. Queue: A First-In-First-Out (FIFO) data structure that allows for enqueue and dequeue operations.
5. Map: An associative container that stores key-value pairs and allows for efficient retrieval of values based on keys.

6. Set: A container that stores unique elements in a sorted order.

Example (Using std::vector):

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

int main() {
    // Creating a vector container
    vector<int> numbers;

    // Adding elements to the vector
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);

    // Accessing elements of the vector
    for (int num : numbers) {
        cout << num << " ";
    }
    return 0;
}
```

Purpose and Usage:

- Data Management: Container classes provide efficient ways to store and manage collections of objects, facilitating data organization and manipulation.
- Dynamic Storage: They dynamically resize to accommodate varying numbers of elements, allowing for flexible storage of data.
- Standardized Interfaces: Container classes in the STL provide standardized interfaces and algorithms for working with collections, promoting code reusability and interoperability.

2.12 Bit Field and Classes

Bit fields in C++ are used to represent and manipulate groups of bits within a data structure, allowing for efficient use of memory and bitwise operations. They are typically used within classes to pack multiple boolean or integer flags into a single storage unit.

- **Definition:** Bit fields are declared within a class using integer data types, specifying the number of bits to allocate for each field.
- **Purpose:** Bit fields are used to conserve memory by efficiently packing multiple boolean flags or integer values into a single storage unit, reducing memory overhead.
- **Usage:**

- Bit fields are accessed and manipulated using bitwise operators such as bitwise AND (&), bitwise OR (|), bitwise XOR (^), and bitwise shifts (<< and >>).
- They are commonly used to represent sets of boolean flags, where each flag corresponds to a single bit within the bit field.

- **Syntax:**

```
class BitFieldClass {
public:
    unsigned int flag1 : 1; // Bit field with 1 bit
    unsigned int flag2 : 1; // Bit field with 1 bit
    unsigned int value : 8; // Bit field with 8 bits
};
```

- **Example:**

```
class Permissions {
public:
    unsigned int read : 1;    // Bit field for read permission
    unsigned int write : 1;   // Bit field for write permission
    unsigned int execute : 1; // Bit field for execute permission
};

int main() {
    Permissions file1;
    file1.read = 1;
    file1.write = 0;
    file1.execute = 1;

    // Check if read permission is granted
    if (file1.read) {
        std::cout << "Read permission granted." << std::endl;
    }

    return 0;
}
```

Purpose and Usage:

- Memory Efficiency: Bit fields allow for efficient use of memory by packing multiple variables into a single storage unit, reducing memory overhead.
- Compact Representation: They provide a compact representation of data, especially for flags, status bits, or other binary values.
- Improved Readability: Bit fields can improve code readability by providing a concise way to define and access binary data within a class.

Unit 2: Inheritance and Pointers

Syllabus :
Inheritance: Introduction, defining derived classes, forms of inheritance, ambiguity in multiple and multipath inheritance, virtual base class, object slicing, overriding member functions, object composition and delegation, order of execution of constructors and destructors.
Pointers and Dynamic Memory Management: Declaring and initializing pointers, accessing data through pointers, pointer arithmetic, memory allocation (static and dynamic), dynamic memory management using new and delete operators, pointer to an object, this pointer, pointer related problems - dangling/wild pointers, null pointer assignment, memory leak and allocation failures.

Section 1: Inheritance

1.1 Introduction to Inheritance

1. Definition of Inheritance

- Inheritance is a fundamental concept in object-oriented programming (OOP) by which a new class (derived class) can inherit properties and behaviors from an existing class (base class).
- The derived class inherits the attributes and methods of the base class, enabling code reusability and establishing a hierarchical relationship between classes.
- The class that is being inherited from is known as the "parent class" or "base class" or "superclass", and the class that inherits from the base class is known as the “child class” or "derived class" or "subclass".
- The child class will inherit all the public and protected properties (attributes) and methods from the parent class. In addition, it can have its own properties and methods,this is called inheritance.

- Example: A Car class may inherit from a Vehicle class. Here, Car is a specific type of Vehicle.

2. Purpose and Benefits

1. Code Reusability: Inheritance allows classes to inherit attributes and methods from other classes, reducing redundancy and promoting code reuse.
2. Extensibility: Inheritance enables the addition of new features to a class without modifying its existing structure, thus facilitating the extension of functionality.
3. Relationship Representation: Inheritance models real-world relationships making the code more intuitive and reflective of real-world scenarios.
4. Specialization: Inheritance allows for the creation of specialized classes (derived classes) that inherit common features from a more general class (base class) and add their own unique functionalities.
5. Transitivity: If class B inherits from class A, and class C inherits from class B, then class C automatically inherits properties and behaviors from class A.
6. Hierarchical Organization: Inheritance enables the creation of a hierarchical organization of classes, where classes can be organized into a hierarchy based on their relationships.
7. Encapsulation: Inheritance encourages encapsulation, as it promotes the creation of well-defined, modular classes with clear boundaries.
8. Polymorphism: Inheritance supports polymorphism, which allows objects of derived classes to be treated as objects of their base class.

1.2 Defining Derived Classes

1. Syntax and Structure:

In C++, a derived class is defined using the following syntax:

```
class DerivedClassName : accessSpecifier BaseClassName {
    // Member variables and functions
};
```

- DerivedClassName: This is the name of the derived class.
- accessSpecifier: Specifies the access level for the base class members inherited by the derived class. It can be one of three access specifiers: public, protected, or private.
- BaseClassName: This is the name of the base class from which the derived class inherits.

Example:

```
// Base class
class Base {
public:
    int baseVar;
    void baseMethod() {
        // Implementation
    }
};

// Derived class inheriting from Base publicly
```

```
class Derived : public Base {
public:
    int derivedVar;
    void derivedMethod() {
        // Implementation
    }
};
```

2. Relationship Between Base and Derived Classes:

- **Inheritance Relationship:**
  - A derived class inherits attributes and behaviors (member variables and functions) from its base class.
  - The derived class can access the public and protected members of the base class.
  - The private members of the base class are not accessible directly by the derived class; however, they can be accessed through public or protected member functions of the base class.
- **Relationship Type:**
  - The relationship between a base class and a derived class is an "is-a" relationship.
  - For example, in the above code, Derived "is-a" Base. This implies that an object of the derived class can be treated as an object of the base class.
- **Access Control:**
  - The public, protected, and private access specifiers determine how the members of the base class are accessed by the derived class:
    - public: The public members of the base class remain public in the derived class.
    - protected: The protected members of the base class remain protected in the derived class.
    - private: The private members of the base class remain inaccessible in the derived class.
  - The choice of access specifier in the derived class declaration determines the level of visibility for the inherited members.

Example:

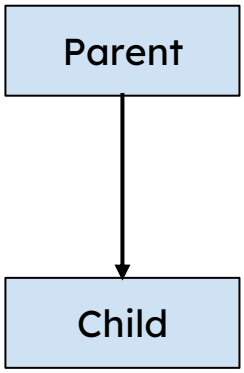
```
// Base class
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};
```

```
// Derived class inheriting publicly
class DerivedPublic : public Base {
    // Accessible: publicVar, protectedVar
    // Inaccessible: privateVar
};

// Derived class inheriting protectedly
class DerivedProtected : protected Base {
    // Accessible: protectedVar
    // Inaccessible: publicVar, privateVar
};

// Derived class inheriting privately
class DerivedPrivate : private Base {
    // Accessible: None (all members are private)
};
```

1.3 Forms of Inheritance

1. Single Inheritance:

- In single inheritance, a derived class inherits from only one base class.
- It forms a simple one-to-one relationship between classes.
- It is like Father -> Child relationship.
- Example:

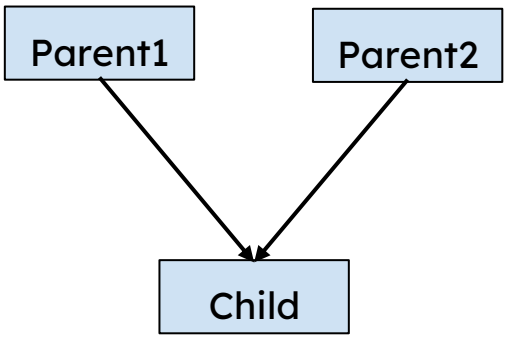



```
#include <iostream>
using namespace std;

// Base class (Parent)
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Derived class (Child) inheriting from Parent
class Child : public Parent {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

int main() {
    Child obj;
```

```
    obj.childMethod(); // Output: Method of Child Class
    obj.parentMethod(); // Output: Method of Parent Class
    return 0;
}
```

2. Multiple Inheritance:

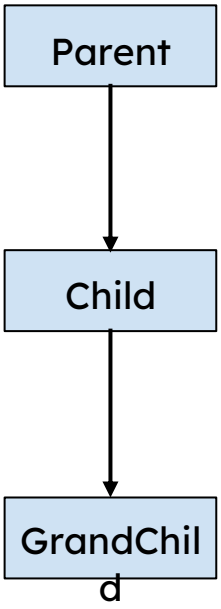
- In multiple inheritance, a derived class inherits from multiple base classes.
- It allows a class to inherit attributes and behaviors from more than one parent class.
- It is like Mother & Father -> Child relationship.
- Example:




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

// Parent1 class
class Parent1 {
public:
    void parent1Method() {
        cout << "Method of Parent1 Class" << endl;
    }
};

// Parent2 class
class Parent2 {
public:
    void parent2Method() {
        cout << "Method of Parent2 Class" << endl;
    }
};

// Child class inheriting from both Parent1 and Parent2
class Child : public Parent1, public Parent2 {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

int main() {
    Child obj;
    obj.childMethod(); // Output: Method of Child Class
    obj.parent1Method(); // Output: Method of Parent1 Class
    obj.parent2Method(); // Output: Method of Parent2 Class
    return 0;
}
```

3. Multilevel Inheritance:

- In multilevel inheritance, a derived class becomes the base class for another derived class, forming a chain of inheritance.
- It allows for creating a hierarchy of classes with each level adding additional features.
- It is like Father -> Child -> Grandchild relationship.



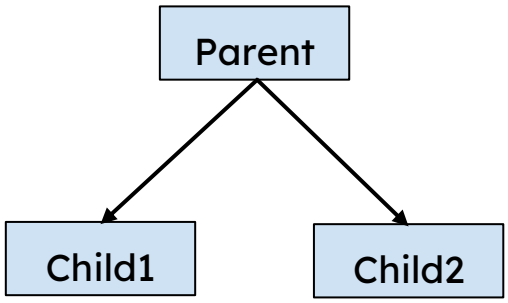
- Example:

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

// Parent class
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Child class inheriting from Parent
class Child : public Parent {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

// GrandChild class inheriting from Child
class GrandChild : public Child {
public:
    void grandChildMethod() {
        cout << "Method of Grand Child Class" << endl;
    }
};

int main() {
    GrandChild obj;
    obj.grandChildMethod(); // Output: Method of Grand Child Class
    obj.childMethod(); // Output: Method of Child Class
    obj.parentMethod(); // Output: Method of Parent Class
    return 0;
}
```

4. Hierarchical Inheritance:

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- It allows for creating a hierarchy of classes with a common base class.

- It is like Father -> Son & Daughter relationship.
- Example:

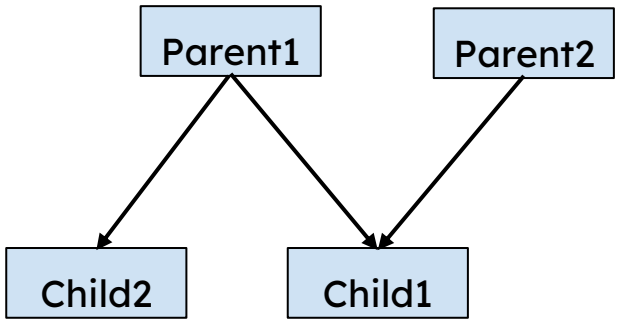



```
#include <iostream>
using namespace std;

// Parent class
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Child1 class inheriting from Parent
class Child1 : public Parent {
public:
    void child1Method() {
        cout << "Method of Child1 Class" << endl;
    }
};

// Child2 class inheriting from Parent
class Child2 : public Parent {
public:
    void child2Method() {
        cout << "Method of Child2 Class" << endl;
    }
};

int main() {
    Child1 obj1;
    obj1.child1Method(); // Output: Method of Child1 Class
    obj1.parentMethod(); // Output: Method of Parent Class

    Child2 obj2;
    obj2.child2Method(); // Output: Method of Child2 Class
    obj2.parentMethod(); // Output: Method of Parent Class

    return 0;
}
```

5. Hybrid Inheritance:

- Hybrid inheritance is a combination of multiple types of inheritance, such as single, multiple, multilevel or hierarchical inheritance.

- It allows for creating complex class hierarchies by combining features of different types of inheritance.
- Example:

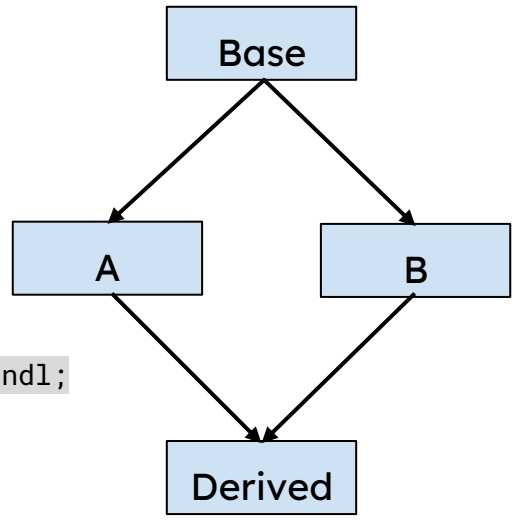



```
#include <iostream>
using namespace std;

// Parent1 class
class Parent1 {
public:
    void parent1Method() {
        cout << "Method of Parent1 Class" << endl;
    }
};

// Parent2 class
class Parent2 {
public:
    void parent2Method() {
        cout << "Method of Parent2 Class" << endl;
    }
};

// Child1 class inheriting from both Parent1 and Parent2
class Child1 : public Parent1, public Parent2 {
public:
    void child1Method() {
        cout << "Method of Child1 Class" << endl;
    }
};

// Child2 class inheriting from Parent1
class Child2 : public Parent1 {
public:
    void child2Method() {
        cout << "Method of Child2 Class" << endl;
    }
};

int main() {
    Child1 obj1;
    obj1.child1Method(); // Output: Method of Child1 Class
    obj1.parent1Method(); // Output: Method of Parent1 Class
    obj1.parent2Method(); // Output: Method of Parent2 Class

    Child2 obj2;
```

```
    obj2.child2Method(); // Output: Method of Child2 Class
    obj2.parent1Method(); // Output: Method of Parent1 Class

    return 0;
}
```

6. Multipath Inheritance:

- Multipath inheritance is a form of hybrid inheritance which occurs when there are multiple paths to reach a common base class through intermediate classes.
- This can happen when a class inherits from another class that indirectly inherits from the same base class.
- Example:




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

// Base class
class Base {
public:
    void baseMethod() {
        cout << "Method of Base Class" << endl;
    }
};

// Intermediate class A
class A : public Base {
public:
    void methodA() {
        cout << "Method of Class A" << endl;
    }
};

// Intermediate class B
class B : public Base {
public:
    void methodB() {
        cout << "Method of Class B" << endl;
    }
};

// Derived class inheriting from both A and B
class Derived : public A, public B {
public:
    void derivedMethod() {
        cout << "Method of Derived Class" << endl;
    }
```

```
};

int main() {
    Derived obj;
    // obj.baseMethod();  // Compile-time error due to ambiguity

    // To resolve ambiguity, explicitly call the base method through a
specific path using scope resolution operator
    obj.A::baseMethod();  // Output: Method of Base Class
    obj.B::baseMethod();  // Output: Method of Base Class
    obj.methodA();        // Output: Method of Class A
    obj.methodB();        // Output: Method of Class B
    obj.derivedMethod();  // Output: Method of Derived Class
    return 0;
}
```

In the above example, the Derived class inherits from both A and B, which in turn inherit from Base. This creates a multipath inheritance situation where there are multiple paths to reach the Base class from the Derived class.

1.4 Ambiguity in Multiple and Multipath Inheritance

Ambiguity in inheritance occurs when the derived class has more than one path to a base class. This situation can arise in both multiple inheritance and multipath inheritance.

1. **Multiple Inheritance:** Ambiguity arises when multiple base classes have methods with the same name. It can be resolved using the scope resolution operator.
2. **Multipath Inheritance:** Ambiguity arises due to multiple paths to a common base class. It can be resolved using the scope resolution operator or using virtual inheritance, ensuring only one instance of the base class is shared.

1. Ambiguity in Multiple Inheritance

In multiple inheritance, a derived class inherits from multiple base classes. Ambiguity occurs when multiple base classes have methods with the same name. The derived class does not know which method to inherit, leading to ambiguity.

Example:

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

// Base class 1
class Base1 {
public:
    void display() {
        cout << "Display from Base1" << endl;
    }
```

```
};

// Base class 2
class Base2 {
public:
    void display() {
        cout << "Display from Base2" << endl;
    }
};

// Derived class inheriting from both Base1 and Base2
class Derived : public Base1, public Base2 {
public:
    void show() {
        cout << "Show from Derived" << endl;
    }
};

int main() {
    Derived obj;
    obj.show();          // Output: Show from Derived

    // obj.display();    // Error: request for member 'display' is ambiguous

    // Resolving ambiguity
    obj.Base1::display();  // Output: Display from Base1
    obj.Base2::display();  // Output: Display from Base2

    return 0;
}
```

2. Ambiguity in Multipath Inheritance

Multipath inheritance occurs when a derived class inherits from two or more classes that have a common base class, creating multiple paths to the base class. This results in two copies of the base class members being inherited by the derived class, causing ambiguity.

Example without Virtual Inheritance:

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

// Base class
class Base {
public:
    void display() {
        cout << "Display from Base" << endl;
```

```
    }
};

// Intermediate class A
class A : public Base {
public:
    void displayA() {
        cout << "Display from A" << endl;
    }
};

// Intermediate class B
class B : public Base {
public:
    void displayB() {
        cout << "Display from B" << endl;
    }
};

// Derived class inheriting from both A and B
class Derived : public A, public B {
public:
    void show() {
        cout << "Show from Derived" << endl;
    }
};

int main() {
    Derived obj;
    obj.show();          // Output: Show from Derived
    // obj.display();    // Error: request for member 'display' is ambiguous

    // Resolving ambiguity
    obj.A::display();  // Output: Display from Base (via A)
    obj.B::display();  // Output: Display from Base (via B)

    return 0;
}
```

Resolving Ambiguity with Virtual Inheritance

To resolve ambiguity in multipath inheritance, virtual inheritance can be used. This ensures that only one instance of the base class is shared by all derived classes.

Example with Virtual Inheritance:

```
#include <iostream>
```

```cpp
using namespace std;

// Base class
class Base {
public:
    void display() {
        cout << "Display from Base" << endl;
    }
};

// Intermediate class A with virtual inheritance
class A : virtual public Base {
public:
    void displayA() {
        cout << "Display from A" << endl;
    }
};

// Intermediate class B with virtual inheritance
class B : virtual public Base {
public:
    void displayB() {
        cout << "Display from B" << endl;
    }
};

// Derived class inheriting from both A and B
class Derived : public A, public B {
public:
    void show() {
        cout << "Show from Derived" << endl;
    }
};

int main() {
    Derived obj;
    obj.show();         // Output: Show from Derived
    obj.display();      // Output: Display from Base

    return 0;
}
```

1.5 Virtual Base Class

A virtual base class in C++ is a mechanism used to solve the diamond problem in multiple inheritance scenarios. The diamond problem occurs when a derived class inherits from two base classes that both inherit from a common base class. Without virtual inheritance, the derived class would have two copies of the common base class, leading to ambiguity and redundancy.

1. Use cases and Benefits

Use Cases:

- Diamond Inheritance Problem: When a derived class inherits from two base classes that both inherit from the same base class.
- Avoid Redundancy: Ensuring that there is only one instance of a base class in a complex inheritance hierarchy.
- Memory Efficiency: Reducing the memory overhead by preventing multiple copies of the base class.

Benefits:

- Eliminates Ambiguity: Ensures that there is a single, unique instance of the base class, eliminating ambiguity when accessing base class members.
- Consistent Data Management: Simplifies data management by maintaining a single instance of base class data members.
- Cleaner and More Maintainable Code: Makes the inheritance hierarchy more manageable and understandable.

2. Syntax and Implementation

To declare a virtual base class, the virtual keyword is used in the inheritance declaration. Below is an example illustrating the diamond problem and how virtual inheritance solves it.

Example without Virtual Inheritance (Diamond Problem):

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

// Base class
class Base {
public:
    void baseMethod() {
        cout << "Method of Base Class" << endl;
    }
};

// Intermediate class A
```

```
class A : public Base {
public:
    void methodA() {
        cout << "Method of Class A" << endl;
    }
};

// Intermediate class B
class B : public Base {
public:
    void methodB() {
        cout << "Method of Class B" << endl;
    }
};

// Derived class inheriting from both A and B
class Derived : public A, public B {
public:
    void derivedMethod() {
        cout << "Method of Derived Class" << endl;
    }
};

int main() {
    Derived obj;
    obj.derivedMethod();  // Output: Method of Derived Class
    // obj.baseMethod();  // Error: request for member 'baseMethod' is
ambiguous

    // Resolving ambiguity
    obj.A::baseMethod();  // Output: Method of Base Class (via A)
    obj.B::baseMethod();  // Output: Method of Base Class (via B)

    return 0;
}
```

In the above example, the Derived class has two copies of Base class, one from A and one from B, leading to ambiguity.

Example with Virtual Inheritance:

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

// Base class
class Base {
public:
```

```
    void baseMethod() {
        cout << "Method of Base Class" << endl;
    }
};

// Intermediate class A with virtual inheritance
class A : virtual public Base {
public:
    void methodA() {
        cout << "Method of Class A" << endl;
    }
};

// Intermediate class B with virtual inheritance
class B : virtual public Base {
public:
    void methodB() {
        cout << "Method of Class B" << endl;
    }
};

// Derived class inheriting from both A and B
class Derived : public A, public B {
public:
    void derivedMethod() {
        cout << "Method of Derived Class" << endl;
    }
};

int main() {
    Derived obj;
    obj.derivedMethod();  // Output: Method of Derived Class
    obj.baseMethod();     // Output: Method of Base Class

    return 0;
}
```

In this example, A and B both virtually inherit from Base. As a result, Derived only has one copy of Base, and there is no ambiguity when calling baseMethod.

1.6 Object Slicing

1. Definition and Examples:

Object slicing occurs when a derived class object is assigned to a base class object. In this process, the derived part of the object is "sliced off," leaving only the base class part. This

means that any member variables or methods defined in the derived class are not accessible through the base class object, and their values are lost.

Examples:

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

// Base class
class Base {
public:
    int baseValue;

    Base(int val) : baseValue(val) {}

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

// Derived class
class Derived : public Base {
public:
    int derivedValue;

    Derived(int baseVal, int derivedVal) : Base(baseVal),
derivedValue(derivedVal) {}

    void display() {
        cout << "Base value: " << baseValue << ", Derived value: " <<
derivedValue << endl;
    }
};

int main() {
    Derived derivedObj(10, 20);
    Base baseObj = derivedObj; // Object slicing occurs here

    baseObj.display();  // Output: Base value: 10
    derivedObj.display(); // Output: Base value: 10, Derived value: 20

    return 0;
}
```

In this example:

- Derived class inherits from Base class.
- Derived class has an additional member variable derivedValue.

- When derivedObj (of type Derived) is assigned to baseObj (of type Base), the derivedValue part of derivedObj is sliced off.
- The display method of Base class only shows baseValue because baseObj is of type Base.

2. Implications and Solutions:

Implications:

- Loss of Data: Any member variables of the derived class are lost when slicing occurs.
- Incorrect Method Calls: If the base class and derived class both have a method with the same name, the base class method will be called, leading to potentially incorrect behavior.
- Runtime Polymorphism Not Achieved: Slicing prevents the use of derived class functionalities through base class pointers or references.

Solutions:

1. Avoid Object Slicing:
    - Avoid assigning objects of derived classes to objects of their base classes to prevent object slicing.
    - Use pointers or references to base classes instead of objects.
2. Use Pointers or References:
    - Use pointers or references to base classes when dealing with polymorphism.
    - This allows the derived class-specific information to be preserved.

    Example of using pointers:

```
int main() {
    Derived derivedObj(10, 20);
    Base* basePtr = &derivedObj; // Using pointer to Base class
    basePtr->print(); // Output: Base Data: 10, Derived Data: 20
    return 0;
}
```

3. Copy Constructor or Assignment Operator:
    - If object slicing is unavoidable, provide a copy constructor or assignment operator in the base class to handle the conversion properly.

```
class Base {
public:
    int baseData;
    Base(const Base& other) : baseData(other.baseData) {} // Copy
constructor
    Base& operator=(const Base& other) {
        if (this != &other) {
            baseData = other.baseData;
        }
        return *this;
    }
};
```

By understanding the implications of object slicing and employing appropriate solutions, you can ensure correct behavior and prevent loss of information when working with inheritance and polymorphism in C++.

1.7 Overriding Member Functions

1. Concept of Function Overriding:

- Function overriding in C++ occurs when a derived class provides a specific implementation for a function that is already defined in its base class. The function in the derived class should have the same name, return type, and parameters as the function in the base class.
- Function overriding is a way to achieve runtime polymorphism. When you call an overridden function using a base class pointer or reference, the derived class's version of the function is called, provided that the base class function is declared virtual.

Rules for Overriding

1. Same Signature: The function in the derived class must have the same name, return type, and parameter list as the function in the base class.
2. Virtual Keyword: The base class function should be marked with the virtual keyword to enable overriding.
3. Override Keyword: It is good practice to use the override keyword in the derived class to indicate that the function is intended to override a base class function. This helps the compiler catch errors if the function does not actually override any base class function.

2. Syntax and Examples:

Syntax:

```
class Base {
public:
    virtual void functionName() {
        // Base class function implementation
    }
};

class Derived : public Base {
public:
    void functionName() override {
        // Derived class function implementation (override)
    }
};
```

Example:

```
#include <iostream>
```

```cpp
using namespace std;

// Base class
class Base {
public:
    // Virtual function
    virtual void display() {
        cout << "Display method of Base class" << endl;
    }
};

// Derived class
class Derived : public Base {
public:
    // Overriding function
    void display() override {
        cout << "Display method of Derived class" << endl;
    }
};

int main() {
    Base* basePtr;
    Derived derivedObj;

    basePtr = &derivedObj;

    // Call overridden method
    basePtr->display(); // Output: Display method of Derived class

    return 0;
}
```

In this example:

- Base class has a virtual function display().
- Derived class overrides the display() function.
- When calling display() through a Base class pointer that points to a Derived class object, the Derived class's display() function is invoked.

1.8 Object Composition and Delegation

1. Object Composition

Object Composition is a design principle where complex objects are created by combining simpler objects. This is often referred to as a "has-a" relationship. In composition, an object is composed of one or more objects from other classes, making it a more flexible and modular approach to building complex systems.

Example

Here's a simple example of object composition:

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

// Engine class
class Engine {
public:
    void start() {
        cout << "Engine starts" << endl;
    }
};

// Car class composed of Engine
class Car {
private:
    Engine engine;  // Car "has-a" Engine
public:
    void start() {
        engine.start();  // Delegate the call to the Engine object
        cout << "Car starts" << endl;
    }
};

int main() {
    Car myCar;
    myCar.start();
    // Output:
    // Engine starts
    // Car starts

    return 0;
}
```

In this example:

- The Car class has an Engine object.
- The Car class delegates the starting functionality to the Engine object by calling its start() method.

2. Delegation

Delegation is a design pattern where an object handles a request by passing it to a second "delegate" object. The delegate object provides a means to share functionality between objects. This allows for more flexible and reusable code by promoting composition over inheritance.

Example

Here's an example of delegation:

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

// Printer class
class Printer {
public:
    void print(const string &text) {
        cout << text << endl;
    }
};

// Document class using Printer for printing
class Document {
private:
    Printer printer;  // Delegating printing responsibility to Printer
public:
    void print(const string &text) {
        printer.print(text);  // Delegation
    }
};

int main() {
    Document myDocument;
    myDocument.print("Hello World!");
    // Output:
    // Hello World!

    return 0;
}
```

In this example:

- The Document class delegates the printing functionality to the Printer class.
- This separation allows the Printer class to handle the printing logic, making the Document class simpler and more focused on its core responsibilities.

Differences Between Composition and Inheritance

1. **Composition (Has-a Relationship):**
    - Combines objects to build complex systems.
    - More flexible and allows for changing behavior at runtime.
    - Promotes code reuse and encapsulation.
    - Example: A Car has an Engine.
2. **Inheritance (Is-a Relationship):**
    - Creates a new class based on an existing class.

- Establishes a parent-child relationship.
- Allows for extending and modifying behavior of the base class.
- Example: A Dog is an Animal.

Advantages of Composition and Delegation

- Flexibility: Objects can be composed and recomposed easily.
- Encapsulation: Internal details of composed objects are hidden.
- Reuse: Common functionality can be reused through composition.
- Modularity: Systems can be built in a modular fashion, promoting maintainability and scalability.
- Avoids Fragile Base Class Problem: Changes in the base class in inheritance can affect all derived classes, while composition mitigates this risk.

1.9 Order of Execution of Constructors and Destructors

In C++, the order of execution of constructors and destructors depends on the inheritance hierarchy and the construction/destruction sequence of objects.

Order of Execution:

- Constructors execute in the order of inheritance (base to derived).
- Destructors execute in the reverse order of inheritance (derived to base).

Constructors:

1. *Base Class Constructors:* Constructors of base classes are executed first, starting from the topmost base class and proceeding down the inheritance hierarchy.
2. *Member Object Constructors:* Constructors for member objects of the class are then executed, in the order they are declared within the class definition.
3. *Derived Class Constructor Body:* Finally, the constructor body of the derived class is executed.

Destructors:

The order of destruction is essentially the reverse of construction:

1. *Derived Class Destructor Body:* The destructor body of the derived class is executed first.
2. *Member Object Destructors:* Destructors for member objects are then called, in the reverse order of their construction.
3. *Base Class Destructors:* Destructors for base classes are called last, starting from the most derived class and proceeding up the inheritance hierarchy.

Example:

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

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

```
    ~Base() { cout << "Base Destructor" << endl; }
};

class Member {
public:
    Member() { cout << "Member Constructor" << endl; }
    ~Member() { cout << "Member Destructor" << endl; }
};

class Derived : public Base {
private:
    Member member;

public:
    Derived() { cout << "Derived Constructor" << endl; }
    ~Derived() { cout << "Derived Destructor" << endl; }
};

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

Output:

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

In the example above:

- Constructors are called in the order: Base, Member, Derived.
- Destructors are called in the order: Derived, Member, Base.

Section 2: Pointers and Dynamic Memory Management

2.1 Declaring and Initializing Pointers

1. Pointer Syntax and Declaration:

- Pointers are variables that store memory addresses of another variable.
- Syntax: `data_type *pointer_name;`
- Example:
  ```
  int *ptr; // Declares a pointer to an integer
  ```

```
float *floatPtr; // Declares a pointer to a float
char *charPtr; // Declares a pointer to a character
```

2. Initialization Methods:

- Pointers can be initialized in several ways:

Direct Initialization:

- Assign the address of a variable to a pointer during declaration.
- Syntax: `data_type *pointer_name = &variable_name;`
- Example:

```
int num = 10;
int *ptr = &num; // Initializes ptr with the address of num
```

Assignment after Declaration:

- Declare a pointer first and then assign it a value (address) later.
- Syntax:

```
data_type *pointer_name;
pointer_name = &variable_name;
```

- Example:

```
int *ptr; // Declare pointer
int num = 20;
ptr = &num; // Assign address of num to ptr
```

Using new Operator:

- Dynamically allocate memory for a variable and assign its address to a pointer.
- Syntax: data_type *pointer_name = new data_type;
- Example:

```
int *ptr = new int; // Allocates memory for an integer dynamically
```

Example of Pointer Declaration and Initialization:

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

int main() {
    // Direct Initialization
    int num = 5;
    int *ptr1 = &num;

    // Assignment after Declaration
    float val = 3.14;
    float *ptr2;
    ptr2 = &val;

    // Using new Operator
    char *charPtr = new char;
```

```
    *charPtr = 'A'; // Assign value to dynamically allocated memory

    // Output
    cout << "Value of num: " << num << endl;
    cout << "Address of num: " << ptr1 << endl;
    cout << "Value of val: " << val << endl;
    cout << "Address of val: " << ptr2 << endl;
    cout << "Value of charPtr: " << *charPtr << endl;

    // Deallocating dynamically allocated memory
    delete charPtr;

    return 0;
}
```

Output:

```
Value of num: 5
Address of num: 0x7ffc7e4f81f4
Value of val: 3.14
Address of val: 0x7ffc7e4f81f8
Value of charPtr: A
```

2.2 Accessing Data through Pointers

1. Dereferencing Pointers:

Dereferencing a pointer means accessing the value stored at the memory address pointed to by the pointer.
Syntax: `*pointer_name;`

Example:

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

int main() {
    int num = 10;
    int *ptr = &num; // Pointer points to the address of num

    cout << "Value of num: " << *ptr << endl; // Dereferencing ptr to
access the value of num

    return 0;
}
```

Output:

```
Value of num: 10
```

2. Pointer Notation for Accessing Elements:

For arrays, pointer notation can be used to access array elements using pointers.

Syntax:

```
*(ptr + i)  // Equivalent to ptr[i]
```

Example:

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer points to the first element of the array

    // Accessing elements using pointer notation
    for (int i = 0; i < 5; ++i) {
        cout << *(ptr + i) << " "; // Equivalent to ptr[i]
    }
    cout << endl;

    return 0;
}
```

Output:

```
1 2 3 4 5
```

In this example, *(ptr + i) and ptr[i] are equivalent and both are used to access array elements using pointers.

2.3 Pointer Arithmetic

1. Increment and Decrement Operations:

Pointers can be incremented and decremented to move to the next or previous memory location.

Increment Operation:

```
ptr++; // Moves the pointer to the next memory location
```

Decrement Operation:

```
ptr--; // Moves the pointer to the previous memory location
```

Example:

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

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer points to the first element of the array

    cout << "Original pointer value: " << ptr << endl;

    ptr++; // Incrementing pointer
    cout << "After incrementing, pointer value: " << ptr << endl;

    ptr--; // Decrementing pointer
    cout << "After decrementing, pointer value: " << ptr << endl;

    return 0;
}
```

Output:

```
Original pointer value: 0x7ffe36a552a0
After incrementing, pointer value: 0x7ffe36a552a4
After decrementing, pointer value: 0x7ffe36a552a0
```

2. Arithmetic Operations on Pointers:

Arithmetic operations such as addition and subtraction can be performed on pointers.

Addition Operation:

```
ptr += n; // Moves the pointer n positions forward
```

Subtraction Operation:

```
ptr -= n; // Moves the pointer n positions backward
```

Example:

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer points to the first element of the array

    cout << "Original pointer value: " << ptr << endl;

    ptr += 2; // Moving pointer 2 positions forward
    cout << "After addition, pointer value: " << ptr << endl;

    ptr -= 1; // Moving pointer 1 position backward
    cout << "After subtraction, pointer value: " << ptr << endl;
```

```
    return 0;
}
```

Output:

```
Original pointer value: 0x7ffe88bfe9d0
After addition, pointer value: 0x7ffe88bfe9d8
After subtraction, pointer value: 0x7ffe88bfe9d4
```

In this example, ptr += 2 moves the pointer two positions forward, and ptr -= 1 moves the pointer one position backward.

2.4 Memory Allocation (Static and Dynamic)

Memory allocation refers to the process of reserving memory space for variables or objects during program execution. In C++, memory allocation can be categorized into two types: static memory allocation and dynamic memory allocation.

1. Static Memory Allocation:

In static memory allocation, memory is allocated at compile-time, and the size of memory is fixed throughout the program execution. Compile time refers to the period when the programming code (such as C++) is converted to the machine code (i.e. binary code).

Example:

```
int arr[5]; // Static allocation of an array of 5 integers
```

- Memory for arr is allocated on the stack.
- Memory size is fixed and determined during compile-time.
- Memory is automatically deallocated when the scope containing the variable ends.

2. Dynamic Memory Allocation:

In dynamic memory allocation, memory is allocated at runtime, and the size of memory can be determined during program execution. Runtime is the period of time when a program is actually running and occurs after compile time.

Dynamic Memory Allocation using new operator:

Single Object Allocation:

```
int *ptr = new int; // Allocates memory for a single integer dynamically
```

- Memory for ptr is allocated on the heap.
- Memory size can be determined during runtime.
- Memory needs to be explicitly deallocated using the delete operator to prevent memory leaks.

Array Allocation:

```
int *arr = new int[5]; // Allocates memory for an array of 5 integers dynamically
```

- Memory for arr is allocated on the heap.
- Size of the array is determined during runtime.
- Individual elements of the array can be accessed using pointer notation.

Deallocating Dynamically Allocated Memory:

Single Object Deallocation:

```
delete ptr; // Deallocates memory allocated for a single variable
```

Array Deallocation:

```
delete[] arr; // Deallocates memory allocated for an array
```

- The delete operator is used to deallocate memory previously allocated using the new operator.
- For arrays, the delete[] operator is used to deallocate memory allocated for the entire array.

Comparison:

- Static memory allocation is simpler and faster compared to dynamic memory allocation.
- Dynamic memory allocation provides flexibility in memory management but requires manual deallocation to prevent memory leaks.
- Static memory allocation is suitable for scenarios where memory size is known in advance, while dynamic memory allocation is preferred when the size of memory is determined during runtime or when memory needs to be allocated and deallocated dynamically.

2.5 Dynamic Memory Management using new and delete Operators

Dynamic memory management in C++ allows the allocation and deallocation of memory during runtime. This is useful for managing memory that cannot be determined at compile time. The new operator is used to allocate memory on the heap, and the delete operator is used to deallocate that memory.

1. Allocating Memory Dynamically:

Memory can be allocated dynamically using the new operator in C++.

Syntax:

```
data_type *pointer_name = new data_type;
```

Example:

```
int *ptr = new int; // Allocates memory for an integer dynamically
```

In the example above, memory is allocated on the heap to store an integer value, and the address of the allocated memory is assigned to the pointer ptr.

2. Releasing Memory using delete:

After dynamically allocating memory, it's essential to release it when it's no longer needed to prevent memory leaks.

Syntax:

```
delete pointer_name;
```

Example:

```
delete ptr; // Deallocates memory allocated for the integer pointed by ptr
```

In the example above, the delete operator is used to deallocate the memory allocated for the integer pointed to by ptr. After deletion, the memory is returned to the system for reuse.

Example:

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

int main() {
    // Dynamic memory allocation
    int *ptr = new int; // Allocates memory for an integer dynamically
    *ptr = 10; // Assigns a value to the dynamically allocated memory

    // Accessing and printing the dynamically allocated value
    cout << "Dynamically allocated value: " << *ptr << endl;

    // Deallocating dynamically allocated memory
    delete ptr;

    return 0;
}
```

In this example:

- Memory for an integer is allocated dynamically using the new operator.
- The value 10 is assigned to the dynamically allocated memory.
- The value stored in the dynamically allocated memory is printed.
- Finally, the dynamically allocated memory is deallocated using the delete operator.

Notes:

- Dynamically allocated memory remains allocated until explicitly deallocated using the delete operator.
- Failure to deallocate dynamically allocated memory can lead to memory leaks, causing the program to consume more memory than necessary.
- It's essential to deallocate dynamically allocated memory when it's no longer needed to ensure efficient memory usage.

2.6 Pointer to an Object

In C++, pointers can also be used to point to objects of classes. This allows for dynamic allocation of objects and provide flexibility in memory management.

1. Creating Pointers to Objects:

Pointers to objects are declared similarly to pointers to primitive data types.

Syntax:

```
ClassName *pointer_name = new ClassName;
```

Example:

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

// Class definition
class MyClass {
public:
    void display() {
        cout << "Inside MyClass" << endl;
    }
};

int main() {
    // Creating an object of MyClass dynamically
    MyClass *ptr = new MyClass;

    // Accessing member function using pointer
    ptr->display(); // Output: Inside MyClass

    // Deallocating dynamically allocated memory
    delete ptr;

    return 0;
}
```

In the example above:

- Memory for an object of MyClass is allocated dynamically using the new operator.
- A pointer ptr of type MyClass is used to point to the dynamically allocated object.
- The display() member function of the MyClass object is called using the pointer.
- Finally, the dynamically allocated memory is deallocated using the delete operator.

2. Accessing Object Members through Pointers:

Once a pointer is pointing to an object, its member functions and variables can be accessed using the arrow (->) operator.

Syntax:

```
pointer_name->member_function();
pointer_name->member_variable;
```

Example:

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

```cpp
// Class definition
class MyClass {
public:
    int data;

    void display() {
        cout << "Data: " << data << endl;
    }
};

int main() {
    // Creating an object of MyClass dynamically
    MyClass *ptr = new MyClass;

    // Accessing and modifying member variable using pointer
    ptr->data = 10;

    // Accessing member function using pointer
    ptr->display(); // Output: Data: 10

    // Deallocating dynamically allocated memory
    delete ptr;

    return 0;
}
```

In this example:

- A pointer ptr of type MyClass points to the dynamically allocated object of MyClass.
- The member variable data is accessed and modified using the pointer.
- The member function display() is called using the pointer to display the value of the member variable.
- Finally, the dynamically allocated memory is deallocated using the delete operator.

2.7 'this' Pointer

In C++, the this pointer is a special pointer that points to the current instance of the class. It is available as a keyword within non-static member functions of a class. It is used to distinguish member variable from parameter.

Key Characteristics of this Pointer:

1. **Points to the Current Object:** The this pointer points to the address of the object invoking the member function.
2. **Distinguishing Member Variables:** It helps distinguish between member variables and parameters with the same name.
3. **Returning the Current Object:** It can be used to return the current object from member functions.

Example of Using this Pointer:

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

class MyClass {
private:
    int value;

public:
    // Constructor with a parameter
    MyClass(int value) {
        // Using 'this' pointer to distinguish member variable from
parameter
        this->value = value;
    }

    // Method to display the value
    void display() {
        cout << "Value: " << this->value << endl;
    }
};

int main() {
    // Creating an object of MyClass
    MyClass obj(10);

    // Displaying the value using the object
    obj.display();  // Output: Value: 10

    return 0;
}
```

2.8 Pointer-Related Problems

Pointers are powerful features in C++ that provide flexibility and control over memory management. However, improper use of pointers can lead to several issues, including dangling pointers, wild pointers, null pointer assignments, memory leaks, and allocation failures.

1. Dangling Pointers

- A dangling pointer is a pointer that references a memory location that has been deallocated (freed). Accessing or modifying the memory location through a dangling pointer leads to undefined behavior.

Example:

```
int* ptr = new int(10); // Allocate memory
delete ptr; // Deallocate memory
// ptr is now a dangling pointer
cout << *ptr << endl; // Undefined behavior
```

Prevention:

- After deallocating memory, set the pointer to nullptr.
  ```
  delete ptr;
  ptr = nullptr;
  ```

2. Wild Pointers

- Wild pointers are uninitialized pointers that point to arbitrary memory locations. Accessing or modifying memory through wild pointers can lead to unpredictable behavior and program crashes.

Example:

```
int* ptr; // Uninitialized pointer
cout << *ptr << endl; // Undefined behavior
```

Prevention:

- Initialize pointers when they are declared.
  ```
  int* ptr = nullptr;
  ```

3. Null Pointer Assignment

- A null pointer is a pointer that points to nothing (typically initialized to nullptr). Dereferencing a null pointer results in undefined behavior and typically crashes the program.

Example:

```
int* ptr = nullptr; // Null pointer
cout << *ptr << endl; // Undefined behavior
```

Prevention:

- Always check if a pointer is nullptr before dereferencing it.
  ```
  if (ptr != nullptr) {
      cout << *ptr << endl;
  }
  ```

4. Memory Leak

- A memory leak occurs when memory is allocated but not deallocated properly, leading to a gradual increase in memory usage. This can eventually exhaust available memory, causing the program to crash or slow down.

Example:

```
for (int i = 0; i < 1000000; ++i) {
    int* ptr = new int(10); // Allocate memory
    // No delete operation, memory leak occurs
}
```

Prevention:

- Ensure that every new operation has a corresponding delete operation.
  `delete ptr;`

5. Allocation Failures

- Memory allocation failures occur when the system cannot allocate the requested memory, typically due to insufficient available memory.

Example:

```
int* ptr = new int[1000000000]; // Large allocation request
```

Prevention:

- Always check the return value of memory allocation functions and handle exceptions for new.

```
try {
    int* ptr = new int[1000000000];
} catch (bad_alloc& ex) {
    cout << "Memory allocation failed: " << ex.what() << endl;
}
```

Unit 3: Constructors, Destructors, Operator Overloading, Type Conversion, Virtual Functions, and Polymorphism

---

Syllabus :

Constructors and Destructors: Need for constructors and destructors, copy constructor, dynamic constructors, explicit constructors, destructors, constructors and destructors with static members, initializer lists.

Operator Overloading and Type Conversion: Overloading operators, rules for overloading operators, overloading of various operators, type conversion - basic type to class type, class type to basic type, class type to another class type.

Virtual functions & Polymorphism: Concept of binding - early binding and late binding, virtual functions, pure virtual functions, abstract clasess, virtual destructors.

Section 1 : Constructors and Destructors

Constructors and destructors are fundamental concepts in object-oriented programming languages like C++. Constructors are special member functions of a class that initialize objects of that class. On the other hand, destructors are used to clean up resources allocated by an object when it goes out of scope or is explicitly destroyed.

1.1 Constructors

Constructors in C++ are special member functions of a class that are automatically called when an object is created. They are primarily used for initializing the objects and setting up any required resources.
Constructors have the same name as the class and can be overloaded to accept different sets of parameters. It is declared in public section of the class. It has no return type so it can't return any value. It can't be inherited though derived classes can call a base class constructor. It can't be virtual.

Need for Constructors

- Initialization: Constructors are used to initialize an object's data members when it is created.
- Default Values: They provide a way to set default values for data members.
- Resource Allocation: They can allocate resources such as memory or file handles when an object is created.

Types of Constructors

1. **Default Constructor:** Initializes an object without any parameters.
2. **Parameterized Constructor:** Initializes an object with given parameters.
3. **Copy Constructor:** Initializes an object using another object of the same class.
4. **Dynamic Constructor:** Allocates memory dynamically for an object.
5. **Explicit Constructor:** Prevents implicit conversions to the class type.

A. Default Constructor

A default constructor is a constructor that does not take any arguments. It is automatically called when an object is created without providing any initialization values. If no constructor is defined in the class, the compiler generates a default constructor with an empty code and no parameter.

Example:

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

class MyClass {
```

```
public:
    int value;

    // Default constructor
    MyClass() {
        value = 0;
        cout << "Default constructor called" << endl;
    }

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

int main() {
    MyClass obj;    // Default constructor is called
    obj.display();  // Value: 0
    return 0;
}
```

B. Parameterized Constructor

A parameterized constructor is a constructor that takes one or more parameters or arguments. It allows the user to initialize objects with specific values at the time of creation.

Example:

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

class MyClass {
public:
    int value;

    // Parameterized constructor
    MyClass(int val) {
        value = val;
        cout << "Parameterized constructor called" << endl;
    }

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

int main() {
```

```
    MyClass obj(42);  // Parameterized constructor is called
    obj.display();    // Value: 42
    return 0;
}
```

C. Copy Constructor

A copy constructor is a special type of constructor that is used to copy data from one object to another. It is called automatically when a new object is created from an existing object, typically using the assignment operator or passing an object by value to a function. The copy constructor performs a deep copy of the object's data members.

Example:

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

class MyClass {
public:
    int value;

    // Parameterized constructor
    MyClass(int val) {
        value = val;
    }

    // Copy constructor
    MyClass(const MyClass &obj) {
        value = obj.value;
        cout << "Copy constructor called" << endl;
    }

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

int main() {
    MyClass obj1(42);     // Parameterized constructor is called
    MyClass obj2 = obj1;  // Copy constructor is called
    obj2.display();       // Value: 42
    return 0;
}
```

D. Dynamic Constructor

A dynamic constructor allocates memory dynamically using new or malloc. It is useful when the size of data members is not known at compile time and needs to be allocated at runtime.
Dynamic memory allocation in C++ is typically done using the new operator. While it's not a constructor itself, it's often used within constructors to allocate memory dynamically for data members of a class.

Example:

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

class MyClass {
private:
    int *ptr;
public:
    MyClass(int size) {
        ptr = new int[size];
        cout << "Dynamic constructor called" << endl;
    }

    ~MyClass() {
        delete[] ptr;
        cout << "Destructor called" << endl;
    }
};

int main() {
    MyClass obj(5); // Dynamic constructor is called
    return 0;
}
```

E. Explicit Constructor

An explicit constructor is used to prevent implicit conversions. It is declared with the explicit keyword, ensuring that the constructor cannot be used for implicit conversions and copy-initialization.

Example:

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

class MyClass {
private:
    int value;
public:
```

```
    // Explicit constructor
    explicit MyClass(int val) {
        value = val;
        cout << "Explicit constructor called with value " << value << endl;
    }
};

int main() {
    // MyClass obj = 10; // Error: Implicit conversion is not allowed
    MyClass obj(10);    // Explicit constructor is called
    return 0;
}
```

1.2 Destructors

Destructors are special member functions in C++ that are automatically called when an object goes out of scope or is explicitly deleted. They are used to release resources acquired by the object, such as dynamic memory, file handles, or network connections. A destructor has the same name as the class preceded by a tilde (~) and does not take any arguments.

Need for Destructors

- Resource Deallocation: The primary purpose of a destructor is to release any resources held by the object, such as dynamic memory allocated during the object's lifetime. This helps prevent memory leaks and ensures proper cleanup.
- Custom Cleanup: In addition to releasing resources, a destructor can perform any custom cleanup actions required by the class, such as closing files, releasing locks, or resetting state.

Example:

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

class MyClass {
public:
    // Constructor
    MyClass() {
        cout << "Constructor called" << endl;
    }

    // Destructor
    ~MyClass() {
        cout << "Destructor called" << endl;
    }
};
```

```
int main() {
    {
        MyClass obj; // Constructor called
    } // Destructor called

    return 0;
}
```

In the above example, the MyClass constructor is called when the object obj is created. As soon as the object goes out of scope (at the closing curly brace), the destructor is automatically called to clean up any resources held by the object.

Automatic vs. Manual Destruction:

1. **Automatic Destruction:** Objects with automatic storage duration (local variables) have their destructors automatically called when they go out of scope. The compiler ensures that destructors are invoked at the appropriate time.
2. **Manual Destruction:** Objects created dynamically (using new) must be explicitly deleted by the programmer to invoke their destructor and release memory. Failure to do so can result in memory leaks.

1.3 Constructors and Destructors with Static Members

Constructors and destructors can interact with static members of a class. Static members belong to the class itself rather than to individual objects, and they are shared among all objects of the class to maintain shared state or resources.

A. Constructors with Static Members

Constructors can initialize static members of a class. Since static members are shared among all instances of the class, their initialization typically occurs only once, before any objects of the class are created. Constructors can be used to set initial values for static members.

B. Destructors with Static Members

Destructors can also interact with static members of a class. However, it's essential to understand that destructors are called when the last object of the class goes out of scope or is explicitly deleted. Therefore, destructors can be used to perform cleanup actions involving static members.

Example with Static Members:

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

class Counter {
public:
    static int count;
```

```
    Counter() {
        count++; // Increment count in the constructor
    }

    ~Counter() {
        count--; // Decrement count in the destructor
    }
};

int Counter::count = 0; // Initialize static member count

int main() {
    Counter obj1; // Constructor called, count = 1
    Counter obj2; // Constructor called, count = 2

    cout << "Count: " << Counter::count << endl; // Output: 2

    return 0; // Destructor of obj2, Destructor of obj1
}
```

1.4 Initializer Lists

Initializer lists in C++ are a powerful feature that allow to initialize class members in a concise and efficient way.
Initializer lists in C++ are used in constructors to initialize data members of a class before the constructor body is executed. They provide a way to initialize class members directly within the constructor declaration, offering several benefits such as efficiency and flexibility.

Benefits of Using Initializer Lists:

1. Efficiency: Initialization is performed before the constructor body is executed, potentially avoiding unnecessary default initialization.
2. Const and Reference Members: Allows initialization of const and reference members, which cannot be assigned values in the constructor body.
3. Performance: May result in better performance, especially for complex objects or non-trivial data types.

Syntax:

```
class MyClass {
private:
    int x;
    int y;
public:
    // Constructor with initializer list
    MyClass(int val1, int val2) : x(val1), y(val2) {
```

```
        // Constructor body
    }
};
```

Example:

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

class Point {
private:
    int x;
    int y;

public:
    // Parameterized Constructor with initializer list
    Point(int xCoord, int yCoord) : x(xCoord), y(yCoord) {
        // Constructor body
    }

    void display() {
        cout << "x = " << x << ", y = " << y << endl;
    }
};

int main() {
    Point p1(5, 10); // Constructor with initializer list
    p1.display();    // Output: x = 5, y = 10
    return 0;
}
```

In the above example, the Point class has a parameterized constructor that initializes its data members x and y using an initializer list. This ensures that x and y are initialized with specific values immediately upon object creation.

Section 2 : Operator Overloading and Type Conversion

2.1 Operator Overloading

Operator overloading is a powerful feature in C++ that allows us to define the behavior of operators such as +, -, *, /, ==, !=, etc., for objects of user-defined classes. It provides a way to to extend the functionality of built-in operators to work with objects of our own classes.

Syntax:

```
return_type operator op (parameters) {
    // Operator implementation
}
```

Where op is the operator being overloaded and parameters represent the operands of the operator.

Conditions where operator overloading is necessary:

1. **When working with user-defined types:** Operator overloading allows user-defined types, such as classes or structures, to behave like built-in types. This can make the code more readable and maintainable by providing a natural syntax for operations involving those types.
2. **When defining mathematical operations for custom types:** For example, when working with complex numbers, matrices, or vectors, overloading operators like +, -, *, etc., can make the code more expressive and concise.
3. **When implementing custom data structures:** Operator overloading can be useful when implementing custom data structures like arrays, lists, or trees. For example, overloading the subscript operator [] can provide array-like access to elements of a custom container class.
4. **When working with streams and IO operations:** Overloading stream insertion (<<) and stream extraction (>>) operators can enable custom formatting and parsing of data types when performing input/output operations.

Rules for Overloading Operators:

1. Only Existing Operators Can Be Overloaded:
    - You cannot create new operators; you can only overload the existing ones.
2. Preserve the Operator's Arity:
    - The number of operands an operator works with cannot be changed. For instance, a binary operator (e.g., +) must remain binary.
3. Operator Overloading Cannot Change Precedence or Associativity:
    - The precedence and associativity of operators remain the same regardless of overloading.
4. Overloaded Operators Must Have at Least One User-Defined Type as Operand:
    - At least one operand must be a user-defined type (class or struct).
5. Certain Operators Cannot Be Overloaded:
    - There are a few operators that cannot be overloaded, including:
        - : : (scope resolution)
        - . (member access)
        - -> (pointer-to-member)
        - ? : (ternary conditional)

Overloading Various Operators:

Operators can be overloaded to perform custom operations depending on the context and the type of objects involved. Some common Unary or Binary operators that are frequently overloaded include:

- Arithmetic Operators: `+`, `-`, `*`, `/`, `%`
- Relational Operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical Operators: `&&`, `||`, `!`
- Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`

- Increment and Decrement Operators: ++, --
- Subscript Operator: [ ]
- Function Call Operator: ( )
- Stream Operators: >>, << (for input/output)

Note: Following operators can't be overloaded :

- Conditional operator: ? : (ternary operator)
- Member access operator: . (dot)
- Pointer to member access operator: ->* (arrow)
- Scope resolution operator: : : (double colon)
- Sizeof operator: sizeof

Operators that Cannot Be Overloaded Using Friend Function and Member Function

1. Using Friend Function:
    - = (assignment operator)
    - [ ] (subscript operator)
    - ( ) (function call operator)
    - -> (member access operator)
2. Using Member Function:
    - All operators can be overloaded using member functions, except for those that cannot be overloaded at all (i.e., : : , . , ->* , ? :).

Example of Overloading a Binary Operator:

Here's how you can overload the binary + operator using both member function and friend function:

Using Member Function:

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

class Complex {
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Overloading the + operator using a member function
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

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

```
};

int main() {
    Complex c1(3.0, 4.0), c2(1.0, 2.0);
    Complex c3 = c1 + c2; // Uses the overloaded + operator
    c3.display(); // Output: 4.0 + 6.0i
    return 0;
}
```

Using Friend Function:

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

class Complex {
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Friend function to overload the + operator
    friend Complex operator+(const Complex& c1, const Complex& c2);

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

// Definition of the friend function
Complex operator+(const Complex& c1, const Complex& c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main() {
    Complex c1(3.0, 4.0), c2(1.0, 2.0);
    Complex c3 = c1 + c2; // Uses the overloaded + operator
    c3.display(); // Output: 4.0 + 6.0i
    return 0;
}
```

2.2 Type Conversion

Type conversion in C++ involves converting values from one data type to another. This can be particularly useful when working with user-defined types (classes) and built-in types. Here, we'll explore different types of conversions involving class types.

A. Basic Type to Class Type Conversion

Basic types, such as integers or floating-point numbers, can be converted to class types using constructors that accept parameters of the corresponding basic types. This allows for initialization of class objects with values from basic types.

Example: Conversion from int to String Class

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

class String {
private:
    string value;

public:
    String(int num) {
        value = to_string(num);
    }

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

int main() {
    int number = 42;
    String str = number; // Conversion from int to String

    str.display();
    return 0;
}
```

B. Class Type to Basic Type Conversion

Class types can be converted to basic types by overloading conversion operators or by providing explicit conversion functions. This allows for extracting relevant information or converting class objects into values of basic types.

Example: Conversion from String Class to int

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

class String {
private:
    string value;
```

```
public:
    String(string val) : value(val) {}

    // Conversion operator for int
    operator int() const {
        return stoi(value); // Convert string to int
    }
};

int main() {
    String str("42");
    int number = str; // Conversion from String to int

    cout << "Integer value: " << number << endl;
    return 0;
}
```

C. Class Type to Another Class Type Conversion

Class types can be converted to other class types through constructors or conversion operators defined in the respective classes. This allows for creating objects of one class type from objects of another class type.

Example: Conversion from Feet Class to Meter Class

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

class Meter {
private:
    double value;

public:
    Meter(double val) : value(val) {}

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

class Feet {
private:
    double value;

public:
    Feet(double val) : value(val) {}
```

```
    // Conversion operator for Meter
    operator Meter() const {
        return Meter(value * 0.3048); // Convert feet to meter
    }
};

int main() {
    Feet feet(10);
    Meter meter = feet; // Conversion from Feet to Meter

    meter.display();
    return 0;
}
```

Section 3 : Virtual Functions & Polymorphism

3.1 Concept of Binding

Binding in C++ refers to the association between function calls and function definitions. It determines which function definition gets executed when a function is called. Binding can be of two types:

1. Static (or Early) Binding
2. Dynamic (or Late) Binding

A. Early Binding (Static or Compile-time Binding)

Early binding, also known as static binding or compile-time binding, occurs when the function call is resolved at compile time. In early binding, the compiler determines which function implementation to call based on the static type of the object or pointer. Static binding is used for normal function calls, function overloading and operator overloading.

Example:

```
include <iostream>
using namespace std;

class Base {
public:
    void display() {
        cout << "Display method of Base class" << endl;
    }
};

class Derived : public Base {
public:
```

```
    void display() {
        cout << "Display method of Derived class" << endl;
    }
};

int main() {
    Derived d;
    Base* ptr = &d; // Pointer of Base class pointing to Derived object

    ptr->display(); // Early binding, calls Base::display() at compile time
    return 0;
}
```

B. Late Binding (Dynamic or Runtime Binding)

Late binding, also known as dynamic binding or runtime binding, occurs when the function call is resolved at runtime. In late binding, the actual function implementation to be called is determined based on the dynamic type of the object pointed to.
Dynamic binding is achieved through the use of virtual functions or method overriding and is used in scenarios involving polymorphism.

Example:

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

class Base {
public:
    virtual void display() {
        cout << "Display method of Base class" << endl;
    }
};

class Derived : public Base {
public:
    void display() {
        cout << "Display method of Derived class" << endl;
    }
};

int main() {
    Derived d;
    Base* ptr = &d; // Pointer of Base class pointing to Derived object

    ptr->display(); // Late binding, calls Derived::display() at runtime
    return 0;
}
```

When to Use Dynamic Binding

Dynamic binding is particularly useful in situations where you need polymorphic behavior. This allows you to write more generic and flexible code. For instance, if you have a base class Shape and derived classes Circle, Square, etc., you can use dynamic binding to call the appropriate draw() function for each shape at runtime.

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

class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
    virtual ~Shape() {}
};

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

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

void displayShape(Shape* shape) {
    shape->draw();
}

int main() {
    Circle circle;
    Square square;

    displayShape(&circle);  // Drawing Circle
    displayShape(&square);  // Drawing Square

    return 0;
}
```

Here, the draw() function call is dynamically bound to the correct derived class implementation at runtime, demonstrating polymorphic behavior.

3.2 Virtual Functions

In C++, a virtual function is a member function of a class that is declared with the virtual keyword. Virtual functions enable polymorphic behavior, allowing derived classes to provide their own implementation of the function by method overriding.

Declaration and Syntax:

- The virtual keyword is used to declare a function as virtual in the base class.

```
class Base {
public:
    virtual void myVirtualFunction() {
        // Base class implementation
    }
};
```

- In the derived class, the override keyword is used to explicitly indicate that the function is overriding a virtual function from the base class.

```
class Derived : public Base {
public:
    void myVirtualFunction() override {
        // Derived class implementation
    }
};
```

Purpose and Benefits:

1. Polymorphism: Virtual functions enable polymorphism, allowing different classes to provide different implementations of the same function.
2. Dynamic Binding: Virtual functions are resolved at runtime based on the actual type of the object, enabling dynamic method dispatch.
3. Base Class Pointers: Virtual functions are often used with base class pointers to achieve runtime polymorphism.

Example:

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

// Base class
class Animal {
public:
    // Virtual function
    virtual void makeSound() {
        cout << "Animal makes a sound" << endl;
    }
};

// Derived class
class Dog : public Animal {
```

```
public:
    // Override virtual function
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};

// Derived class
class Cat : public Animal {
public:
    // Override virtual function
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog(); // Creating a Dog object
    Animal* animal2 = new Cat(); // Creating a Cat object

    // Calling virtual function on Dog object
    animal1->makeSound(); // Output: Dog barks

    // Calling virtual function on Cat object
    animal2->makeSound(); // Output: Cat meows

    return 0;
}
```

In this example:

- We have a base class Animal with a virtual function makeSound().
- There are two derived classes Dog and Cat, each overriding the makeSound() function with their specific implementation.
- In the main() function, we create objects of type Dog and Cat using base class pointers.
- When we call the makeSound() function on these objects, the appropriate version of the function is invoked based on the actual type of the object, demonstrating polymorphic behavior through dynamic dispatch.

3.3 Pure Virtual Functions

In C++, a pure virtual function (or abstract function) is a virtual function declared in a base class that has no implementation. It is declared by assigning 0 in the base class, must be overridden in derived classes. It serves as a placeholder for derived classes to override and provide their own implementation. A class containing at least one pure virtual function is known as abstract class and cannot be instantiated directly.

Declaration and Syntax:

- Pure virtual functions are declared with = 0 at the end of their declaration.

```
class AbstractBase {
public:
    virtual void pureVirtualFunction() = 0; // Pure virtual function
};
```

Example:

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

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

// Derived class
class Dog : public Animal {
public:
    // Override pure virtual function
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};

// Derived class
class Cat : public Animal {
public:
    // Override pure virtual function
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog(); // Creating a Dog object
    Animal* animal2 = new Cat(); // Creating a Cat object

    // Calling pure virtual function on Dog object
    animal1->makeSound(); // Output: Dog barks

    // Calling pure virtual function on Cat object
    animal2->makeSound(); // Output: Cat meows
```

```
    return 0;
}
```

In this example:

- We have an abstract base class Animal with a pure virtual function makeSound().
- There are two derived classes Dog and Cat, each overriding the makeSound() function with their specific implementation.
- In the main() function, we create objects of type Dog and Cat using base class pointers.
- When we call the makeSound() function on these objects, the appropriate version of the function is invoked based on the actual type of the object, demonstrating polymorphic behavior through dynamic dispatch.

Abstract Classes and Interfaces:

- **Abstract Class:** A class that contains at least one pure virtual function. It cannot be instantiated and serves as a blueprint for derived classes to implement common behavior while allowing specific implementations for their own unique features.
- **Interface:** C++ does not have a built-in concept of interfaces like Java or C#. However, interfaces can be simulated using abstract classes. An interface in C++ is an abstract class that has only pure virtual functions and no data members or non-virtual member functions. This ensures that the derived classes implement the specific methods defined by the interface.

Differences between Abstract Classes and Interfaces:

Feature	Abstract Classes	Interfaces
Definition	A class containing at least one pure virtual function	An abstract class containing only pure virtual functions
Purpose	To provide a common base class with some implementation and some methods to be overridden by derived classes	To define a contract that derived classes must follow
Implementation	Can contain some implementation (non-pure virtual functions) and member variables	Contains only pure virtual functions and no member variables
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Inheritance	Derived classes can inherit only one abstract class (single inheritance)	A class can implement multiple interfaces (multiple inheritance)
Use Case	Use when you need a base class with some common behavior	Use when you need to define a clear contract for behavior without any implementation details

3.4 Virtual Destructors

In C++, when dealing with polymorphism and inheritance, it is often necessary to use virtual destructors to ensure that the proper destructors are called for objects of derived classes. A virtual destructor is a destructor declared in a base class using the virtual keyword, and it ensures that the destructors of both the base and derived classes are called in the correct order when deleting an object through a pointer to the base class. Virtual destructors become essential to ensure proper cleanup of resources allocated by derived classes.

Declaration and Syntax:

- The virtual keyword is used in the base class destructor to make it virtual.

```
class Base {
public:
    virtual ~Base() {
        // Virtual destructor
    }
};
```

- The derived class destructor overrides the base class destructor and provides its own implementation.

```
class Derived : public Base {
public:
    ~Derived() override {
        // Derived class destructor
    }
};
```

Purpose and Benefits:

1. Proper Destruction Order: Virtual destructors ensure that the destructors are called in the correct order when deleting objects through base class pointers, preventing memory leaks and undefined behavior.
2. Polymorphic Deletion: Virtual destructors enable the polymorphic deletion of objects, allowing for the correct cleanup of resources allocated by derived classes.
3. Prevents Memory Leaks: Helps avoid memory leaks by ensuring derived class destructors are called.

Example:

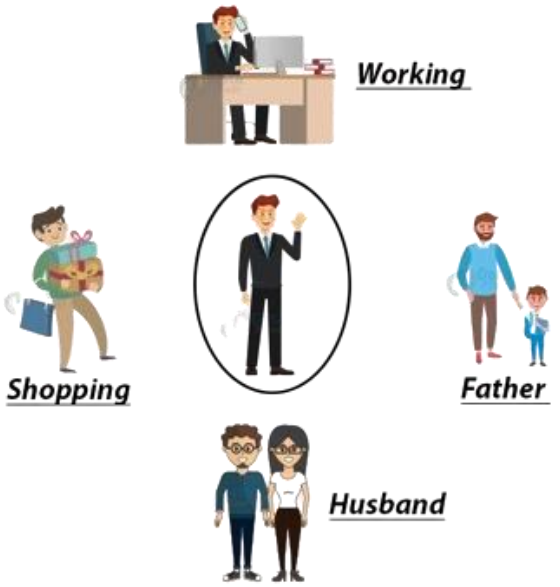
```
#include <iostream>
using namespace std;

// Base class with a virtual destructor
class Base {
public:
    virtual ~Base() {
        cout << "Base class destructor" << endl;
    }
```

```
};

// Derived class with its own destructor
class Derived : public Base {
public:
    ~Derived() override {
        cout << "Derived class destructor" << endl;
    }
};

int main() {
    Base* basePtr = new Derived(); // Creating a Derived object through a
Base pointer
    delete basePtr; // Deleting through a base class pointer

    return 0;
}
```

3.5 Polymorphism

Definition

- Polymorphism is derived from the Greek words "poly" (many) and "morphos" (forms).
- Polymorphism refers to the ability of objects to take on different forms or behaviors based on their context.
- In OOP, polymorphism refers to the ability of objects of different classes to be treated as objects of a common superclass.
- It allows a single interface (method or function) to represent multiple implementations.
- It allows a single function or operator to exhibit different behaviors based on the context in which it is called.
- Example: A man acts a father, husband, son, employee and many more.



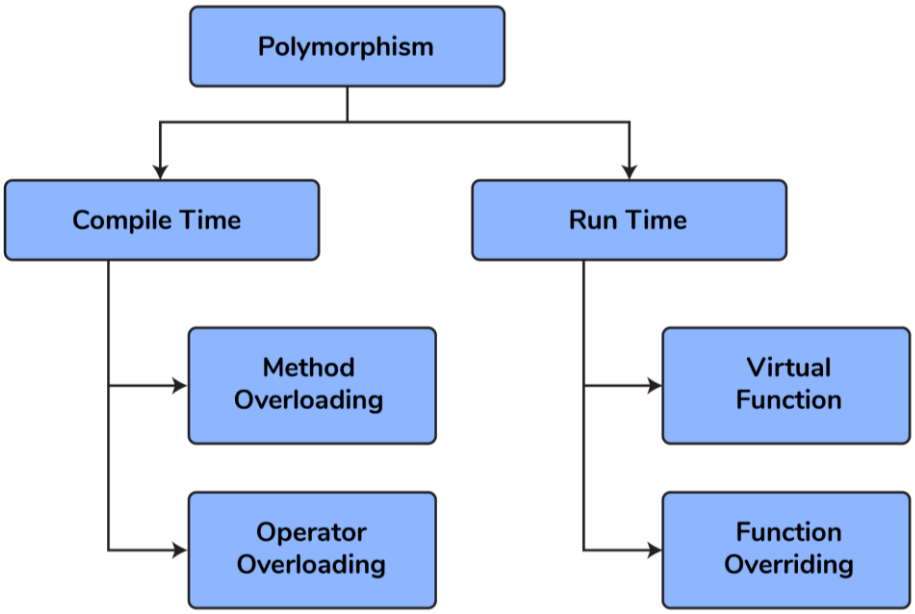
Benefits

1. Code Reusability: Polymorphism allows the same code to be reused with different objects, reducing duplication and improving maintainability.
2. Simplification: Polymorphism simplifies code maintenance and enhances readability by promoting a more modular and organized code structure.
3. Flexibility and Extensibility: It provides a flexible way to add new functionality to existing code by extending existing classes.
4. Encapsulation: Polymorphism promotes encapsulation by abstracting away the implementation details of objects and focusing on their behavior through a common interface.

Types of Polymorphism

Polymorphism can be of two types:

1. **Compile Time Polymorphism (Early / Static Binding)**
    - Achieved through method overloading and operator overloading.
    - Decisions about method calls are made at compile time.
    - Determined by the number and types of arguments and return type.
2. **Run Time Polymorphism (Late / Dynamic Binding)**
    - Achieved through virtual functions or method overriding.
    - Decisions about method calls are made at runtime.
    - Facilitated by pointers or references to base class objects.



Compile-time and Run-time

- **Compile Time (Early/Static):**
  - This is the phase when the **source code** written in programming language is being **converted into executable code** by a compiler.
  - During compile time, the compiler checks for **syntax errors** (like missing semicolons or mismatched brackets) and **semantic errors** (such as using a variable that hasn't been declared).
  - The compiler will not create an executable file until all such errors are resolved.
- **Run Time (Late/Dynamic):**
  - Run time refers to the period when the **executable code is actually running** on computer.
  - It's the phase where the program interacts with inputs, performs calculations, and may encounter **runtime errors**. These errors occur during execution and can include issues like division by zero or accessing an array out of bound.

A. Compile-time Polymorphism (Static Polymorphism):

- Polymorphism that is resolved at compile time when the **source code** is being **converted into executable code** by compiler.
- It is achieved through method overloading and operator overloading, where the compiler selects the appropriate function or operator based on the arguments and context at compile time.
- Also known as **static or early binding**, this type of polymorphism is achieved through method overloading and operator overloading.

Method Overloading:

- Definition: Method overloading is a form of compile-time polymorphism where multiple methods in the same class have the same name but differ in the number or type of their parameters.

- Methods can be overloaded by changing the number or type of arguments.
- It provides flexibility and clarity in code by allowing multiple functions with similar functionality to be grouped under the same name.
- Example:

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

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

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

    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(5, 7) << endl;         // Output: 12
    cout << calc.add(5, 7, 3) << endl;      // Output: 15
    cout << calc.add(3.5, 2.5) << endl;     // Output: 6
    return 0;
}
```

Operator Overloading:

- Definition: Operator overloading is a form of compile-time polymorphism where operators are overloaded to work with user-defined data types.
- It allows defining custom behavior for operators based on the data types involved.
- Example:

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

class Complex {
private:
    int real, imag;
public:
    // Constructor to initialize real and imaginary parts, default
values are 0
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}
```

```
    // Overloading the + operator to add two complex numbers
    Complex operator+(Complex const& obj) {
        // Adding real parts and imaginary parts separately
        return Complex(real + obj.real, imag + obj.imag);
    }

    // Function to print the complex number in the format "real + imagi"
    void print() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    // Creating two complex numbers
    Complex c1(10, 5), c2(2, 4);

    // Adding two complex numbers using overloaded + operator
    Complex c3 = c1 + c2;

    // Printing the result
    cout << "Result of addition: ";
    c3.print();

    return 0;
}
```

Output:

```
Result of addition: 12 + 9i
```

B. Run-time Polymorphism (Dynamic Polymorphism):

- Polymorphism that is resolved at runtime when the **executable code is actually running** on computer.
- It is achieved through method overriding or virtual functions, where the appropriate function to call is determined dynamically based on the actual object type at runtime.
- Also known as **dynamic or late binding**, this occurs during program execution.
- Achieved through virtual functions (using the virtual keyword).
- Allows a base class pointer to invoke derived class methods.

Virtual Functions:

- Definition: Virtual functions are used in run-time polymorphism to enable dynamic method binding. They are member functions which are declared in the base class with the virtual keyword and can be overridden in derived classes.
- They enable dynamic binding of function calls, allowing the correct function to be called at runtime based on the type of object.

- Example:

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

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

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

int main() {
    Shape* shape = new Circle();
    shape->draw();  // Output: Drawing Circle

    return 0;
}
```

Method Overriding:

- Definition: Method overriding is a form of run-time polymorphism where a method in a base class is redefined in a derived class. The method in the derived class must have the same signature (name and parameters) as the one in the base class.
- It allows derived classes to provide a specific implementation of a method defined in the base class, promoting flexibility and extensibility.
- Example:

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

class Animal {
public:
    virtual void sound() {
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
```

```
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* animal = new Dog();
    animal->sound();  // Output: Dog barks

    return 0;
}
```

Unit 4: Exception Handling and Templates Programming

Syllabus :
Exception Handling: Review of traditional error handling, basics of exception handling, exception handling mechanism, throwing mechanism, catching mechanism, rethrowing an exception, specifying exceptions.
Templates and Generic Programming: Template concepts, Function templates, class templates, illustrative examples.

Section 1 : Exception Handling in C++

Exception handling in C++ is a structured way to handle errors and exceptional conditions that occur during program execution. It provides a way to separate error-handling code from regular code, making programs easier to read and maintain.

1.1 Review of Traditional Error Handling

Traditional error handling refers to the conventional methods used in programming languages to deal with errors or exceptional situations that may occur during program execution.

A. Methods:

1. Error Codes: Traditional error handling often involves the use of error codes or special return values to indicate the occurrence of an error. Functions or methods may return specific values (e.g., -1) to signal an error condition.
2. Global Variables: Some programming languages utilize global variables to store error information. These variables are checked after each operation to determine if an error occurred.

3. Conditional Statements: Developers typically use conditional statements such as if-else or switch-case to handle errors. These statements check for error conditions and execute appropriate error-handling code.

B. Example:

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

int divide(int a, int b) {
    if (b == 0) {
        return -1; // Indicate error
    }
    return a / b;
}

int main() {
    int a = 10, b = 0;
    int result = divide(a, b);
    if (result == -1) {
        cout << "Error: Division by zero" << endl;
    } else {
        cout << "Result: " << result << endl;
    }
    return 0;
}
```

C. Limitations:

1. Error-Prone: Requires manual error checking, which can be easily overlooked.
2. Cluttered Code: Can lead to scattered error handling logic, making the code harder to read and maintain.
3. Limited Information: Return values and error codes provide limited information about the error context.
4. Control Flow Issues: Can lead to convoluted control flow, especially in deeply nested functions or complex logic.

1.2 Basics of Exception Handling

Exception

- An exception in programming refers to an abnormal condition or unexpected event that occurs during the execution of a program, disrupting the normal flow of control. Exceptions are typically caused by errors or exceptional conditions that arise at runtime and may prevent the program from continuing its execution as expected.
- Common reasons that cause exceptions include division by zero (run-time error), invalid input, null pointer dereference, out-of-bounds access, memory allocation

failure, file not found, resource exhaustion, concurrency issues, hardware failures, and errors from system calls or library functions.

Exception Handling

- Exception handling is a mechanism in programming that deals with runtime errors or exceptional situations that may occur during the execution of a program. These exceptional situations could include division by zero, file not found, out-of-memory errors, and so on.
- Exception handling allows a program to respond to such situations in a controlled and graceful manner rather than abruptly terminating or producing undefined behavior.
- Exception handling in C++ involves the use of **try, catch, and throw** keywords to manage runtime errors and handle exceptional conditions gracefully.
    - **try Block:** The code that might generate an exception is placed inside a try block.
    - **throw Statement:** When an error occurs, an exception is thrown using the throw statement.
    - **catch Block:** The exception is caught by a catch block that handles the exception.

Example of Exception Handling

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

int divide(int num, int den) {
    if (den == 0) {
        throw runtime_error("Division by zero"); // Throwing an exception
    }
    return num / den;
}

int main() {
    int num = 10, den = 0;

    try {
        int result = divide(num, den); // Code that may throw an exception
        cout << "Result: " << result << endl;
    } catch (const runtime_error& e) { // Catching the exception
        cout << "Error: " << e.what() << endl;
    }

    return 0;
}
```

Benefits of Exception Handling

1. Separating Error-Handling Code from Regular Code:
    - Exceptions allow you to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
    - In traditional error management, error detection, reporting, and handling often lead to confusing spaghetti code. By using exceptions, you can keep the main flow of your code clean and deal with exceptional cases elsewhere.
2. Enhancing Robustness:
    - Exception handling ensures the continuity of your program even when unexpected errors occur.
    - Instead of crashing, your program gracefully handles exceptions, making it more robust and reliable.
3. Improving Readability and Maintainability:
    - Separating error-handling code from regular code improves the readability of your program.
    - Developers can focus on the main logic without getting distracted by error-related clutter.
4. Accurate Error Reporting:
    - Exceptions provide meaningful error messages, making it easier to identify the cause of failures.
    - When an exception occurs, you can include relevant information (such as stack traces) to pinpoint the issue.
5. Facilitating Debugging and Troubleshooting:
    - Exception stack traces help you trace the sequence of method calls that led to the error.
    - Debugging becomes more efficient because you can quickly locate the problematic code.
6. Improved Security:
    - Proper exception handling prevents security vulnerabilities caused by unexpected behavior.
    - By handling exceptions gracefully, you reduce the risk of exposing sensitive information or allowing unauthorized access.
7. Better User Experience:
    - When exceptions are handled well, users experience fewer crashes or abrupt program terminations.
    - A smooth user experience contributes to overall satisfaction with your software.
8. Enabling Error Recovery Mechanisms:
    - Exceptions allow you to recover from errors gracefully.
    - You can catch exceptions, log relevant information, and take corrective actions without disrupting the program flow.

1.3 Exception Handling Mechanism

Exception handling in C++ provides a structured way to handle runtime errors or exceptional conditions that may occur during program execution, allowing the program to recover without crashing.
It involves three key components: try, throw, and catch.

- **try Block:** The try block encloses the code that might throw an exception. If an exception occurs within the try block, it is transferred to the appropriate catch block.
- **throw Statement:** The throw statement is used to explicitly throw an exception. It can be followed by an expression, which is typically an object of an exception class, to provide information about the error.
- **catch Block:** The catch block catches and handles exceptions thrown by the associated try block. It specifies the type of exception it can handle and provides code to deal with the exception.

A. Throwing Mechanism:

- The throwing mechanism is used to signal that an exceptional condition has occurred during program execution. It involves the throw statement.
- **throw Statement:** The throw statement is followed by an expression, typically an object of an exception class. This expression is the exception that is being thrown.
- Throwing Syntax:

```
throw ExceptionType(arguments);
```

- Example:

```
throw runtime_error("Division by zero");
```

B. Catching Mechanism:

- The catching mechanism is used to handle exceptions thrown by the try block. It involves catch blocks.
- **catch Block:** A catch block is associated with a try block and specifies the type of exception it can handle. When an exception of that type is thrown, the corresponding catch block is executed.
- Example:

```
catch (const runtime_error& e) {
    cout << "Error: " << e.what() << endl;
}
```

- try-catch Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
} catch (...) {
    // Catch-all block to handle any other exceptions
}
```

C. Rethrowing an Exception:

- Rethrowing an exception allows an exception caught in one catch block to be thrown again to be handled by another catch block.
- **throw Statement (inside catch block):** In a catch block, we can use the throw statement without any argument to rethrow the caught exception.
- Rethrow Syntax:

```
catch (ExceptionType e) {
    // Code to handle the exception
    throw; // Rethrow the exception
}
```

- Example:

```
catch (const runtime_error& e) {
    cout << "Caught error: " << e.what() << endl;
    throw; // Rethrow the exception
}
```

D. Specifying Exception :

- In older versions of C++, it was possible to specify the types of exceptions that a function could throw. This feature, known as exception specifications, is deprecated in C++11 and removed in C++17.
- Specifying exceptions allows to document the types of exceptions that a function may throw during its execution.
- Syntax:

```
returnType functionName(parameters) throw(ExceptionType1, ExceptionType2, ...)
{
    // Function body
}
```

- Instead of specifying exceptions, it's better to document the exceptions a function might throw in comments and handle them accordingly.

E. Standard Exception Classes:

In C++, the Standard Template Library (STL) provides a set of standard exception classes that can be used for common error scenarios. These classes are defined in the <stdexcept> header and serve as a hierarchy of exception types that cover a range of standard error conditions.

Common Standard Exception Classes:

1. std::exception:

- The base class for all standard C++ exceptions.
- Provides a virtual function what() that returns a descriptive string representing the exception.
- Developers can define their own custom exception types by deriving from std::exception or its subclasses.

2. std::runtime_error:

- Represents errors that occur during runtime and are typically not detectable before the program is executed.
- Derived from std::exception.
- Commonly used to report logical errors or exceptional conditions that arise during program execution.
- Example:

```
try {
    throw std::runtime_error("A runtime error occurred");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

3. std::logic_error:

- Represents errors that occur due to logical errors in the program's design or implementation.
- Derived from std::exception.
- Examples include out-of-range errors, domain errors, and invalid argument errors.
- Example:

```
try {
    throw std::logic_error("A logic error occurred");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

4. std::invalid_argument:

- Indicates that a function has received an invalid argument.
- Derived from std::logic_error.
- Used when a function is called with an argument that is not acceptable or within the expected range.
- Example:

```
try {
    throw std::invalid_argument("Invalid argument provided");
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

5. std::out_of_range:

- Indicates that an index or value is out of the valid range.
- Derived from std::logic_error.
- Commonly used in situations where accessing elements beyond the bounds of a container or array.
- Example:

```
try {
    throw std::out_of_range("Out of range exception");
```

```
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

6. std::bad_alloc:

- Represents errors that occur when memory allocation fails.
- Derived from std::exception.
- Typically thrown by the new operator when it fails to allocate memory.
- Example:

```
try {
    throw std::bad_alloc();
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

7. std::overflow_error:

- Indicates arithmetic overflow errors, where the result of an arithmetic operation exceeds the range of representable values.
- Derived from std::runtime_error.
- Commonly used in situations where arithmetic operations result in overflow.
- Example:

```
try {
    int result = std::numeric_limits<int>::max() + 1; // Overflow
} catch (const std::overflow_error& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

8. std::underflow_error:

- Indicates arithmetic underflow errors, where the result of an arithmetic operation is smaller than the minimum representable value.
- Derived from std::runtime_error.
- Less common than std::overflow_error but used in similar situations where arithmetic underflow occurs.
- Example:

```
try {
    float result = std::numeric_limits<float>::min() / 2; // Underflow
} catch (const std::underflow_error& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

Section 2 : Templates and Generic Programming

Templates and generic programming are powerful features of C++ that allow developers to write code that works with any data type. Templates provide a mechanism for creating generic classes and functions, allowing them to operate on multiple data types without the need for code duplication.

2.1 Template Concepts

- Definition: Templates are a feature of C++ that allows functions and classes to operate with generic types. They enable the creation of generic code that works with any data type.
- Benefits:
    - Code Reusability: Templates allow you to write code once and use it with different data types, promoting reuse.
    - Flexibility: Templates provide flexibility by allowing algorithms to work with various data types without sacrificing performance or type safety.
- Syntax: Template definitions begin with the template keyword followed by a list of template parameters enclosed in angle brackets < >.

2.2 Function Templates

- Definition: Function templates allow you to create a single function that can operate with different data types. They are instantiated to create specific functions for each data type when called.
- They are defined using the template keyword followed by template parameters enclosed in angle brackets (<>).
- Template parameters can be type parameters or non-type parameters.
- Syntax:

```
template <typename T>
T functionName(T parameter) {
    // Function body
}
```

- Example:

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

// Function template for adding two values of the same type
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Instantiation for int
    int result1 = add(5, 3);
    cout << "Result 1: " << result1 << endl;

    // Instantiation for double
    double result2 = add(3.5, 2.7);
    cout << "Result 2: " << result2 << endl;

    return 0;
```

```
}
```

In this example, the function template add can add two values of any data type (int, double, etc.).

2.3 Class Templates

- Definition: Class templates allow the creation of generic classes that can work with any data type. They are instantiated to create specific classes for each data type when used.
- They are defined similar to function templates, but instead of functions, entire classes are templated.
- Syntax:

```
template <typename T>
class ClassName {
    // Class members and methods
};
```

- Example:

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

// Class template for a generic Pair
template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 f, T2 s) : first(f), second(s) {}
    T1 getFirst() const { return first; }
    T2 getSecond() const { return second; }
};

int main() {
    // Usage of Pair class template with different types
    Pair<int, double> pair1(5, 3.14);
    cout << "First: " << pair1.getFirst() << ", Second: " <<
pair1.getSecond() << endl;

    Pair<char, string> pair2('A', "Hello");
    cout << "First: " << pair2.getFirst() << ", Second: " <<
pair2.getSecond() << endl;

    return 0;
}
```

In this example, the class template Pair represents a generic pair of two values of potentially different types (int, double, char, string, etc.).

Class Templates vs. Function Templates

Aspect	Class Template	Function Template
Purpose	Class templates are used to create generic classes	Function templates are used to create generic functions
Syntax	template <typename T> class ClassName	template <typename T> returnType functionName
Usage	Class templates are instantiated with specific data types when creating objects	Function templates are instantiated with specific data types when calling the function
Example	Class templates define structure and behavior of generic classes	Function templates define logic of generic functions