```
    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.

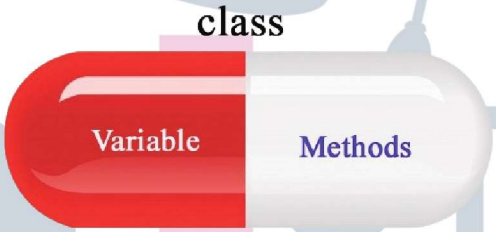
4. Polymorphism

- 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

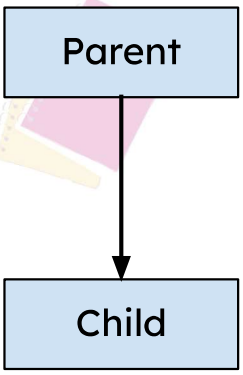
1. Code Reusability: Inheritance allows classes to inherit attributes and methods from other classes, reducing redundancy and promoting code reuse.
2. Extensibility: Inheritance enables the addition of new features to a class without modifying its existing structure, thus facilitating the extension of 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:

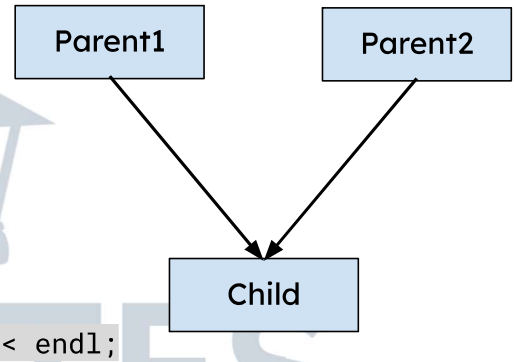
```
#include <iostream>
using namespace std;

// 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:

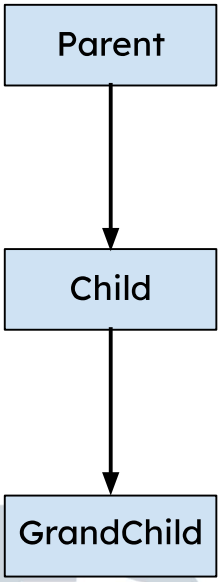



```
#include <iostream>
using namespace std;

// 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:

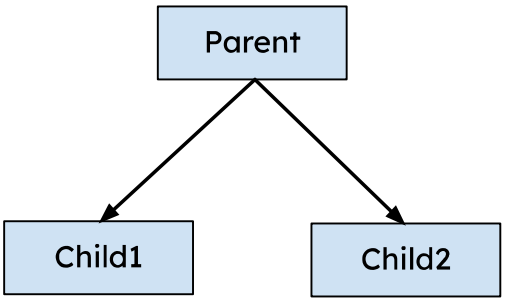



```
#include <iostream>
using namespace std;

// 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:

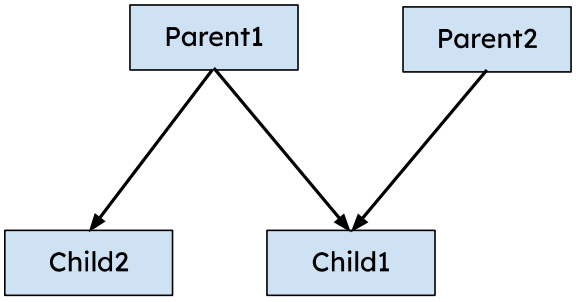
- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- It allows for creating a hierarchy of classes with a common base class.
- It is like Father -> Son & Daughter relationship.
- Example:




```
#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:



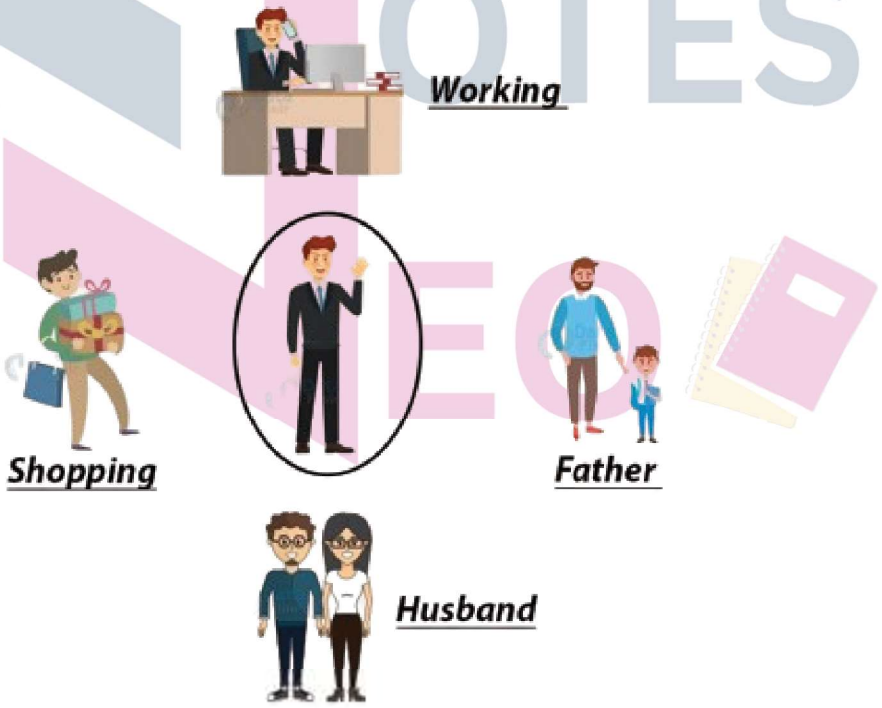

```
#include <iostream>
using namespace std;

// 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

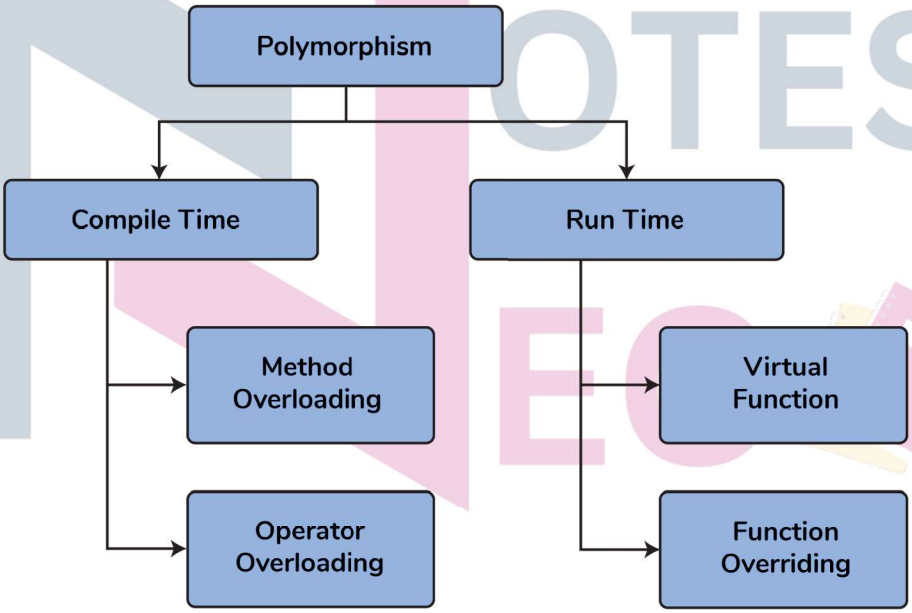
1. Code Reusability: Polymorphism allows the same code to be reused with different objects, reducing duplication and improving maintainability.

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:

```
#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:

```cpp
#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:
```