ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

CLASSES & OBJECTS: CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS

Constructors are special member functions in a class that are automatically invoked when an object of that class is created. The primary role of a constructor is to initialize the object's data members and allocate resources when necessary. Constructors have the same name as the class and do not return any value. They play a vital role in ensuring that objects are initialized correctly when they are instantiated.

Constructors can be classified into two main types:

1. **Default Constructor**: A constructor that takes no arguments and initializes an object with default values.
2. **Parameterized Constructor**: A constructor that takes arguments and initializes an object with the provided values.

Example: Default Constructor

```
#include <iostream>

using namespace std;
```

```
class Car {
public:
   string brand;
   string model;
   int year;

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

   void display() {
      cout << "Car Brand: " << brand << ", Model: " << model << ", Year:
" << year << endl;
   }
};

int main() {
   Car car1;  // Default constructor is called
```

```
    car1.display();

    return 0;

}
```

In this example, the Car class has a default constructor that initializes the data members to default values when an object of the class is created. The car1 object is instantiated, and the constructor is called automatically to initialize the object's attributes.

PARAMETERIZED CONSTRUCTOR EXAMPLE

```
#include <iostream>

using namespace std;


class Car {

public:

    string brand;

    string model;

    int year;


    // Parameterized constructor

    Car(string b, string m, int y) {

        brand = b;

        model = m;

        year = y;
```

```
    }


    void display() {
        cout << "Car Brand: " << brand << ", Model: " << model << ", Year: " << year << endl;
    }
};


int main() {
    Car car1("Toyota", "Corolla", 2021);  // Parameterized constructor is called
    car1.display();
    return 0;
}
```

In this example, the Car class has a parameterized constructor that allows for custom initialization of the object. The constructor accepts values for brand, model, and year, which are passed when creating the object car1.

DESTRUCTORS

Destructors are special member functions in a class that are invoked when an object goes out of scope or is explicitly deleted. They are used to release resources or perform cleanup operations, such as closing files or freeing dynamically allocated memory, that were acquired during the lifetime of an object. A destructor has the same

name as the class but is prefixed with a tilde ~ and does not take any arguments nor return any value.

Unlike constructors, destructors cannot be overloaded, and there is only one destructor allowed per class. The destructor is called automatically when an object is destroyed, and it is a key feature in resource management, particularly in cases involving dynamic memory allocation or file handling.

Example: Destructor

```
#include <iostream>

using namespace std;


class Car {

public:

  string brand;

  string model;

  int year;


  // Parameterized constructor

  Car(string b, string m, int y) {

    brand = b;

    model = m;

    year = y;

  }
```

```
    // Destructor
    ~Car() {
        cout << "Destructor called for Car: " << brand << " " << model << endl;
    }

    void display() {
        cout << "Car Brand: " << brand << ", Model: " << model << ", Year: " << year << endl;
    }
};

int main() {
    Car car1("Honda", "Civic", 2020);
    car1.display();  // Destructor will be called automatically when the object goes out of scope
    return 0;
}
```

In this example, the destructor of the Car class is called automatically when the object car1 goes out of scope. The destructor displays a message indicating that it was invoked.

DESTRUCTOR AND DYNAMIC MEMORY MANAGEMENT

When objects use dynamic memory allocation (e.g., with new), destructors are crucial for freeing that memory to avoid memory leaks. In the following example, we allocate memory for an array dynamically in the constructor and release it in the destructor.

```
#include <iostream>

using namespace std;


class Array {

public:

  int *arr;

  int size;


  // Constructor that dynamically allocates memory

  Array(int s) {

    size = s;

    arr = new int[size];

    cout << "Memory allocated for array of size " << size << endl;

  }


  // Destructor that frees the dynamically allocated memory

  ~Array() {

    delete[] arr;
```

```
        cout << "Memory freed for array" << endl;
    }

    void display() {
        for (int i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Array arr(5);  // Constructor is called, memory is allocated
    arr.display();
    return 0;  // Destructor is automatically called, memory is freed
}
```

In this example, the Array class has a constructor that dynamically allocates memory for an array, and the destructor is responsible for freeing the memory. The destructor ensures that memory leaks are avoided by releasing the memory when the object is destroyed.

Case Study: Resource Management Using Constructors and Destructors

Consider a system that involves managing database connections. When a database connection object is created, it might need to establish a connection to the database (which is a resource that needs to be managed). The constructor would be responsible for establishing the connection, and the destructor would ensure that the connection is closed when the object is destroyed.

For instance, in a database management system, an object might be responsible for connecting to a database and performing queries. Using a constructor, the system can ensure the connection is properly opened when the object is created, and using a destructor, the system can ensure that the connection is closed when the object goes out of scope.

```
#include <iostream>

using namespace std;


class DatabaseConnection {

public:

  string connectionString;


  // Constructor to establish the database connection

  DatabaseConnection(string connStr) {

    connectionString = connStr;

    cout << "Database connection established: " << connectionString
<< endl;

  }
```

```
    // Destructor to close the database connection
    ~DatabaseConnection() {
        cout << "Database connection closed: " << connectionString << endl;
    }
};

int main() {
    DatabaseConnection dbConn("Server=localhost; Database=TestDB;");  // Constructor establishes connection
    // Perform database operations...
    return 0;  // Destructor closes connection when the object is destroyed
}
```

In this case study, the constructor establishes a connection to a database, and the destructor ensures that the connection is closed, demonstrating resource management through constructors and destructors.

CONCLUSION

Constructors and destructors are essential features of object-oriented programming in C++. Constructors allow for the proper initialization of objects, ensuring they start with valid states, while

destructors provide a mechanism for releasing resources when objects are no longer needed. Together, they enable efficient resource management and prevent memory leaks, especially in systems involving dynamic memory allocation or external resources like database connections or file handling.

EXERCISES

1. **Exercise on Constructors:**
    - Write a class called Book that has three attributes: title, author, and price. Implement both a default constructor and a parameterized constructor to initialize these attributes. Display the book details using a member function.
2. **Exercise on Destructors:**
    - Modify the Book class to dynamically allocate memory for the title and author. Write a destructor to free the allocated memory when the Book object is destroyed.
3. **Exercise on Dynamic Memory Management:**
    - Create a class that simulates a dynamic array of integers. Implement a constructor to allocate memory for the array and a destructor to free the memory. Implement a member function to display the array's elements.

ENCAPSULATION, INHERITANCE, POLYMORPHISM

ENCAPSULATION

Encapsulation is one of the fundamental principles of object-oriented programming (OOP), referring to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class. This concept helps in controlling access to the internal state of an object, ensuring that an object's attributes are not directly modified. Instead, access to the attributes is provided through well-defined methods, often referred to as getters and setters.

The primary goal of encapsulation is to protect the integrity of the object by restricting access to its internal details and only exposing a controlled interface to the outside world. By hiding the internal implementation and allowing access only through defined interfaces, encapsulation helps in reducing complexity and promoting modularity.

ACCESS MODIFIERS IN ENCAPSULATION

In C++, encapsulation is achieved using access modifiers:

- **Public**: Members that are accessible from outside the class.
- **Private**: Members that are accessible only within the class itself.
- **Protected**: Members that are accessible within the class and its derived classes.

Example: Encapsulation in C++

```
#include <iostream>

using namespace std;


class Person {

private:

   string name; // Private data member

   int age;    // Private data member


public:

   // Setter functions (Mutators)

   void setName(string n) {

      name = n;

   }


   void setAge(int a) {

      if (a > 0) {

         age = a;

      } else {

         cout << "Invalid age!" << endl;

      }

   }
```

```
    // Getter functions (Accessors)
    string getName() {
        return name;
    }

    int getAge() {
        return age;
    }

    void display() {
        cout << "Name: " << getName() << ", Age: " << getAge() << endl;
    }
};

int main() {
    Person person;
    person.setName("Alice");
    person.setAge(25);
    person.display();
```

```
    return 0;
}
```

In this example, the Person class encapsulates the name and age attributes. These attributes are marked as private and cannot be accessed directly. The class provides public setter and getter functions (setName(), setAge(), getName(), and getAge()) to interact with the private attributes, thus ensuring controlled access and modification.

INHERITANCE

Inheritance is another key concept in OOP that allows a class (called a subclass or derived class) to inherit properties and behaviors (attributes and methods) from another class (called a superclass or base class). This allows for code reuse, making it possible to create a new class that builds upon an existing class. Inheritance also enables a hierarchical relationship between classes.

C++ supports single inheritance (where a subclass inherits from one superclass) and multiple inheritance (where a subclass inherits from more than one class).

TYPES OF INHERITANCE IN C++

- **Single Inheritance**: A class inherits from only one base class.
- **Multiple Inheritance**: A class inherits from more than one base class.
- **Multilevel Inheritance**: A class inherits from another class, which is itself derived from another class.

- **Hierarchical Inheritance**: Multiple classes inherit from a single base class.
- **Hybrid Inheritance**: A combination of multiple types of inheritance.

Example: Single Inheritance in C++

```
#include <iostream>

using namespace std;


class Animal {

public:

  void eat() {

    cout << "Eating..." << endl;

  }

};


class Dog : public Animal {

public:

  void bark() {

    cout << "Barking..." << endl;

  }

};
```

```
int main() {
   Dog dog;
   dog.eat();  // Inherited method from Animal class
   dog.bark(); // Method of Dog class

   return 0;
}
```

In this example, the Dog class inherits the eat() method from the Animal class. The Dog class has its own method, bark(), which is not shared by other classes.

Example: Multiple Inheritance in C++

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

class Vehicle {
public:
   void drive() {
      cout << "Driving vehicle..." << endl;
   }
};

class Engine {
```

```
public:
    void start() {
        cout << "Starting engine..." << endl;
    }
};

class Car : public Vehicle, public Engine {
public:
    void honk() {
        cout << "Honking horn..." << endl;
    }
};

int main() {
    Car car;
    car.drive();  // From Vehicle class
    car.start();  // From Engine class
    car.honk();   // From Car class

    return 0;
}
```

Here, the Car class demonstrates multiple inheritance by inheriting from both the Vehicle and Engine classes. It has access to the methods of both parent classes.

POLYMORPHISM

Polymorphism is the ability of a class or function to take on multiple forms. In OOP, polymorphism allows you to define methods that can behave differently based on the object that invokes them. It can be broadly categorized into:

- **Compile-time Polymorphism** (Method Overloading and Operator Overloading)
- **Run-time Polymorphism** (Achieved via function overriding)

COMPILE-TIME POLYMORPHISM

This type of polymorphism is resolved during the compile time, and it typically involves:

- **Method Overloading**: The same method name is used, but with different parameters.
- **Operator Overloading**: Operators are given new behavior based on the operands' types.

Example: Method Overloading

```
#include <iostream>

using namespace std;


class Display {
```

```
public:
    void show(int i) {
        cout << "Integer: " << i << endl;
    }

    void show(double d) {
        cout << "Double: " << d << endl;
    }
};

int main() {
    Display obj;
    obj.show(5);    // Calls show(int)
    obj.show(3.14); // Calls show(double)

    return 0;
}
```

In this example, the show() method is overloaded with different parameter types. Based on the argument passed, the appropriate version of the function is invoked.

RUN-TIME POLYMORPHISM (FUNCTION OVERRIDING)

Run-time polymorphism is achieved using function overriding, where a derived class redefines a function from the base class. The function is resolved during runtime based on the type of the object.

Example: Run-time Polymorphism (Function Overriding)

```
#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;

  }

};


class Cat : public Animal {
```

```
public:
    void sound() override {
        cout << "Cat meows." << endl;
    }
};

int main() {
    Animal* animal;
    Dog dog;
    Cat cat;

    animal = &dog;
    animal->sound();  // Calls Dog's sound()

    animal = &cat;
    animal->sound();  // Calls Cat's sound()

    return 0;
}
```

In this example, the sound() function is overridden in the Dog and Cat classes. Using a base class pointer (Animal*), we can achieve

run-time polymorphism. The actual method that gets called depends on the object that the pointer is pointing to.

Case Study: Polymorphism in Real-World Applications

Consider a simulation program for different types of payment systems in an online store. The store may have several payment methods, such as CreditCard, PayPal, and Cryptocurrency. Each payment method might have a different implementation for processing payments. Using polymorphism, we can create a common interface in the form of a base class PaymentMethod, and then override a processPayment() method in each derived class to implement the specific payment logic for each type.

```
#include <iostream>

using namespace std;


class PaymentMethod {

public:

   virtual void processPayment() = 0;  // Pure virtual function

};


class CreditCard : public PaymentMethod {

public:

   void processPayment() override {

      cout << "Processing credit card payment..." << endl;
```

```
    }
};

class PayPal : public PaymentMethod {
public:
    void processPayment() override {
        cout << "Processing PayPal payment..." << endl;
    }
};

class Cryptocurrency : public PaymentMethod {
public:
    void processPayment() override {
        cout << "Processing cryptocurrency payment..." << endl;
    }
};

int main() {
    PaymentMethod* payment;

    CreditCard creditCard;
```

```
    PayPal payPal;
    Cryptocurrency crypto;

    payment = &creditCard;
    payment->processPayment();  // Calls CreditCard's processPayment

    payment = &payPal;
    payment->processPayment();  // Calls PayPal's processPayment

    payment = &crypto;
    payment->processPayment();  // Calls Cryptocurrency's processPayment

    return 0;
}
```

In this case, polymorphism allows the store to process payments through any of the available methods without changing the core functionality, making the code more extensible and modular.

CONCLUSION

Encapsulation, inheritance, and polymorphism are the foundational pillars of object-oriented programming that enable developers to

write efficient, maintainable, and reusable code. Encapsulation helps in controlling access to object data, inheritance facilitates code reuse and hierarchical relationships, and polymorphism allows objects to take multiple forms, making it easier to extend and modify the behavior of a program. By mastering these concepts, you will be equipped to build robust OOP-based systems that are both flexible and scalable.

EXERCISES

1. **Encapsulation Exercise**:
    - Create a BankAccount class that encapsulates data like account balance and account number. Provide public getter and setter methods to interact with these attributes.
2. **Inheritance Exercise**:
    - Create a Shape class and derived classes like Circle, Rectangle, and Triangle. Implement methods for calculating the area and perimeter of each shape.
3. **Polymorphism Exercise**:
    - Write a program using polymorphism where different types of vehicles (e.g., Car, Bike, and Truck) inherit from a base class Vehicle. Override the fuelEfficiency() method in each class to display the fuel efficiency of each vehicle type.

OPERATOR OVERLOADING & FRIEND FUNCTIONS

CHAPTER 1: INTRODUCTION TO OPERATOR OVERLOADING

Operator overloading is an important feature in C++ that allows operators to be redefined to work with user-defined data types. This capability enables developers to make objects behave more intuitively and logically by defining operations on them as if they were built-in types.

In simple terms, operator overloading allows the same operator to have different meanings depending on the operands involved. For example, the + operator is normally used to add two numbers, but with operator overloading, it can be used to concatenate two strings, add two complex numbers, or merge two objects in a class.

Advantages of Operator Overloading

- **Improves Readability**: By enabling operators to work with user-defined types, code becomes more natural and easier to understand.
- **Enhances Code Reusability**: Operator functions can be reused across different parts of the program, reducing redundancy.
- **Encapsulation and Abstraction**: Operator overloading supports object-oriented principles by keeping operations related to an object within its class.
- **Efficiency**: With overloaded operators, expressions involving objects can be more efficient compared to using functions.

Example of Operator Overloading

Below is a simple C++ program demonstrating operator overloading:

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

class Complex {
   int real, imag;
public:
   Complex(int r = 0, int i = 0) { real = r; imag = i; }
   Complex operator+(const Complex& obj) {
      return Complex(real + obj.real, imag + obj.imag);
   }
   void display() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
   Complex c1(3, 4), c2(1, 2);
   Complex c3 = c1 + c2; // Operator Overloading
   c3.display();
   return 0;
}
```

This program overloads the + operator to add two complex numbers. The operator+ function returns a new object representing the sum of two complex numbers.

CHAPTER 2: TYPES OF OPERATOR OVERLOADING

Overloading Unary Operators

Unary operators operate on a single operand. These operators can be overloaded in C++ using member functions or friend functions.

Example: Overloading ++ operator (prefix and postfix)

```
#include <iostream>

using namespace std;


class Count {

   int value;

public:

   Count() { value = 0; }

   void operator++() { ++value; } // Prefix

   void operator++(int) { value++; } // Postfix

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

};


int main() {
```

```
    Count c;
    ++c;
    c.display();
    c++;
    c.display();
    return 0;
}
```

Overloading Binary Operators

Binary operators require two operands. They can be overloaded as member functions or non-member functions.

Example: Overloading * operator to multiply two objects:

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

class Multiply {
    int value;
public:
    Multiply(int v) { value = v; }
    Multiply operator*(const Multiply& obj) {
        return Multiply(value * obj.value);
    }
```

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

int main() {
    Multiply m1(4), m2(5);
    Multiply m3 = m1 * m2;
    m3.display();
    return 0;
}
```

CHAPTER 3: FRIEND FUNCTIONS

Understanding Friend Functions

A friend function is a special function that is allowed to access private and protected members of a class. Unlike member functions, a friend function is not called using an object. Instead, it is declared using the friend keyword in the class definition.

Advantages of Friend Functions

- They allow non-member functions to access private data.
- Useful when overloading binary operators that require access to private members.
- Enhances the flexibility of function implementation.

Example of a Friend Function

```
#include <iostream>

using namespace std;


class Box {

   int length;

public:

   Box(int l) { length = l; }

   friend int getLength(Box b);

};


int getLength(Box b) {

   return b.length; // Accessing private data

}


int main() {

   Box b1(10);

   cout << "Length of box: " << getLength(b1) << endl;

   return 0;

}
```

CHAPTER 4: CASE STUDY

Case Study: Implementing a Fraction Class

Consider a case where a developer needs to create a Fraction class that allows mathematical operations such as addition, subtraction, multiplication, and division using operator overloading.

Requirements:

1. Implement a class Fraction with private members numerator and denominator.
2. Overload the +, -, *, and / operators to perform arithmetic operations.
3. Ensure the fractions are simplified after each operation.
4. Use a friend function to access private members.

Solution:

```
#include <iostream>

using namespace std;


class Fraction {

  int num, den;

public:

  Fraction(int n, int d) { num = n; den = d; }

  friend Fraction operator+(const Fraction& f1, const Fraction& f2);

  void display() { cout << num << "/" << den << endl; }

};
```

```
Fraction operator+(const Fraction& f1, const Fraction& f2) {
    int n = f1.num * f2.den + f2.num * f1.den;
    int d = f1.den * f2.den;
    return Fraction(n, d);
}

int main() {
    Fraction f1(1, 2), f2(1, 3);
    Fraction f3 = f1 + f2;
    f3.display();
    return 0;
}
```

CHAPTER 5: EXERCISES

Exercise 1: Implement a Matrix Class

1. Create a Matrix class with private members rows, cols, and a dynamic 2D array.
2. Overload the + operator to add two matrices.
3. Overload the * operator to multiply two matrices.
4. Overload the << operator to display the matrix.

Exercise 2: Implement a Time Class

1. Create a Time class with private members hours and minutes.
2. Overload the + operator to add two time objects.
3. Overload the - operator to find the difference between two time objects.
4. Use a friend function to display the result.

VIRTUAL FUNCTIONS & DYNAMIC BINDING

VIRTUAL FUNCTIONS

In object-oriented programming (OOP), **virtual functions** are functions that are declared in a base class and are meant to be overridden in derived classes. The main feature of virtual functions is that they enable **dynamic polymorphism**, where the function to be invoked is determined at runtime based on the actual type of the object rather than the type of the pointer or reference used to call the function.

A function in a base class is made virtual by using the virtual keyword. When a function is marked as virtual, it allows derived classes to override it. This concept is a key feature of OOP that provides flexibility in handling different object types through the same interface. The primary benefit of using virtual functions is that they allow objects of derived classes to be treated as objects of the base class while still preserving the ability to call the overridden methods specific to the derived class.

In a typical class hierarchy, when a base class pointer or reference is used to call a method, the base class method is invoked by default. However, if the method is virtual, the program will invoke the method that is most appropriate based on the actual type of the object being pointed to or referenced.

Example: Virtual Functions in C++

```
#include <iostream>

using namespace std;
```

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

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

class Cat : public Animal {
public:
   void sound() override {
      cout << "Cat meows." << endl;
   }
};
```

```
int main() {
    Animal* animal;
    Dog dog;
    Cat cat;

    animal = &dog;
    animal->sound();  // Calls Dog's sound() due to virtual function

    animal = &cat;
    animal->sound();  // Calls Cat's sound() due to virtual function

    return 0;
}
```

In this example, the sound() function is marked as virtual in the Animal base class. Even though animal is a pointer of type Animal*, the correct sound() method is called based on the actual type of the object it points to—Dog or Cat. This demonstrates dynamic polymorphism, where the method call is resolved at runtime based on the type of the object.

Key Points About Virtual Functions

1. Virtual functions must be overridden in derived classes to exhibit dynamic behavior.
2. The base class function should generally be declared as virtual to allow derived classes to override it.

3. Virtual functions enable polymorphism, a core principle of OOP, by ensuring that the correct function is called based on the actual object type.

Dynamic Binding

Dynamic Binding, also referred to as late binding or runtime binding, is a process in which the method to be invoked is determined at runtime. This is in contrast to **static binding**, where the method to be invoked is resolved during the compile time. Dynamic binding occurs when we use virtual functions, allowing for method calls to be resolved based on the actual object type at runtime rather than the type of the reference or pointer used.

The primary advantage of dynamic binding is that it allows for more flexible and reusable code, as the same function call can perform different operations depending on the object's actual class. It is one of the key mechanisms behind polymorphism in OOP, enabling objects of different classes to be treated in a uniform way while maintaining their specific behaviors.

Dynamic binding is achieved through the use of virtual functions in C++. When a base class pointer or reference is used to invoke a virtual function, the compiler does not know which function to call until runtime, which is when the actual type of the object pointed to is determined. This ensures that the correct version of the function is called, even when using base class pointers to handle derived class objects.

Example: Dynamic Binding in C++

```
#include <iostream>

using namespace std;
```

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

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

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

```
int main() {
    Shape* shape;
    Circle circle;
    Square square;

    shape = &circle;
    shape->draw();  // Resolves to Circle's draw()

    shape = &square;
    shape->draw();  // Resolves to Square's draw()

    return 0;
}
```

In this example, we define a base class Shape with a virtual method draw(). Derived classes Circle and Square override this method to provide their specific implementations. Using a base class pointer shape, we can call the draw() method for both Circle and Square objects, and dynamic binding ensures the correct method is called based on the actual type of the object.

Benefits of Dynamic Binding:

1. **Flexibility**: The same code can work with objects of different classes, allowing for code reuse.

2. **Extensibility**: New classes can be added without modifying the existing code, making the system more maintainable.
3. **Polymorphism**: The same function call can invoke different behaviors based on the object type, enabling polymorphism.

CASE STUDY: VIRTUAL FUNCTIONS & DYNAMIC BINDING IN A PAYMENT SYSTEM

Consider a payment processing system where various types of payments (e.g., CreditCardPayment, PayPalPayment, BitcoinPayment) inherit from a common base class PaymentMethod. Each payment type has a specific implementation for processing payments. Using virtual functions and dynamic binding, we can design a flexible system that handles different payment types uniformly.

Code Example: Payment Processing System

```
#include <iostream>

using namespace std;


class PaymentMethod {

public:

  virtual void processPayment() {

    cout << "Processing generic payment..." << endl;

  }

};
```

```
class CreditCardPayment : public PaymentMethod {
public:
    void processPayment() override {
        cout << "Processing payment through Credit Card..." << endl;
    }
};

class PayPalPayment : public PaymentMethod {
public:
    void processPayment() override {
        cout << "Processing payment through PayPal..." << endl;
    }
};

class BitcoinPayment : public PaymentMethod {
public:
    void processPayment() override {
        cout << "Processing payment through Bitcoin..." << endl;
    }
};
```

```
int main() {
    PaymentMethod* paymentMethod;
    CreditCardPayment creditCard;
    PayPalPayment payPal;
    BitcoinPayment bitcoin;

    // Processing payments using dynamic binding
    paymentMethod = &creditCard;
    paymentMethod->processPayment();  // Calls
CreditCardPayment's processPayment()

    paymentMethod = &payPal;
    paymentMethod->processPayment();  // Calls PayPalPayment's
processPayment()

    paymentMethod = &bitcoin;
    paymentMethod->processPayment();  // Calls BitcoinPayment's
processPayment()

    return 0;
}
```

In this system, we use virtual functions to enable dynamic binding. The processPayment() method in the PaymentMethod class is overridden in each derived class (CreditCardPayment, PayPalPayment, and BitcoinPayment). Regardless of the type of payment method, we can call the processPayment() method using a base class pointer, and the correct method will be invoked at runtime based on the actual object type. This approach makes the payment system scalable and flexible, as new payment methods can be added without modifying the existing codebase.

CONCLUSION

Virtual functions and dynamic binding are essential components of modern object-oriented systems, enabling flexibility, extensibility, and maintainability. Virtual functions allow derived classes to provide their specific implementations, while dynamic binding ensures that the correct method is called at runtime based on the actual object type. Together, these concepts enable polymorphism, making it possible to write code that can handle various object types uniformly, without needing to know the exact class of the objects involved. These principles are widely used in real-world applications such as payment systems, graphical user interfaces, and more, where flexibility and extensibility are crucial.

EXERCISES

1. **Virtual Functions Exercise:**
    - Create a class hierarchy with a base class Shape and derived classes Circle, Rectangle, and Triangle. Implement a virtual function area() that calculates the

area for each shape. Use dynamic binding to call the appropriate method.

2. **Dynamic Binding Exercise:**
    - Implement a banking system with a base class Account and derived classes SavingsAccount and CurrentAccount. Use virtual functions to implement calculateInterest() for both types of accounts and use dynamic binding to compute the interest for various account types.
3. **Case Study Exercise:**
    - Design an online store with a base class Product and derived classes Electronics, Clothing, and Furniture. Use virtual functions to calculate discounts based on the product type. Create a shopping cart that can hold different product types and calculate the total price, utilizing dynamic binding to handle each product type correctly.

ABSTRACT CLASSES & INTERFACES

ABSTRACT CLASSES

An **abstract class** in object-oriented programming (OOP) is a class that cannot be instantiated directly and is meant to be inherited by other classes. An abstract class is typically used to define a base class with common functionality and a common interface for derived classes. The purpose of an abstract class is to provide a common interface while allowing derived classes to implement specific details. Abstract classes contain at least one **pure virtual function,** which is a function that has no definition in the base class and must be implemented by derived classes.

An abstract class serves two purposes:

1. **Providing a common interface**: It establishes a set of methods that must be implemented in any subclass.
2. **Partial implementation**: It can also provide some default implementations, leaving certain functions to be overridden by subclasses as needed.

In C++, an abstract class is created by declaring one or more **pure virtual functions** using the = 0 syntax. Any class that contains a pure virtual function becomes abstract and cannot be instantiated.

Pure Virtual Function

A **pure virtual function** is a function that is declared in the base class but has no implementation in the base class. It is marked with the = 0 syntax, and any derived class must provide an implementation for it.

Example: Abstract Class in C++

```
#include <iostream>

using namespace std;


class Shape {

public:

   // Pure virtual function (making Shape an abstract class)

   virtual void draw() = 0;


   // A regular member function

   void display() {

      cout << "Displaying the shape" << endl;

   }

};


class Circle : public Shape {

public:

   // Implementing the pure virtual function

   void draw() override {

      cout << "Drawing a circle" << endl;

   }

};
```

```
class Square : public Shape {
public:
   // Implementing the pure virtual function
   void draw() override {
      cout << "Drawing a square" << endl;
   }
};

int main() {
   // Shape shape; // This would result in an error since Shape is abstract
   Circle circle;
   Square square;

   circle.draw();
   square.draw();

   return 0;
}
```

In this example, the class Shape is abstract because it contains a pure virtual function draw(). The Circle and Square classes inherit

from Shape and provide their specific implementations of the draw() function. We cannot instantiate the Shape class directly, as it is abstract.

Key Points About Abstract Classes

1. An abstract class cannot be instantiated directly.
2. Abstract classes are meant to be inherited by derived classes.
3. Abstract classes can contain both pure virtual functions and regular functions.
4. A derived class that inherits an abstract class must implement all pure virtual functions, or it will also be considered abstract.

Interfaces

An **interface** in programming is a contract or blueprint that defines a set of methods without implementing them. In C++, an interface is typically implemented using an abstract class where all member functions are pure virtual functions. An interface specifies what methods a class should implement but does not provide any implementation for them. This allows any class that implements the interface to define its own specific behavior for the methods.

An interface is not tied to any specific class hierarchy. It is used to define capabilities that can be implemented by any class, irrespective of the class hierarchy. The purpose of an interface is to establish a common set of methods that different classes can implement, even if they are unrelated in terms of class inheritance.

While C++ does not have a keyword specifically for interfaces, we achieve the same effect by creating an abstract class with only pure virtual functions.

Interface Example:

```
#include <iostream>

using namespace std;


class Drawable {

public:

  // Pure virtual function to be implemented by any class that inherits Drawable

  virtual void draw() = 0;

};


class Circle : public Drawable {

public:

  void draw() override {

    cout << "Drawing a circle" << endl;

  }

};


class Square : public Drawable {

public:

  void draw() override {
```

```
        cout << "Drawing a square" << endl;
    }
};

int main() {
    Circle circle;
    Square square;

    Drawable* drawable1 = &circle;
    Drawable* drawable2 = &square;

    drawable1->draw();  // Drawing a circle
    drawable2->draw();  // Drawing a square

    return 0;
}
```

In this example, the Drawable class acts as an interface with a single pure virtual function draw(). Both Circle and Square implement the draw() function according to their specific needs. The Drawable pointer can point to any object that implements the draw() method, demonstrating the flexibility of interfaces.

Key Points About Interfaces

1. An interface defines a contract that classes must adhere to by implementing the specified methods.
2. Interfaces do not provide any implementation; they only declare method signatures.
3. Any class that implements an interface must provide implementations for all the interface's methods.
4. In C++, an interface is typically implemented as an abstract class with only pure virtual functions.

Comparison Between Abstract Classes and Interfaces

While both abstract classes and interfaces allow defining common method signatures that must be implemented by derived classes, they differ in certain key ways:

Feature	Abstract Class	Interface
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Methods	Can have both pure virtual and concrete methods	All methods are pure virtual (no implementation)
Inheritance	A class can inherit from one abstract class	A class can implement multiple interfaces (multiple inheritance)
Purpose	Defines common functionality and behavior, allowing partial implementation	Defines a contract that classes must follow, without providing any implementation

Access Modifiers	Can have different access modifiers (public, protected, private) for data members and methods	Methods are implicitly public

---

Case Study: Using Abstract Classes and Interfaces in a Payment System

Consider a scenario where you are developing a payment processing system. The system supports multiple payment methods like CreditCardPayment, PayPalPayment, and BitcoinPayment. All of these payment methods need to implement a common interface for processing payments, but each payment method has its own specific logic for doing so.

You can use an **abstract class** to define common functionality for payment processing, such as validating payment details, while using an **interface** to define a common method signature for processing the payment.

```
#include <iostream>

using namespace std;


// Abstract class for common payment functionality

class PaymentProcessor {

public:

   virtual bool validatePaymentDetails() = 0;  // Pure virtual function
```

```
    void processTransaction() {
        cout << "Processing the transaction..." << endl;
    }
};

// Interface for payment methods
class PaymentMethod {
public:
    virtual void processPayment() = 0;  // Pure virtual function
};

class CreditCardPayment : public PaymentProcessor, public
PaymentMethod {
public:
    bool validatePaymentDetails() override {
        cout << "Validating credit card details..." << endl;
        return true;
    }

    void processPayment() override {
```

```
        cout << "Processing credit card payment..." << endl;
    }
};

class PayPalPayment : public PaymentProcessor, public
PaymentMethod {
public:
    bool validatePaymentDetails() override {
        cout << "Validating PayPal account..." << endl;
        return true;
    }

    void processPayment() override {
        cout << "Processing PayPal payment..." << endl;
    }
};

int main() {
    CreditCardPayment ccPayment;
    PayPalPayment ppPayment;
```

```
    ccPayment.validatePaymentDetails();
    ccPayment.processPayment();

    ppPayment.validatePaymentDetails();
    ppPayment.processPayment();

    return 0;
}
```

In this case study, the PaymentProcessor class is abstract and contains a common method (processTransaction()) and a pure virtual method (validatePaymentDetails()). The PaymentMethod interface defines the processPayment() method that all payment types must implement. The CreditCardPayment and PayPalPayment classes implement both the validatePaymentDetails() and processPayment() methods, ensuring that they conform to the requirements of both the abstract class and the interface.

CONCLUSION

Abstract classes and **interfaces** are fundamental concepts in object-oriented programming, particularly for designing flexible, extensible, and reusable systems. Abstract classes provide a way to define common behavior that can be partially implemented, while interfaces define a contract that must be adhered to by any class that implements them. Understanding the differences and appropriate use cases for abstract classes and interfaces will help

you write better, more maintainable code and design scalable systems.

Exercises

1. **Abstract Classes Exercise**:
    - Create an abstract class Employee with a pure virtual function calculateSalary(). Implement derived classes Manager and Developer that provide their own implementations of calculateSalary().
2. **Interface Exercise**:
    - Design an interface Drawable with a method draw(). Implement this interface in classes Circle, Square, and Triangle, and use an array of Drawable* to display all shapes.
3. **Payment System Exercise**:
    - Modify the case study to add a new payment method, such as BankTransferPayment. Implement the necessary methods in the PaymentProcessor class and the PaymentMethod interface.

Assignment Solution: Develop a C++ Program that Implements Different Types of Inheritance

In this assignment, we will demonstrate different types of inheritance in C++: **Single Inheritance**, **Multilevel Inheritance**, **Multiple Inheritance**, and **Hierarchical Inheritance**. Each type of inheritance will be implemented step by step, showcasing how C++ allows for code reuse and a hierarchical structure in class relationships.

Step-by-Step Guide to Solution

Step 1: Understand the Types of Inheritance

Before we dive into coding, let's briefly discuss each type of inheritance we will be implementing:

1. **Single Inheritance**: In this type, a class derives from a single base class.
2. **Multilevel Inheritance**: In this type, a class is derived from another derived class.
3. **Multiple Inheritance**: In this type, a class derives from more than one base class.
4. **Hierarchical Inheritance**: In this type, multiple classes derive from a single base class.

Step 2: Set Up Your Development Environment

1. Open your IDE (e.g., Code::Blocks, Visual Studio, or any text editor with a C++ compiler).
2. Create a new C++ file and name it inheritance_types.cpp.

Step 3: Implement the Program

Now, we will write the C++ program that demonstrates each of the inheritance types.

Single Inheritance Example

In single inheritance, one class (derived class) inherits from a single base class.

```
#include <iostream>

using namespace std;


// Base class

class Animal {

public:

  void eat() {

    cout << "This animal eats food." << endl;

  }

};


// Derived class

class Dog : public Animal {

public:

  void bark() {

    cout << "The dog barks." << endl;
```

```
    }
};

int main() {
    Dog dog; // Create a Dog object
    dog.eat(); // Call the base class method
    dog.bark(); // Call the derived class method
    return 0;
}
```

EXPLANATION:

- The Dog class is derived from the Animal class using the public inheritance.
- The Dog class inherits the eat() method from Animal and also adds its own method bark().
- We create a Dog object and call both the inherited method and the new method.

Multilevel Inheritance Example

In multilevel inheritance, a class is derived from another derived class.

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

// Base class
```

```
class Animal {
public:
  void eat() {
    cout << "This animal eats food." << endl;
  }
};

// Derived class
class Dog : public Animal {
public:
  void bark() {
    cout << "The dog barks." << endl;
  }
};

// Further derived class
class Puppy : public Dog {
public:
  void play() {
    cout << "The puppy plays." << endl;
  }
};
```

```
int main() {
   Puppy puppy; // Create a Puppy object
   puppy.eat();  // Call the method from Animal class
   puppy.bark(); // Call the method from Dog class
   puppy.play(); // Call the method from Puppy class
   return 0;
}
```

Explanation:

- The Puppy class is derived from Dog, which is in turn derived from Animal.
- The Puppy class inherits the eat() and bark() methods, and also has its own method play().
- We create a Puppy object and call methods from both Animal, Dog, and Puppy.

Multiple Inheritance Example

In multiple inheritance, a class can inherit from more than one base class.

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

// Base class 1
class Animal {
```

```
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

// Base class 2
class Mammal {
public:
    void hasFur() {
        cout << "This mammal has fur." << endl;
    }
};

// Derived class
class Dog : public Animal, public Mammal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};
```

```
int main() {
   Dog dog; // Create a Dog object
   dog.eat();   // Call the method from Animal class
   dog.hasFur(); // Call the method from Mammal class
   dog.bark();   // Call the method from Dog class
   return 0;
}
```

Explanation:

- The Dog class inherits from both Animal and Mammal classes.
- The Dog object has access to methods from both base classes (eat() from Animal and hasFur() from Mammal) as well as its own method bark().
- We create a Dog object and call methods from multiple base classes.

Hierarchical Inheritance Example

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

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

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

```
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

// Derived class 1
class Dog : public Animal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};

// Derived class 2
class Cat : public Animal {
public:
    void meow() {
        cout << "The cat meows." << endl;
    }
};

int main() {
```

```
    Dog dog;   // Create a Dog object
    Cat cat;   // Create a Cat object

    dog.eat(); // Call the method from Animal class
    dog.bark(); // Call the method from Dog class

    cat.eat(); // Call the method from Animal class
    cat.meow(); // Call the method from Cat class

    return 0;
}
```

Explanation:

- Both Dog and Cat classes inherit from the Animal class.
- Both derived classes have their specific methods (bark() for Dog and meow() for Cat).
- We create objects for both classes and demonstrate how they both inherit the eat() method from Animal.

STEP 4: COMPILE AND RUN THE PROGRAM

1. **Compile the Program**: After writing the code, save it and compile it using your C++ compiler.
    - In a terminal, you can use the following command to compile the program if you're using g++:
    - g++ inheritance_types.cpp -o inheritance_types

2. **Run the Program**: After compiling, you can run the program using the following command:
3. ./inheritance_types
4. **Expected Output**:
5. This animal eats food.
6. The dog barks.
7. This animal eats food.
8. The dog barks.
9. The puppy plays.
10. This animal eats food.
11. The dog barks.
12. The puppy plays.
13. This animal eats food.
14. This mammal has fur.
15. The dog barks.
16. This animal eats food.
17. The cat meows.

Concepts Covered in This Assignment:

1. **Single Inheritance**: A derived class inherits from a single base class.
2. **Multilevel Inheritance**: A class inherits from another derived class, creating a chain.

3. **Multiple Inheritance**: A class inherits from multiple base classes.
4. **Hierarchical Inheritance**: Multiple classes inherit from a single base class.

CONCLUSION

In this assignment, we have explored different types of inheritance in C++: **single inheritance, multilevel inheritance, multiple inheritance,** and **hierarchical inheritance**. Each type demonstrates how C++ supports different inheritance structures that promote code reuse and create relationships between classes. Understanding inheritance is crucial for building modular and maintainable code in object-oriented programming.

EXERCISES

1. **Inheritance Exercise**:
    - Create a class hierarchy where you have a base class Person with derived classes Employee and Manager. Implement methods to display information about employees and managers.
2. **Multiple Inheritance Exercise**:
    - Write a program that uses multiple inheritance where a class Student inherits from Person and Marks. Implement methods to calculate and display student grades.
3. **Real-World Application**:
    - Implement a class hierarchy for a vehicle system where the Vehicle class is the base class, and derived classes like

Car, Truck, and Motorcycle have specific methods to describe their characteristics.

Assignment Solution: Create a Simple Banking System Using OOP Concepts

In this assignment, we will design and implement a simple banking system using **Object-Oriented Programming (OOP)** concepts such as **classes**, **objects**, **encapsulation**, and **inheritance**. The banking system will include basic functionalities such as creating a bank account, depositing money, withdrawing money, and checking the balance.

We will also introduce a **derived class** for different types of accounts (e.g., **SavingAccount** and **CurrentAccount**) to demonstrate **inheritance**.

Step-by-Step Guide to Solution

Step 1: Define the Problem

Our banking system will have:

1. **BankAccount Class**: This will be the base class for handling the basic functionality of an account (e.g., deposit, withdraw, and check balance).
2. **SavingAccount and CurrentAccount Classes**: These will be derived classes inheriting from BankAccount. They will represent different types of accounts and may have additional features.

Step 2: Set Up Your Development Environment

1. Open your IDE (e.g., Code::Blocks, Visual Studio, or any text editor with a C++ compiler).
2. Create a new C++ file and name it banking_system.cpp.

Step 3: Implement the Program

We will now define the BankAccount class and the derived classes SavingAccount and CurrentAccount. Here's the implementation:

BankAccount Class (Base Class)

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

class BankAccount {
private:
  string accountHolder;
  double balance;

public:
  // Constructor to initialize the account holder's name and initial balance
  BankAccount(string name, double initialBalance) {
    accountHolder = name;
    balance = initialBalance > 0 ? initialBalance : 0;
  }

  // Deposit method to add money to the account
  void deposit(double amount) {
```

```
    if (amount > 0) {
        balance += amount;
        cout << "Deposited: $" << amount << endl;
    } else {
        cout << "Invalid deposit amount!" << endl;
    }
}

// Withdraw method to remove money from the account
void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        cout << "Withdrew: $" << amount << endl;
    } else {
        cout << "Invalid withdraw amount or insufficient balance!" <<
endl;
    }
}

// Method to check the account balance
void checkBalance() const {
    cout << "Current balance: $" << balance << endl;
}
```

```
    // Method to display account holder information
    void displayAccountInfo() const {
        cout << "Account Holder: " << accountHolder << endl;
        checkBalance();
    }
};
```

Explanation:

- The BankAccount class has private data members accountHolder and balance.
- We define methods for:
    - deposit: Adds money to the account.
    - withdraw: Subtracts money from the account.
    - checkBalance: Displays the current balance.
    - displayAccountInfo: Displays the account holder's name and current balance.

The constructor initializes the account holder's name and sets an initial balance for the account.

SavingAccount and CurrentAccount Classes (Derived Classes)

```
// Derived class for SavingAccount
class SavingAccount : public BankAccount {
private:
```

```
    double interestRate;

public:
    // Constructor to initialize the SavingAccount with interest rate
    SavingAccount(string name, double initialBalance, double rate)
        : BankAccount(name, initialBalance) {
        interestRate = rate > 0 ? rate : 0;
    }

    // Method to apply interest on the balance
    void applyInterest() {
        double interest = (interestRate / 100) * getBalance(); // Assuming getBalance is public or provided as a method
        deposit(interest);  // Adding interest to the balance
        cout << "Interest applied: $" << interest << endl;
    }

    void displayAccountInfo() const {
        cout << "Saving Account: ";
        BankAccount::displayAccountInfo();  // Call the base class method to show account info
    }
};
```

```
// Derived class for CurrentAccount
class CurrentAccount : public BankAccount {
private:
  double overdraftLimit;

public:
  // Constructor to initialize the CurrentAccount with overdraft limit
  CurrentAccount(string name, double initialBalance, double overdraft)
    : BankAccount(name, initialBalance) {
    overdraftLimit = overdraft > 0 ? overdraft : 0;
  }

  // Method to check if the withdrawal is within the overdraft limit
  void withdraw(double amount) {
    if (amount > 0 && (getBalance() + overdraftLimit) >= amount) {
      BankAccount::withdraw(amount);  // Call the base class method to withdraw
    } else {
      cout << "Invalid withdrawal amount or overdraft limit exceeded!" << endl;
    }
```

```
    }

    void displayAccountInfo() const {
        cout << "Current Account: ";
        BankAccount::displayAccountInfo(); // Call the base class method to show account info
    }
};
```

Explanation:

- The SavingAccount class inherits from the BankAccount class and adds an interest rate. It also includes an applyInterest() method to apply interest to the balance.
- The CurrentAccount class inherits from BankAccount and adds an overdraft limit. The withdraw() method is overridden to check if the withdrawal is within the overdraft limit before proceeding.

Step 4: Main Function to Demonstrate the Banking System

```
int main() {
    // Creating instances of different account types
    SavingAccount savingAcc("John Doe", 1000, 5);  // Saving account with 5% interest
    CurrentAccount currentAcc("Alice Smith", 500, 200);  // Current account with $200 overdraft limit
```

```
    // Demonstrating saving account operations
    savingAcc.displayAccountInfo();
    savingAcc.deposit(200);     // Deposit to saving account
    savingAcc.applyInterest();  // Apply interest
    savingAcc.withdraw(100);    // Withdraw from saving account
    savingAcc.checkBalance();   // Check balance

    // Demonstrating current account operations
    currentAcc.displayAccountInfo();
    currentAcc.deposit(300);    // Deposit to current account
    currentAcc.withdraw(700);     // Withdraw from current account within overdraft limit
    currentAcc.checkBalance();  // Check balance

    return 0;
}
```

Explanation:

- We create objects of SavingAccount and CurrentAccount.
- The SavingAccount object is initialized with a 5% interest rate, and the CurrentAccount object is initialized with a $200 overdraft limit.
- We perform various operations such as deposit, withdrawal, and applying interest for both types of accounts.

- We use the displayAccountInfo() method to display account information after each operation.

STEP 5: COMPILE AND RUN THE PROGRAM

1. **Compile the Program**: After writing the code, save the file and compile it using your C++ compiler.
    - In a terminal, you can use the following command to compile the program if you're using g++:
    - g++ banking_system.cpp -o banking_system
2. **Run the Program**: After compiling, you can run the program using the following command:
3. ./banking_system
4. **Expected Output**:
5. Saving Account:
6. Account Holder: John Doe
7. Current balance: $1000
8. Deposited: $200
9. Interest applied: $60
10. Withdrew: $100
11. Current balance: $1160
12.
13. Current Account:
14. Account Holder: Alice Smith
15. Current balance: $500

16. Deposited: $300
17. Withdrew: $700
18. Current balance: $100

Concepts Covered in This Assignment:

1. **Encapsulation**: Protecting the internal data of the BankAccount class and providing access through methods (deposit, withdraw, check balance).
2. **Inheritance**: The SavingAccount and CurrentAccount classes inherit from the BankAccount class and extend its functionality.
3. **Polymorphism**: The displayAccountInfo() method is overridden in the derived classes to show more specific account information.

CONCLUSION

In this assignment, we created a simple banking system using OOP principles like **encapsulation**, **inheritance**, and **polymorphism**. The BankAccount class serves as a base class, while the SavingAccount and CurrentAccount classes demonstrate inheritance by adding their specific behaviors. This system showcases how OOP can help structure and organize code in a way that promotes reusability, flexibility, and maintainability.

Exercises

1. **Add a feature to transfer money between accounts**:

- Implement a transferMoney method that allows transferring money from one account to another (i.e., transfer from currentAcc to savingAcc).

2. **Add an Account Type Enum**:
   - Add an enum for account types (SAVINGS, CURRENT) and modify the constructors of SavingAccount and CurrentAccount to set the account type.
3. **Implement Account Number and Validation**:
   - Add a unique account number to each account, and implement validation to ensure that the account number is unique.