**ASSIGNMENT**

**NAME: CYNTHIA NYAWIRA**

**REG NO: SCT121-0872/2022**

**COURSE: DIT**

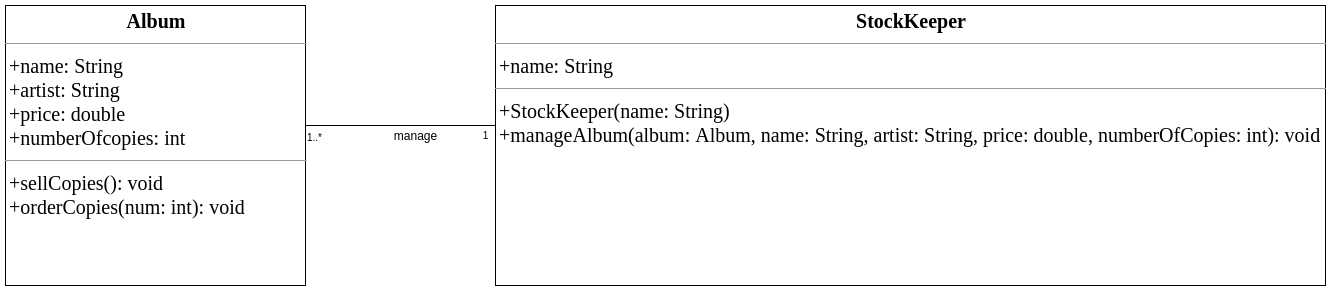
**UNIT: OBJECT ORIENTED PROGRAMMING**

**DIT: OOP Assignment**

**Part A:**

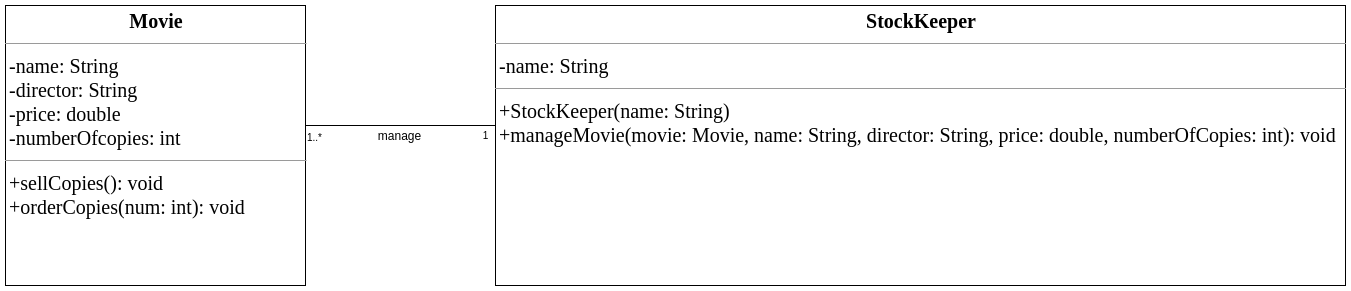
1. **Using a well labeled diagram, explain the steps of creating a system using OOP principles.**

1. Encapsulation: Encapsulation involves bundling the data (attributes) and the methods (behaviors) that operate on the data within a single unit, i.e., the class. It helps in data hiding and ensures that the internal workings of an object are hidden from the outside world.



2. Abstraction: Abstract away unnecessary details and focus on essential features relevant to the system. Create interfaces or abstract classes that define a set of methods without specifying their implementation. This allows for better organization and modularity.

3. Inheritance: Identify relationships between classes. If there are similarities between classes, you can use inheritance to create a hierarchy. This allows child classes to inherit attributes and methods from a parent class, reducing redundancy and promoting code reuse.



4. Polymorphism: This refers to the ability of different objects to respond to the same message or method call in different ways. It allows objects of different classes to be treated as objects of a common superclass through method overriding and method overloading.

1. **What is the Object Modeling Techniques (OMT).**

The Object Modeling Technique (OMT) is an object-oriented analysis, design, and implementation methodology that focuses creating a model of objects from the real world and then using this model to develop object-oriented software.

1. **Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP).**

OOA: we find and describe business objects or concepts in the problem domain OOD: we define how these software objects collaborate to meet the requirements. Attributes and methods. OOP: Implementation: we implement the design objects in, say, Java, C++, C#, etc.

1. **Discuss Main goals of UML.**

The primary goal of UML is to define some general-purpose simple modeling language so that all modelers can use and understand.

1. **DESCRIBE three advantages of using object oriented to develop an information system.**

Modularity and Reusability: OOP allows for the creation of modular code through the use of classes and objects. Modules or components can be built independently, making it easier to manage and maintain the codebase. Objects can be reused in different parts of the system or even in entirely different systems, promoting code reusability. Once a class is defined, it can be used to create multiple instances (objects) with their own unique data.

Encapsulation and Security: Encapsulation in OOP hides the internal workings of an object and allows access to its properties and methods only through a well-defined interface. This enhances security by preventing unintended manipulation of data, ensuring that only the designated methods can modify the object's state. It also prevents external code from directly accessing sensitive information.

Flexibility and Maintenance: OOP provides flexibility in adapting to changes and adding new features. Modifications can be made within a specific class without affecting other parts of the system, reducing the risk of unintended consequences.

1. **Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept. [12 Marks]**
   1. **Constructor**

A constructor is a special method in a class that is automatically invoked when an object of that class is created. Its purpose is to initialize the object's state or set up its initial values.

// Constructor example

class Car {

private String brand;

// Constructor

public Car(String carBrand) {

brand = carBrand;

}

public void displayBrand() {

System.out.println("Car brand: " + brand);

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of the Car class using the constructor

Car myCar = new Car("Toyota");

myCar.displayBrand(); // Output: Car brand: Toyota

}

}

* 1. **object**

An object is a fundamental unit of a class and represents a real-world entity. It is an instance of a class that encapsulates data (attributes) and behaviors (methods).

// Object example

class Student {

private String name;

public Student(String studentName) {

name = studentName;

}

public void displayInfo() {

System.out.println("Student name: " + name);

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of the Student class

Student newStudent = new Student("Alice");

newStudent.displayInfo(); // Output: Student name: Alice

}

}

* 1. **Destructor**

In Java, there's no explicit destructor like in some other languages. Java uses automatic garbage collection to reclaim memory occupied by objects that are no longer referenced.

* 1. **polymorphism**

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used for entities of different types.

// Polymorphism example

class Animal {

public void makeSound() {

System.out.println("Some sound");

}

}

class Dog extends Animal {

public void makeSound() {

System.out.println("Woof");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Dog(); // Polymorphic behavior

myAnimal.makeSound(); // Output: Woof

}

}

* 1. **class**

A class is a blueprint or template that defines the properties (attributes) and behaviors (methods) common to all objects of that type.

* 1. **Inheritance**

Inheritance is a mechanism in which a new class (derived or child class) is created by inheriting the features (attributes and methods) of an existing class (base or parent class).

// Inheritance example

class Vehicle {

protected String brand;

public void displayBrand() {

System.out.println("Vehicle brand: " + brand);

}

}

class Car extends Vehicle {

public Car(String carBrand) {

brand = carBrand;

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car("Honda");

myCar.displayBrand(); // Output: Vehicle brand: Honda

}

}

**vi. *EXPLAIN* the three types of associations (relationships) between objects in object oriented. [6 Marks]**

1. One-to-One (1:1) Relationship: In a one-to-one association, a single instance of one class is associated with a single instance of another class. Each object of one class is related to exactly one object of another class.

2. One-to-Many (1:N) Relationship: In a one-to-many association, a single instance of one class is associated with multiple instances of another class. However, each instance of the latter class is associated with only one instance of the former class.

3. Many-to-Many (N:M) Relationship: In a many-to-many association, multiple instances of one class are associated with multiple instances of another class. Both classes can have multiple associations with each other.

**Vii. What do you mean by class diagram? Where it is used and also discuss the steps to draw the class diagram with any one example.**

Uses of Class Diagram:

Designing Software Systems: It helps in planning the structure of software systems before implementation.

Communicating System Structure: Developers use it to communicate and understand the architecture of a system.

Modeling Business Processes: It's used in modeling business domains and their relationships.

Steps to Draw a Class Diagram:

1. Identify Classes: Determine the main entities or classes in the system. For instance, in a simple library system, classes could include Book, Author, Library, etc.

2. Identify Attributes: Define the attributes or properties of each class. For the Book class, attributes might include title, author, ISBN, genre, etc.

3. Define Methods: Determine the methods or functions associated with each class. For Book, methods might include checkOut(), returnBook(), calculateLateFee(), etc.

4. Determine Relationships: Identify relationships between classes. For example, a Library class might have a relationship with the Book class through an association like "contains" or "has."

5. Draw Classes and Relationships: Using a modeling tool or paper, draw boxes representing each class. Inside these boxes, list the class name, attributes, and methods. Use lines with appropriate notations (such as arrows) to depict relationships between classes.

Example: Library Management System

Classes:

Book: Attributes - title, author, ISBN; Methods - checkOut(), returnBook()

Author: Attributes - name, biography; Methods - getBooks()

Library: Attributes - name, location; Methods - addBook(), removeBook()

Relationships:

Library has a relationship with Book through an association, indicating that the library contains books.

Drawing the diagram involves creating boxes for each class (Book, Author, Library) and listing their attributes and methods inside. Then, draw lines to indicate relationships between these classes, such as an arrow from Library to Book with a label showing the association.

1. **Given that you are creating area and perimeter calculator using C++, to computer area and perimeter of various shaped like Circles, Rectangle, Triangle and Square, use well written code to explain and implement the calculator using the following OOP concepts.**
   1. **Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance)**

#include <iostream>

// Base class for shapes

class Shape {

public:

virtual float calculateArea() = 0;

virtual float calculatePerimeter() = 0;

};

// Derived class Circle inheriting from Shape

class Circle : public Shape {

private:

float radius;

public:

Circle(float r) : radius(r) {}

float calculateArea() override {

return 3.14 \* radius \* radius;

}

float calculatePerimeter() override {

return 2 \* 3.14 \* radius;

}

};

// Derived class Rectangle inheriting from Shape

class Rectangle : public Shape {

private:

float length, width;

public:

Rectangle(float l, float w) : length(l), width(w) {}

float calculateArea() override {

return length \* width;

}

float calculatePerimeter() override {

return 2 \* (length + width);

}

};

// Derived class Triangle inheriting from Shape

class Triangle : public Shape {

private:

float side1, side2, side3;

public:

Triangle(float s1, float s2, float s3) : side1(s1), side2(s2), side3(s3) {}

float calculateArea() override {

float s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

float calculatePerimeter() override {

return side1 + side2 + side3;

}

};

// Derived class Square inheriting from Rectangle

class Square : public Rectangle {

public:

Square(float side) : Rectangle(side, side) {}

};

int main() {

Circle c(5);

std::cout << "Circle - Area: " << c.calculateArea() << ", Perimeter: " << c.calculatePerimeter() << std::endl;

Rectangle r(4, 6);

std::cout << "Rectangle - Area: " << r.calculateArea() << ", Perimeter: " << r.calculatePerimeter() << std::endl;

Triangle t(3, 4, 5);

std::cout << "Triangle - Area: " << t.calculateArea() << ", Perimeter: " << t.calculatePerimeter() << std::endl;

Square s(5);

std::cout << "Square - Area: " << s.calculateArea() << ", Perimeter: " << s.calculatePerimeter() << std::endl;

return 0;

}

This code demonstrates single inheritance where Circle, Rectangle, Triangle, and Square classes inherit from the Shape class. Each derived class implements its own calculateArea() and calculatePerimeter() methods as required for the respective shapes.

* 1. **Friend functions**

#include <iostream>

#include <cmath>

// Forward declarations

class Circle;

class Rectangle;

class Triangle;

class Square;

// Base class for shapes

class Shape {

public:

virtual float calculateArea() = 0;

virtual float calculatePerimeter() = 0;

};

// Friend function to calculate area of Square (using friend classes)

float calculateAreaSquare(const Square& s);

// Friend function to calculate perimeter of Triangle (using friend functions)

float calculatePerimeterTriangle(const Triangle& t);

// Derived class Circle

class Circle : public Shape {

private:

float radius;

public:

Circle(float r) : radius(r) {}

float calculateArea() override {

return 3.14 \* radius \* radius;

}

float calculatePerimeter() override {

return 2 \* 3.14 \* radius;

}

friend float calculateAreaSquare(const Square& s); // Friend function declaration

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

float length, width;

public:

Rectangle(float l, float w) : length(l), width(w) {}

float calculateArea() override {

return length \* width;

}

float calculatePerimeter() override {

return 2 \* (length + width);

}

};

// Derived class Triangle

class Triangle : public Shape {

private:

float side1, side2, side3;

public:

Triangle(float s1, float s2, float s3) : side1(s1), side2(s2), side3(s3) {}

float calculateArea() override {

float s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

friend float calculatePerimeterTriangle(const Triangle& t); // Friend function declaration

};

// Derived class Square

class Square : public Shape {

private:

float side;

public:

Square(float s) : side(s) {}

float calculateArea() override {

return calculateAreaSquare(\*this); // Using friend function for area calculation

}

float calculatePerimeter() override {

return 4 \* side;

}

friend float calculateAreaSquare(const Square& s); // Friend function definition

};

// Friend function definition for Square to calculate its area using Circle's formula

float calculateAreaSquare(const Square& s) {

Circle temp(s.side \* sqrt(2) / 2); // Inscribing a circle in the square

return temp.calculateArea();

}

// Friend function definition for Triangle to calculate its perimeter

float calculatePerimeterTriangle(const Triangle& t) {

return t.side1 + t.side2 + t.side3;

}

int main() {

Circle c(5);

std::cout << "Circle - Area: " << c.calculateArea() << ", Perimeter: " << c.calculatePerimeter() << std::endl;

Rectangle r(4, 6);

std::cout << "Rectangle - Area: " << r.calculateArea() << ", Perimeter: " << r.calculatePerimeter() << std::endl;

Triangle t(3, 4, 5);

std::cout << "Triangle - Area: " << t.calculateArea() << ", Perimeter: " << calculatePerimeterTriangle(t) << std::endl;

Square s(5);

std::cout << "Square - Area: " << s.calculateArea() << ", Perimeter: " << s.calculatePerimeter() << std::endl;

return 0;

}

* 1. **Method overloading and method overriding**

Method overriding is a fundamental aspect of object-oriented programming (OOP) where a subclass provides a specific implementation of a method that is already present in its superclass. In the context of an area and perimeter calculator for shapes, method overriding allows each shape to provide its own implementation for calculating area and perimeter, even though they share a common method signature.

#include <iostream>

#include <cmath>

// Base class Shape

class Shape {

public:

virtual double area() const = 0;

virtual double perimeter() const = 0;

virtual ~Shape() {} // virtual destructor for polymorphic behavior

};

// Circle class

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

double area() const override {

return M\_PI \* radius \* radius;

}

double perimeter() const override {

return 2 \* M\_PI \* radius;

}

};

// Rectangle class

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double area() const override {

return length \* width;

}

double perimeter() const override {

return 2 \* (length + width);

}

};

// Triangle class

class Triangle : public Shape {

private:

double side1;

double side2;

double side3;

public:

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

double area() const override {

// Heron's formula for the area of a triangle

double s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

double perimeter() const override {

return side1 + side2 + side3;

}

};

// Square class (inherits from Rectangle)

class Square : public Rectangle {

public:

Square(double side) : Rectangle(side, side) {}

};

// Function to display shape details

void displayDetails(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

std::cout << "Perimeter: " << shape.perimeter() << std::endl;

std::cout << std::endl;

}

int main() {

// Creating instances of different shapes

Circle circle(5);

Rectangle rectangle(4, 6);

Triangle triangle(3, 4, 5);

Square square(5);

// Displaying details of each shape

std::cout << "Circle Details:" << std::endl;

displayDetails(circle);

std::cout << "Rectangle Details:" << std::endl;

displayDetails(rectangle);

std::cout << "Triangle Details:" << std::endl;

displayDetails(triangle);

std::cout << "Square Details:" << std::endl;

displayDetails(square);

return 0;

* 1. **Late binding and early binding**

In object-oriented programming (OOP), binding refers to the association between a method call and the actual code that gets executed. C++ uses both early binding (also known as static binding) and late binding (also known as dynamic binding or runtime polymorphism) based on the context of the method calls and object types involved.

Early Binding (Static Binding):

Early binding occurs when the compiler determines at compile-time which function to call based on the declared type of the object. It is resolved before the program runs.

Late Binding (Dynamic Binding / Runtime Polymorphism):

Late binding occurs when the determination of which function to call happens at runtime based on the actual object type. This is achieved using virtual functions and is typical when dealing with inheritance and polymorphism.

Here's a code example demonstrating both early binding and late binding using C++:

#include <iostream>

#include <cmath>

// Base class Shape

class Shape {

public:

// Virtual functions for area and perimeter

virtual double area() const = 0;

virtual double perimeter() const = 0;

virtual ~Shape() {} // Virtual destructor for polymorphic behavior

};

// Circle class

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

double area() const override {

return M\_PI \* radius \* radius;

}

double perimeter() const override {

return 2 \* M\_PI \* radius;

}

};

// Rectangle class

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double area() const override {

return length \* width;

}

double perimeter() const override {

return 2 \* (length + width);

}

};

// Function to calculate and display area and perimeter

void displayDetails(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

std::cout << "Perimeter: " << shape.perimeter() << std::endl;

std::cout << std::endl;

}

int main() {

// Early Binding - Static Binding

Circle circle(5);

Rectangle rectangle(4, 6);

// Calls to displayDetails use early binding as the object types are known at compile-time

std::cout << "Circle Details (Early Binding):" << std::endl;

displayDetails(circle);

std::cout << "Rectangle Details (Early Binding):" << std::endl;

displayDetails(rectangle);

// Late Binding - Dynamic Binding (Polymorphism)

Shape\* shapes[2];

shapes[0] = new Circle(3);

shapes[1] = new Rectangle(2, 4);

// Calls to displayDetails use late binding as the object types are determined at runtime

for (int i = 0; i < 2; ++i) {

std::cout << "Shape " << i + 1 << " Details (Late Binding):" << std::endl;

displayDetails(\*shapes[i]);

}

// Clean up dynamically allocated memory

for (int i = 0; i < 2; ++i) {

delete shapes[i];

}

return 0;

}

In this example, the main() function demonstrates both early and late binding scenarios:

Early Binding: Objects circle and rectangle are created and passed to displayDetails(). The function calls are resolved at compile-time based on the declared types.

Late Binding: An array of Shape pointers holds instances of Circle and Rectangle. The function displayDetails() is called through these pointers, demonstrating late binding as the actual function to call is determined at runtime based on the object's true type.

Late binding through virtual functions enables polymorphic behavior, allowing flexibility in handling different object types through a common interface (Shape in this case). This way, the appropriate method is called based on the actual type of the object, even when accessed through a base class pointer or reference.

* 1. **Abstract class and pure functions [6 Marks]**

Abstract classes and pure virtual functions are crucial in C++ for defining a common interface among various classes while ensuring that each subclass implements its own version of the required methods. Abstract classes cannot be instantiated on their own but serve as a blueprint for other classes to inherit from.

Here's an example demonstrating the use of an abstract class with pure virtual functions for calculating the area and perimeter of shapes:

#include <iostream>

#include <cmath>

// Abstract base class Shape

class Shape {

public:

// Pure virtual functions to calculate area and perimeter

virtual double area() const = 0;

virtual double perimeter() const = 0;

virtual ~Shape() {} // Virtual destructor for polymorphic behavior

};

// Circle class

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

double area() const override {

return M\_PI \* radius \* radius;

}

double perimeter() const override {

return 2 \* M\_PI \* radius;

}

};

// Rectangle class

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double area() const override {

return length \* width;

}

double perimeter() const override {

return 2 \* (length + width);

}

};

// Triangle class

class Triangle : public Shape {

private:

double side1;

double side2;

double side3;

public:

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

double area() const override {

// Heron's formula for the area of a triangle

double s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

double perimeter() const override {

return side1 + side2 + side3;

}

};

// Square class (inherits from Rectangle)

class Square : public Rectangle {

public:

Square(double side) : Rectangle(side, side) {}

};

// Function to display shape details

void displayDetails(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

std::cout << "Perimeter: " << shape.perimeter() << std::endl;

std::cout << std::endl;

}

int main() {

// Creating instances of different shapes

Circle circle(5);

Rectangle rectangle(4, 6);

Triangle triangle(3, 4, 5);

Square square(5);

// Displaying details of each shape

std::cout << "Circle Details:" << std::endl;

displayDetails(circle);

std::cout << "Rectangle Details:" << std::endl;

displayDetails(rectangle);

std::cout << "Triangle Details:" << std::endl;

displayDetails(triangle);

std::cout << "Square Details:" << std::endl;

displayDetails(square);

return 0;

}

In this code:

Shape is an abstract base class that contains pure virtual functions area() and perimeter().

Each shape class (Circle, Rectangle, Triangle, and Square) inherits from Shape and implements these pure virtual functions according to its own specific formulae.

The main() function demonstrates creating instances of each shape and displaying their area and perimeter using the displayDetails() function, which takes a Shape reference.

The use of abstract classes and pure virtual functions ensures that any class inheriting from Shape must implement its own version of area() and perimeter(). This structure enforces a common interface among different shapes while allowing specific implementations for each shape's calculation methods.

1. **Using a program written in C++, differentiate between the following.** 
   1. **Function overloading and operator overloading**

In C++, both function overloading and operator overloading involve reusing names for different purposes, but they differ in how they're used and what they achieve.

Function Overloading:

Definition: Function overloading allows creating multiple functions in the same scope with the same name but different parameters or types.

Purpose: It enables using the same function name for different operations based on the type or number of arguments passed.

Example:

#include <iostream>

using namespace std;

// Function Overloading

void print(int num) {

cout << "Integer: " << num << endl;

}

void print(float num) {

cout << "Float: " << num << endl;

}

int main() {

print(5); // Calls print(int)

print(3.14f); // Calls print(float)

return 0;

}

Operator Overloading:

Definition: Operator overloading allows defining custom behaviors for operators like +, -, \*, /, etc., for user-defined types.

Purpose: It enables the use of operators with user-defined types, making the code more intuitive and readable.

Example:

#include <iostream>

using namespace std;

class Complex {

private:

float real, imag;

public:

Complex(float r, float i) : real(r), imag(i) {}

// Operator Overloading for '+'

Complex operator+(const Complex& c) {

Complex temp(0, 0);

temp.real = real + c.real;

temp.imag = imag + c.imag;

return temp;

}

void display() {

cout << "Real: " << real << " Imaginary: " << imag << endl;

}

};

int main() {

Complex c1(2.5, 3.5);

Complex c2(1.6, 2.6);

Complex c3 = c1 + c2; // Calls operator+

c3.display();

return 0;

}

Key Differences:

Function Overloading involves using the same function name with different parameters or types within the same scope.

Operator Overloading involves defining custom behaviors for operators when used with user-defined types, allowing these operators to work with objects of those types.

Both techniques provide flexibility and readability to code by allowing the reuse of names while catering to different scenarios, whether it's for functions with different parameters or enabling operators to work with custom types.

* 1. **Pass by value and pass by reference**

In C++, when passing arguments to functions, you can use two different methods: pass by value and pass by reference. They fundamentally differ in how they handle and manipulate the data passed to a function.

Pass by Value:

Definition: Pass by value involves sending a copy of the actual data to the function.

Method: When an argument is passed by value, a copy of the variable's value is made and passed to the function. Any modifications made within the function do not affect the original variable.

Example:

#include <iostream>

using namespace std;

void increment(int num) {

num++; // Incrementing the copied value

cout << "Inside function: " << num << endl;

}

int main() {

int value = 5;

increment(value); // Pass by value

cout << "Outside function: " << value << endl;

return 0;

}

Output

Inside function: 6

Outside function: 5

Pass by Reference:

Definition: Pass by reference involves passing the memory address (reference) of the actual variable to the function.

Method: When an argument is passed by reference, any changes made to the parameter within the function affect the original variable directly as they share the same memory location.

Example:

#include <iostream>

using namespace std;

void incrementByReference(int &num) {

num++; // Incrementing the original variable through reference

cout << "Inside function: " << num << endl;

}

int main() {

int value = 5;

incrementByReference(value); // Pass by reference

cout << "Outside function: " << value << endl;

return 0;

}

Output

Inside function: 6

Outside function: 6

* 1. **Parameters and arguments**

In programming, particularly in functions and methods, parameters and arguments are key concepts that relate to passing and receiving data.

Parameters:

Definition: Parameters refer to the variables declared in the function/method signature that define what kind of data a function expects to receive when it is called.

Role: They act as placeholders for the values that will be passed into the function when it is invoked.

Example:

#include <iostream>

using namespace std;

// Function with parameters

void greet(string name, int age) {

cout << "Hello, " << name << "! You are " << age << " years old." << endl;

}

int main() {

string personName = "Alice";

int personAge = 30;

greet(personName, personAge); // Arguments passed to parameters

return 0;

}

In this example, name and age in the function greet are parameters.

Arguments:

Definition: Arguments are the actual values or expressions passed to a function when it is called, fulfilling the placeholders (parameters) defined in the function's signature.

Role: They provide the real data that the function will work with during its execution.

Example:

#include <iostream>

using namespace std;

// Function with parameters

void greet(string name, int age) {

cout << "Hello, " << name << "! You are " << age << " years old." << endl;

}

int main() {

string personName = "Bob";

int personAge = 25;

greet(personName, personAge); // Arguments passed to parameters

return 0;

}

In this case, personName and personAge are the arguments passed to the greet function.

Key Differences:

Parameters are variables declared in a function's signature that define the type and number of values the function expects.

Arguments are the actual values or expressions passed into a function when it is called, matching the parameters' types and order.

**Create a new class called *CalculateG.*Copy and paste the following initial version of the code. Note variables declaration and the types.**

**class** *CalculateG* **{  
int** main**(){**

(*datatype*) gravity =-9.81; // Earth's gravity in m/s^2 (*datatype*) fallingTime = 30;

(*datatype*)initialVelocity = 0.0; (*datatype*) finalVelocity = ;

(*datatype*) initialPosition = 0.0; (*datatype*) finalPosition = ;

// Add the formulas for position and velocity

Cout<<"The object's position after " << fallingTime << " seconds is "

+ finalPosition + << m."<<endl;

// Add output line for velocity (similar to position)

} }

#include <iostream>

using namespace std;

class CalculateG {

public:

void calculateGravity() {

double gravity = -9.81; // Earth's gravity in m/s^2

double fallingTime = 30;

double initialVelocity = 0.0;

double finalVelocity;

double initialPosition = 0.0;

double finalPosition;

// Calculating final position using the formula: s = ut + 0.5 \* a \* t^2

finalPosition = initialPosition + (initialVelocity \* fallingTime) + (0.5 \* gravity \* fallingTime \* fallingTime);

// Calculating final velocity using the formula: v = u + a \* t

finalVelocity = initialVelocity + (gravity \* fallingTime);

cout << "The object's position after " << fallingTime << " seconds is " << finalPosition << " m." << endl;

cout << "The object's velocity after " << fallingTime << " seconds is " << finalVelocity << " m/s." << endl;

}

};

int main() {

CalculateG calculator;

calculator.calculateGravity();

return 0;

}

Modify the example program to compute the position and velocity of an object after falling for 30 seconds, outputting the position in meters. The formula in Math notation is:

𝑥(𝑡)=0.5∗𝑎𝑡2 +𝑣𝑖𝑡+𝑥𝑖 𝑣(𝑡)=𝑎𝑡+𝑣𝑖

Run the completed code in Eclipse (Run → Run As → Java Application). 5. Extend *datatype* class with the following code:

**public class** *CalculateG* {

**public double** multi(**......**){ // method for multiplication

}

// add 2 more methods for powering to square and summation (similar to multiplication)

**public void** outline(**......**){  
// method for printing out a result

}  
**int** main() {

// compute the position and velocity of an object with defined methods and print out the

result

} }

public class CalculateG {

public double position(double a, double vInitial, double xInitial, double t) {

return 0.5 \* a \* Math.pow(t, 2) + vInitial \* t + xInitial;

}

public double velocity(double a, double vInitial, double t) {

return a \* t + vInitial;

}

public double multiply(double a, double b) {

return a \* b;

}

public double powerToSquare(double num) {

return Math.pow(num, 2);

}

public double summation(double a, double b) {

return a + b;

}

public void outline(String message) {

System.out.println(message);

}

public static void main(String[] args) {

CalculateG calculator = new CalculateG();

double gravity = -9.81;

double fallingTime = 30.0;

double initialVelocity = 0.0;

double initialPosition = 0.0;

double finalPosition = calculator.position(gravity, initialVelocity, initialPosition, fallingTime);

double finalVelocity = calculator.velocity(gravity, initialVelocity, fallingTime);

calculator.outline("The object's position after " + fallingTime + " seconds is " + finalPosition + " meters.");

calculator.outline("The object's velocity after " + fallingTime + " seconds is " + finalVelocity + " m/s."); }

}

6. Create methods for multiplication, powering to square, summation and printing out a result in *CalculateG* class.

public class CalculateG {

public double multiplication(double a, double b) {

return a \* b;

}

public double powerToSquare(double num) {

return Math.pow(num, 2);

}

public double summation(double a, double b) {

return a + b;

}

public void outline(String message) {

System.out.println(message);

}

public double position(double a, double vInitial, double xInitial, double t) {

return 0.5 \* a \* Math.pow(t, 2) + vInitial \* t + xInitial;

}

public double velocity(double a, double vInitial, double t) {

return a \* t + vInitial;

}

public static void main(String[] args) {

CalculateG calculator = new CalculateG();

double gravity = -9.81;

double fallingTime = 30.0;

double initialVelocity = 0.0;

double initialPosition = 0.0;

double finalPosition = calculator.position(gravity, initialVelocity, initialPosition, fallingTime);

double finalVelocity = calculator.velocity(gravity, initialVelocity, fallingTime);

calculator.outline("The object's position after " + fallingTime + " seconds is " + finalPosition + " meters.");

calculator.outline("The object's velocity after " + fallingTime + " seconds is " + finalVelocity + " m/s.");

}

}

In this version, the CalculateG class contains methods multiplication, powerToSquare, summation, and outline for performing multiplication, squaring a number, summation, and printing a message, respectively. The position and velocity calculation methods are also retained from the previous example. The main method computes the position and velocity of an object after falling for 30 seconds and prints the results using the outline method.

**Part B:**

Write a C++ program that takes 15 values of type integer as inputs from user, store the values in an array.

1. Print the values stored in the array on screen.

#include <iostream>

int main() {

const int SIZE = 15;

int values[SIZE];

// Taking input from the user

std::cout << "Enter 15 integer values:" << std::endl;

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

std::cout << "Enter value " << i + 1 << ": ";

std::cin >> values[i];

}

// Printing the values stored in the array

std::cout << "\nThe values stored in the array are:" << std::endl;

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

std::cout << values[i] << " ";

}

std::cout << std::endl;

return 0;

}

This program declares an array of size 15 (values) to store the user-input integers. It uses a loop to take input for each element of the array and another loop to print the values stored in the array.

1. Ask user to enter a number, check if that number (entered by user) is present in array or not. If it is present print, “the number found at index (index of the number) ” and the text “number not found in this array”

#include <iostream>

int main() {

const int SIZE = 15;

int values[SIZE];

// Taking input from the user

std::cout << "Enter 15 integer values:" << std::endl;

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

std::cout << "Enter value " << i + 1 << ": ";

std::cin >> values[i];

}

// Asking the user for a number to search

int numberToFind;

std::cout << "\nEnter a number to search in the array: ";

std::cin >> numberToFind;

bool found = false;

int index = -1;

// Searching for the number in the array

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

if (values[i] == numberToFind) {

found = true;

index = i;

break;

}

}

// Printing whether the number is found or not

if (found) {

std::cout << "The number " << numberToFind << " found at index " << index << std::endl;

} else {

std::cout << "Number not found in this array" << std::endl }

return 0;}

This program includes an additional section where it prompts the user to enter a number to search within the array. It then iterates through the array to check if the number exists. If found, it prints the index where the number is located; otherwise, it displays a message indicating that the number is not present in the array.

1. Create another array, copy all the elements from the existing array to the new array but in reverse order. Now print the elements of the new array on the screen

#include <iostream>

int main() {

const int SIZE = 15;

int values[SIZE];

// Taking input from the user

std::cout << "Enter 15 integer values:" << std::endl;

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

std::cout << "Enter value " << i + 1 << ": ";

std::cin >> values[i];

}

// Asking the user for a number to search

int numberToFind;

std::cout << "\nEnter a number to search in the array: ";

std::cin >> numberToFind;

bool found = false;

int index = -1;

// Searching for the number in the array

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

if (values[i] == numberToFind) {

found = true;

index = i;

break;

}

}

// Printing whether the number is found or not

if (found) {

std::cout << "The number " << numberToFind << " found at index " << index << std::endl;

} else {

std::cout << "Number not found in this array" << std::endl; }

// Creating a new array and copying elements in reverse order

int reversedValues[SIZE];

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

reversedValues[i] = values[SIZE - 1 - i]; }

// Printing elements of the new array (reversed)

std::cout << "\nThe elements of the new array (in reverse order) are:" << std::endl;

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

std::cout << reversedValues[i] << " ";

}

std::cout << std::endl;

return 0;}

This code now includes a new array named reversedValues. It copies elements from the original values array to this new array in reverse order. Afterward, it prints the elements of the new array (reversedValues) on the screen.

1. Get the sum and product of all elements of your array. Print product and the sum each on its own line.

#include <iostream>

int main() {

const int SIZE = 15;

int values[SIZE];

// Taking input from the user

std::cout << "Enter 15 integer values:" << std::endl;

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

std::cout << "Enter value " << i + 1 << ": ";

std::cin >> values[i];

}

// Asking the user for a number to search

int numberToFind;

std::cout << "\nEnter a number to search in the array: ";

std::cin >> numberToFind;

bool found = false;

int index = -1;

// Searching for the number in the array

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

if (values[i] == numberToFind) {

found = true;

index = i;

break;

}

}

// Printing whether the number is found or not

if (found) {

std::cout << "The number " << numberToFind << " found at index " << index << std::endl;

} else {

std::cout << "Number not found in this array" << std::endl;

}

// Creating a new array and copying elements in reverse order

int reversedValues[SIZE];

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

reversedValues[i] = values[SIZE - 1 - i];

}

// Printing elements of the new array (reversed)

std::cout << "\nThe elements of the new array (in reverse order) are:" << std::endl;

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

std::cout << reversedValues[i] << " ";

}

std::cout << std::endl;

// Calculating sum and product of elements in the original array

int sum = 0;

long long product = 1; // Using long long for product to avoid overflow for larger values

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

sum += values[i];

product \*= values[i];

}

// Printing the sum and product

std::cout << "\nThe sum of all elements: " << sum << std::endl;

std::cout << "The product of all elements: " << product << std::endl;

return 0;

}

This program calculates the sum and product of all elements in the array values and prints them separately on their own lines.