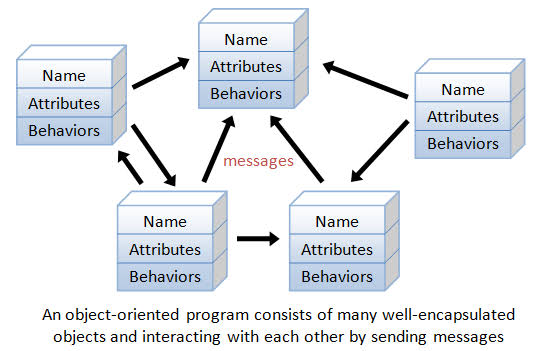
A.

1. 
2. OMT is a method for modeling systems using objects and is associated with Unified Modeling Language (UML). It captures the static and dynamic aspects of a system for software design and analysis.
3. OOAD focuses on the entire software development process, emphasizing requirements analysis and system design using object-oriented concepts. OOP, while related, has a broader scope, encompassing the application of object-oriented principles in programming without explicitly covering the entire software development lifecycle.
4. UML’s main goals are standardization, establishing a common modeling language, and visualization, providing a graphical notation for clear representation of system elements and behaviors.
5. 1. Modularity: Advantage: Enhances maintainability and reusability through the isolation of components, easing updates without affecting the entire system.

2. Reusability: Advantage: Reduces development time by promoting the reuse of tested components across different parts of the system or in new projects.

3. Encapsulation: Advantage: Improves data security and integrity by bundling data and methods into a single unit, fostering a robust and maintainable codebase.

VI. 1. Constructor:Constructors in C++ are special member functions that initialize objects when they are created.

#include <iostream>

using namespace std;

class Car {

public:

string model;

// Constructor

Car(string carModel) {

model = carModel;

}

};

int main() {

// Creating an object of the Car class

Car myCar("Toyota");

return 0;

}

2. Object:An object is an instance of a class in C++, representing a real-world entity.

#include <iostream>

using namespace std;

class Car {

public:

string model;

};

int main() {

// Creating an object of the Car class

Car myCar;

myCar.model = "Toyota";

return 0;

}

3. Destructor: In C++, destructors are special member functions used to clean up resources when an object is destroyed.

#include <iostream>

using namespace std;

class MyClass {

public:

// Destructor

~MyClass() {

// Clean up code here

}

};

int main() {

// Destructor called when the object goes out of scope

MyClass obj;

return 0;

}

4. Polymorphism:Polymorphism in C++ allows objects of different types to be treated as objects of a common type

#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;

}

};

5. Class: A class in C++ is a blueprint for creating objects, defining attributes and behaviors common to all objects of that type

#include <iostream>

using namespace std;

class Person {

public:

string name;

int age;

};

6. Inheritance: Inheritance in C++ allows a class to inherit attributes and methods from another class

#include <iostream>

using namespace std;

class Animal {

public:

void eat() {

cout << "Animal is eating" << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Dog is barking" << endl;

}

};

VII.1. Association:Represents a bi-directional relationship between two classes, where one class is associated with another.

- Can be one-to-one, one-to-many, or many-to-many.

- Each class remains independent and has its own lifecycle.

Class Teacher {

Public:

String name;

};

Class Student {

Public:

String name;

// Association with Teacher

Teacher\* instructor;

};

2. Aggregation:Describes a more specialized form of association where one class is part of another class.

- Represents a “whole-part” relationship.

- The parts can exist independently of the whole.

Class Engine {

Public:

String type;

};

Class Car {

Public:

String model;

// Aggregation with Engine

Engine\* carEngine;

};

3. Composition: Similar to aggregation but with a stronger relationship, where the part (sub-object) is integral to the whole (main object).

- Implies a strong ownership, and the part cannot exist without the whole.

- Often implemented using composition by value.

Class Heart {

Public:

String type;

};

Class Human {

Public:

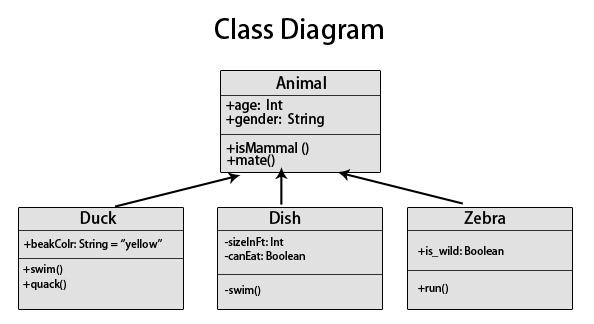
String name;

// Composition with Heart

Heart heart;

};

VIII.A class diagram is a visual representation of class objects in a model system, categorized by class types.



IX.

#include <iostream>

#include <cmath>

// Abstract class Shape

Class Shape {

Public:

Virtual double calculateArea() const = 0;

Virtual double calculatePerimeter() const = 0;

Virtual ~Shape() = default;

};

// Concrete class Circle inheriting from Shape

Class Circle : public Shape {

Private:

Double radius;

Public:

Circle(double r) : radius® {}

Double calculateArea() const override {

Return 3.14 \* radius \* radius;

}

Double calculatePerimeter() const override {

Return 2 \* 3.14 \* radius;

}

};

// Concrete class Rectangle inheriting from Shape

Class Rectangle : public Shape {

Private:

Double length, width;

Public:

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

Double calculateArea() const override {

Return length \* width;

}

Double calculatePerimeter() const override {

Return 2 \* (length + width);

}

};

// Concrete class Triangle inheriting from Shape

Class Triangle : public Shape {

Private:

Double side1, side2, side3;

Public:

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

Double calculateArea() 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 calculatePerimeter() const override {

Return side1 + side2 + side3;

}

};

// Concrete class Square inheriting from Rectangle (Single Inheritance)

Class Square : public Rectangle {

Public:

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

};

// Concrete class MultiShape inheriting from Circle and Rectangle (Multiple Inheritance)

Class MultiShape : public Circle, public Rectangle {

Public:

MultiShape(double circleRadius, double rectLength, double rectWidth)

: Circle(circleRadius), Rectangle(rectLength, rectWidth) {}

};

// Concrete class Animal (for Hierarchical Inheritance)

Class Animal {

Public:

Virtual void sound() const = 0;

Virtual ~Animal() = default;

};

// Derived class Dog inheriting from Animal

Class Dog : public Animal {

Public:

Void sound() const override {

Std::cout << “Woof!” << std::endl;

}

};

// Friend function to output area and perimeter

Void printAreaAndPerimeter(const Shape& shape) {

Std::cout << “Area: “ << shape.calculateArea() << std::endl;

Std::cout << “Perimeter: “ << shape.calculatePerimeter() << std::endl;

}

Int main() {

Circle circle(5);

Rectangle rectangle(4, 6);

Triangle triangle(3, 4, 5);

Square square(5);

MultiShape multiShape(2, 3, 4);

// Single Inheritance

Std::cout << “Circle – Single Inheritance” << std::endl;

printAreaAndPerimeter(circle);

// Multiple Inheritance

Std::cout << “\nMultiShape – Multiple Inheritance” << std::endl;

printAreaAndPerimeter(multiShape);

// Hierarchical Inheritance

Std::cout << “\nDog – Hierarchical Inheritance” << std::endl;

Dog dog;

Dog.sound();

Return 0;

}

X.Function Overloading and Operator Overloading:

#include <iostream>

Class OverloadingExample {

Public:

// Function Overloading

Void display(int x) {

Std::cout << “Displaying integer: “ << x << std::endl;

}

Void display(double y) {

Std::cout << “Displaying double: “ << y << std::endl;

}

Operator Overloading

OverloadingExample operator+(const OverloadingExample& other) {

OverloadingExample result;

Result.value = this->value + other.value;

Return result;

}

Private:

Int value;

};

Int main() {

OverloadingExample obj;

Obj.display(42); // Function Overloading

Obj.display(3.14); // Function Overloading

OverloadingExample obj1, obj2, obj3;

Obj1.display(5); // Function Overloading

Obj2.display(2.71); // Function Overloading

Obj3 = obj1 + obj2; // Operator Overloading

Return 0;

}

Pass by Value and Pass by Reference:

#include <iostream>

Void passByValue(int x) {

X = 10;

}

Void passByReference(int& y) {

Y = 20;

}

Int main() {

Int value1 = 5;

Int value2 = 5;

passByValue(value1); // Pass by Value

passByReference(value2); // Pass by Reference

std::cout << “Value after passByValue: “ << value1 << std::endl; // Output: 5

std::cout << “Value after passByReference: “ << value2 << std::endl; // Output: 20

return 0;

}

Parameters and Arguments:

#include <iostream>

// Parameters: x and y are parameters

Void add(int x, int y) {

Std::cout << “Sum: “ << x + y << std::endl;

}

Int main() {

Int a = 5;

Int b = 7;

Add(a, b); // Arguments: a and b are arguments

Return 0;

}

XI. a)Here’s the corrected version:

#include <iostream>

Class CalculateG {

Public:

Int main() {

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;

// Formulas for position and velocity

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

finalVelocity = gravity \* fallingTime + initialVelocity;

// Output for position

Std::cout << “The object’s position after “ << fallingTime << “ seconds is “ << finalPosition << “ m.” << std::endl;

// Output for velocity

Std::cout << “The object’s velocity after “ << fallingTime << “ seconds is “ << finalVelocity << “ m/s.” << std::endl;

Return 0;

}

}

b).

#include <iostream>

Class CalculateG {

Public:

// Constants

Static const double GRAVITY;

// Method for multiplication

Double multi(double a, double b) {

Return a \* b;

}

// Method for powering to square

Double square(double num) {

Return num \* num;

}

// Method for summation

Double sum(double a, double b) {

Return a + b;

}

// Method for calculating position

Double calculatePosition(double time, double initialVelocity, double initialPosition) {

Return 0.5 \* GRAVITY \* square(time) + multi(initialVelocity, time) + initialPosition;

}

// Method for calculating velocity

Double calculateVelocity(double time, double initialVelocity) {

Return multi(GRAVITY, time) + initialVelocity;

}

// Method for printing the result

Void outline(double position) {

Std::cout << “The object’s position is “ << position << “ meters.” << std::endl;

}

};

// Initialize gravity constant

Const double CalculateG::GRAVITY = -9.81;

Int main() {

CalculateG calculator;

Double fallingTime = 30.0;

Double initialVelocity = 0.0;

Double initialPosition = 0.0;

// Compute position and velocity

Double finalPosition = calculator.calculatePosition(fallingTime, initialVelocity, initialPosition);

Double finalVelocity = calculator.calculateVelocity(fallingTime, initialVelocity);

// Print the result

Calculator.outline(finalPosition);

Return 0;

}

B. 1)

#include <iostream>

Int main() {

Int limit = 4000000;

Int term1 = 1, term2 = 2, nextTerm = 0, sum = 0;

While (term1 <= limit) {

If (term1 % 2 == 0) {

Sum += term1;

}

nextTerm = term1 + term2;

term1 = term2;

term2 = nextTerm;

}

Std::cout << “Sum of even-valued Fibonacci terms below “ << limit << “: “ << sum << std::endl;

Return 0;

}

3)

#include<iostream>

Using namespace std;

Int main() {

Const int size = 15;

Int arr[size];

// Taking input from the user

Cout << “Enter 15 integers:\n”;

For (int I = 0; I < size; ++i) {

Cout << “Enter value “ << I + 1 << “: “;

Cin >> arr[i];

}

// Printing values stored in the array

Cout << “\nValues stored in the array:\n”;

For (int I = 0; I < size; ++i) {

Cout << arr[i] << “ “;

}

// Asking the user to enter a number

Int searchNumber;

Cout << “\nEnter a number to search: “;

Cin >> searchNumber;

// Checking if the number is present in the array

Bool found = false;

Int index;

For (int I = 0; I < size; ++i) {

If (arr[i] == searchNumber) {

Found = true;

Index = I;

Break;

}

}

// Printing the result of the search

If (found) {

Cout << “The number found at index “ << index << endl;

} else {

Cout << “Number not found in this array\n”;

}

// Creating another array and copying elements in reverse order

Int reverseArr[size];

For (int I = 0, j = size – 1; I < size; ++I, --j) {

reverseArr[j] = arr[i];

}

// Printing elements of the new array in reverse order

Cout << “\nElements of the new array in reverse order:\n”;

For (int I = 0; I < size; ++i) {

Cout << reverseArr[i] << “ “;

}

// Calculating sum and product of array elements

Int sum = 0;

Long long product = 1;

For (int I = 0; I < size; ++i) {

Sum += arr[i];

Product \*= arr[i];

}

// Printing sum and product

Cout << “\nSum of array elements: “ << sum << endl;

Cout << “Product of array elements: “ << product << endl;

Return 0;

}