Q-1: You are developing a geometry calculator application that can calculate the area of different shapes. To achieve this, you create a base class Shape and derived classes Circle and Rectangle.

Sample test case:

|  |
| --- |
| Input: Circle(radius=5)  Output: Area of Circle: 78.5  Input: Rectangle(length= 4, width= 6)  Output: Area of Rectangle: 24 |

Constraints:

* The values of width, length and radius should be positive integers.

Solution:

#include <iostream>

class Shape {

public:

virtual void calculateArea() {

std::cout << "Area calculation not defined for generic shape." << std::endl;

}

};

class Rectangle : public Shape {

private:

int length, width;

public:

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

void calculateArea() override {

std::cout << "Area of Rectangle: " << length \* width << std::endl;

}

};

class Circle : public Shape {

private:

int radius;

public:

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

void calculateArea() override {

std::cout << "Area of Circle: " << 3.14 \* radius \* radius << std::endl;

}

};

int main() {

Shape\* shape1 = new Rectangle(5, 3);

Shape\* shape2 = new Circle(4);

shape1->calculateArea();

shape2->calculateArea();

delete shape1;

delete shape2;

return 0;

}

Q-2: Imagine you are organizing a grand Multimedia Festival, where various artists and creators will be showcasing their creative works, including music and videos. To manage the multimedia playback during the festival, you decide to create a C++ program that efficiently

handles the media players for different types of content such as audio or video.

Sample test case:

|  |
| --- |
| Input: AudioPlayer("Song1.mp3")  Output: Playing audio: Song1.mp3  Input: VideoPlayer("Movie1.mp4")  Output: Playing video: Movie1.mp4 |

Solution:

#include <iostream>

#include <string>

class MediaPlayer {

protected:

std::string media;

public:

MediaPlayer(const std::string& m) : media(m) {}

virtual void play() {

std::cout << "Playing generic media." << std::endl;

}

};

class AudioPlayer : public MediaPlayer {

public:

AudioPlayer(const std::string& m) : MediaPlayer(m) {}

void play() override {

std::cout << "Playing audio: " << media << std::endl;

}

};

class VideoPlayer : public MediaPlayer {

public:

VideoPlayer(const std::string& m) : MediaPlayer(m) {}

void play() override {

std::cout << "Playing video: " << media << std::endl;

}

};

int main() {

MediaPlayer\* media1 = new AudioPlayer("Song1.mp3");

MediaPlayer\* media2 = new VideoPlayer("Movie1.mp4");

media1->play();

media2->play();

delete media1;

delete media2;

return 0;

}

Q-3: Imagine you are working on a smart home control system. The system allows users to control various devices remotely using a centralized control panel. As part of this project, you are tasked with creating a C++ program to implement the functionality of remote

controlling TVs and DVD players. To achieve this, you will define a base class called RemoteControl and two derived classes, TVRemote and DVDRemote.

Sample test case:

|  |
| --- |
| Input:  remote1 = new TVRemote()  remote1->turnOn()  remote1->turnOff()  Output:  TV: Turning ON  TV: Turning OFF  Input:  remote2 = new DVDRemote()  remote2->turnOn()  remote2->turnOff()  Output:  DVD Player: Turning ON  DVD Player: Turning OFF |
|  |

Solution:

#include <iostream>

#include <string>

class RemoteControl {

public:

virtual void turnOn() {

std::cout << "Remote control: Turning ON" << std::endl;

}

virtual void turnOff() {

std::cout << "Remote control: Turning OFF" << std::endl;

}

};

class TVRemote : public RemoteControl {

public:

void turnOn() override {

std::cout << "TV: Turning ON" << std::endl;

}

void turnOff() override {

std::cout << "TV: Turning OFF" << std::endl;

}

};

class DVDRemote : public RemoteControl {

public:

void turnOn() override {

std::cout << "DVD Player: Turning ON" << std::endl;

}

void turnOff() override {

std::cout << "DVD Player: Turning OFF" << std::endl;

}

};

int main() {

RemoteControl\* remote1 = new TVRemote();

RemoteControl\* remote2 = new DVDRemote();

remote1->turnOn();

remote1->turnOff();

remote2->turnOn();

remote2->turnOff();

delete remote1;

delete remote2;

return 0;

}

Q-4: Imagine you are working on a payroll management system for a company called "TechCorp." The company has different types of employees, including Regular Employees and Contract Employees. As part of the system, you need to create a C++ program to calculate the pay for each employee based on their employment type.

Sample test case:

|  |
| --- |
| Input:  RegularEmployee(name="John Doe",id=1001,rate=20.0,hours=160)  Output:  Regular Employee Pay: $3520  Input:  ContractEmployee(name="Jane Smith", id=2001, rate=2000, hours=0,contractRate=500)  Output:  Contract Employee Pay: $500 |

Solution:

#include <iostream>

#include <string>

class Employee {

protected:

std::string name;

int empID;

double hourlyRate;

int hoursWorked;

public:

Employee(const std::string& n, int id, double rate, int hours)

: name(n), empID(id), hourlyRate(rate), hoursWorked(hours) {}

virtual double calculatePay() {

return hourlyRate \* hoursWorked;

}

};

class RegularEmployee : public Employee {

public:

RegularEmployee(const std::string& n, int id, double rate, int hours)

: Employee(n, id, rate, hours) {}

double calculatePay() override {

double basePay = Employee::calculatePay();

return basePay + (basePay \* 0.1); // 10% bonus for regular employees

}

};

class ContractEmployee : public Employee {

private:

double contractRate;

public:

ContractEmployee(const std::string& n, int id, double rate, int hours, double conRate)

: Employee(n, id, rate, hours), contractRate(conRate) {}

double calculatePay() override {

return contractRate;

}

};

int main() {

Employee\* emp1 = new RegularEmployee("John Doe", 1001, 20.0, 160);

Employee\* emp2 = new ContractEmployee("Jane Smith", 2001, 2000, 0, 500);

std::cout << "Regular Employee Pay: $" << emp1->calculatePay() << std::endl;

std::cout << "Contract Employee Pay: $" << emp2->calculatePay() << std::endl;

delete emp1;

delete emp2;

return 0;

}

Q-5: Adventura town needs a virtual simulation of the thrilling experience of driving different vehicles on their scenic trails. Design a base class called "Vehicle," which will serve as a blueprint for all types of vehicles in Adventura. Additionally, craft two unique derived classes, "Car" and "Bike," to embody specific types of vehicles that adventurers frequently use. Each class in the program has a special function called "drive()." This function represents the exhilarating action of driving the corresponding

vehicle to experience a virtual journey through Adventura's captivating terrains.

Sample test case:

|  |
| --- |
| Input:Car()  Output:Driving car.  Input:Bike()  Output:Riding bike. |

Solution:

#include <iostream>

class Vehicle {

public:

virtual void drive() {

std::cout << "Driving generic vehicle." << std::endl;

}

};

class Car : public Vehicle {

public:

void drive() override {

std::cout << "Driving car." << std::endl;

}

};

class Bike : public Vehicle {

public:

void drive() override {

std::cout << "Riding bike." << std::endl;

}

};

int main() {

Vehicle\* vehicle1 = new Car();

Vehicle\* vehicle2 = new Bike();

vehicle1->drive();

vehicle2->drive();

delete vehicle1;

delete vehicle2;

return 0;

}

Q-6: Imagine you are developing a virtual pet simulator where users can interact with different types of animals.

To achieve this, you decide to create a C++ program that simulates the behavior of various animals.

The program will include a base class called Animal and two derived classes, Dog and Cat.

In this virtual pet world, users can select a virtual pet of their choice, either a dog or a cat, and interact with them.

Each pet will have its unique characteristics, including the sounds they make. To represent the sounds made by the animals,

each class will have a function called makeSound().

Sample test case:

|  |
| --- |
| Input:  Dog(name="Buddy")  Cat(name="Whiskers")  Output:  Buddy barks: Woof Woof!  Whiskers meows: Meow Meow! |

Solution:

#include <iostream>

#include <string>

class Animal {

protected:

std::string name;

public:

Animal(const std::string& n) : name(n) {}

virtual void makeSound() {

std::cout << "Unknown animal sound." << std::endl;

}

};

class Dog : public Animal {

public:

Dog(const std::string& n) : Animal(n) {}

void makeSound() override {

std::cout << name<<" barks: Woof Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

Cat(const std::string& n) : Animal(n) {}

void makeSound() override {

std::cout << name<<" meows: Meow Meow!" << std::endl;

}

};

int main() {

Animal\* animal1 = new Dog("Buddy");

Animal\* animal2 = new Cat("Whiskers");

animal1->makeSound();

animal2->makeSound();

delete animal1;

delete animal2;

return 0;

}

Q-7: Write a program in which we have four classes: Person, Student, Employee, and Manager. Both Student and Employee inherit virtually from Person. The Manager class is derived from both Student and Employee. Without using a virtual base class, the Manager class would face the diamond problem, as there would be two copies of the Person class in the inheritance hierarchy. By making Person a virtual base class, we resolve the diamond problem, and the manager.display() function correctly calls the Student and Employee versions of display().

Sample test case:

|  |
| --- |
| Input:  manager(1001, 3.8, 5000);  manager.display()  Output: Manager ID: 1001, GPA: 3.8, Salary: 5000 |

Solution:

#include <iostream>

using namespace std;

class Person {

public:

Person(int id) : id(id) {}

virtual void display() {

cout << "Person ID: " << id << endl;

}

int getId(){

return id;

}

private:

int id;

};

class Student : virtual public Person {

public:

Student(int id, float gpa) : Person(id), gpa(gpa) {}

void display() override {

cout << "Student ID: " << Person::getId() << ", GPA: " << gpa << endl;

}

float getGPA(){

return gpa;

}

private:

float gpa;

};

class Employee : virtual public Person {

public:

Employee(int id, float salary) : Person(id), salary(salary) {}

void display() override {

cout << "Employee ID: " << Person::getId() << ", Salary: " << salary << endl;

}

int getSalary(){

return salary;

}

private:

float salary;

};

class Manager : public Student, public Employee {

public:

Manager(int id, float gpa, float salary) : Person(id), Student(id, gpa), Employee(id, salary) {}

void display() override {

cout << "Manager ID: " << Person::getId() << ", GPA: " << Student::getGPA() << ", Salary: " << Employee::getSalary() << endl;

}

};

int main() {

Manager manager(1001, 3.8, 5000);

manager.display(); // Resolves the diamond problem using virtual base class

return 0;

}

Q-8: Develop a graphical user interface (GUI) library with a virtual base class Widget and a derived class Button. Implement a virtual function draw() in the base class and override it in the Button class to display the button on the screen.

Sample test case:

|  |
| --- |
| Input: Widget\* widget = new Button(100, 50, "Click Me!");  widget->draw();  Output: Drawing a button with label "Click Me!" at (100, 50) |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class representing a Widget

class Widget {

protected:

int x, y; // Coordinates of the widget on the screen

public:

Widget(int \_x, int \_y) : x(\_x), y(\_y) {}

// Virtual function to draw the widget

virtual void draw() const {

cout << "Drawing a generic widget at (" << x << ", " << y << ")" << endl;

}

};

// Derived class representing a Button, inheriting from Widget

class Button : public Widget {

private:

string label; // Label of the button

public:

Button(int \_x, int \_y, string \_label) : Widget(\_x, \_y), label(\_label) {}

// Override the draw function to display the button

void draw() const override {

cout << "Drawing a button with label \"" << label << "\" at (" << x << ", " << y << ")" << endl;

}

};

int main() {

Widget\* widget = new Button(100, 50, "Click Me!");

// When we call the draw method on the base class pointer, the virtual function

// will be dispatched to the appropriate version based on the actual object type.

widget->draw();

delete widget;

return 0;

}

Q-9: Create a program for managing restaurant orders. Design a virtual base class Order and derived class Invoice. Implement a virtual function calculateTotal() in the Order class, and override it in the Invoice class to calculate the total

amount to be paid including tax rate.

Sample test case:

|  |
| --- |
| Input:  Order("Pizza", 2, 12.50)  Order("Burger", 3, 8.75)  Invoice("Pasta", 4, 9.25, 0.1)  Output:  Item: Pizza, Quantity: 2, Price Per Item: $12.5, Total Amount: $25  Item: Burger, Quantity: 3, Price Per Item: $8.75, Total Amount: $26.25  Item: Pasta, Quantity: 4, Price Per Item: $9.25, Total Amount: $40.7 |

Solution:

#include <iostream>

#include <vector>

using namespace std;

// Base class representing an Order

class Order {

public:

string itemName;

int quantity;

double pricePerItem;

Order(string \_itemName, int \_quantity, double \_pricePerItem)

: itemName(\_itemName), quantity(\_quantity), pricePerItem(\_pricePerItem) {}

// Virtual function to calculate the total amount for the order

virtual double calculateTotal() const {

return quantity \* pricePerItem;

}

};

// Derived class representing an Invoice, inheriting from Order

class Invoice : public Order {

private:

double taxRate;

public:

Invoice(string \_itemName, int \_quantity, double \_pricePerItem, double \_taxRate)

: Order(\_itemName, \_quantity, \_pricePerItem), taxRate(\_taxRate) {}

// Overriding the calculateTotal() function to include tax calculation

double calculateTotal() const override {

double subTotal = Order::calculateTotal();

return subTotal + subTotal \* taxRate;

}

};

int main() {

vector<Order\*> orders;

// Adding different orders to the vector

orders.push\_back(new Order("Pizza", 2, 12.50));

orders.push\_back(new Order("Burger", 3, 8.75));

orders.push\_back(new Invoice("Pasta", 4, 9.25, 0.1)); // Tax rate of 10%

// Calculating and displaying the total amount for each order

for (Order\* order : orders) {

cout << "Item: " << order->itemName << ", Quantity: " << order->quantity

<< ", Price Per Item: $" << order->pricePerItem

<< ", Total Amount: $" << order->calculateTotal() << endl;

}

// Cleaning up the allocated memory.

for (Order\* order : orders) {

delete order;

}

return 0;

}

Q-10: Create a game development framework with a virtual base class GameObject and derived class Character. Implement a virtual function update() in the GameObject class, and override it in the Character class to update the character's position and other attributes.

Sample test case:

|  |
| --- |
| GameObject\* obj1 = new GameObject(5, 5);  obj1->update(); // Output: Updating GameObject: (5, 5)  Character\* char1 = new Character(10, 10, 100);  char1->update(); // Output: Updating Character: (11, 11), Health: 110 |

Solution:

#include <iostream>

using namespace std;

// Base class representing a GameObject

class GameObject {

protected:

int posX;

int posY;

public:

GameObject(int x, int y) : posX(x), posY(y) {}

// Virtual function to update the GameObject's position

virtual void update() {

// Base implementation, just print the position

cout << "Updating GameObject: (" << posX << ", " << posY << ")" << endl;

}

};

// Derived class representing a Character, inheriting from GameObject

class Character : public GameObject {

private:

int health;

public:

Character(int x, int y, int \_health) : GameObject(x, y), health(\_health) {}

// Override the update function to update the Character's position and health

void update() override {

// Simulate movement, increase health, or other character-specific updates

posX += 1;

posY += 1;

health += 10;

cout << "Updating Character: (" << posX << ", " << posY << "), Health: " << health << endl;

}

};

int main() {

GameObject\* obj1 = new GameObject(5, 5);

Character\* char1 = new Character(10, 10, 100);

// Calling the update function on both GameObject and Character objects

cout << "Before Update:\n";

obj1->update(); // Calls the base class implementation

char1->update(); // Calls the overridden function in Character

// Clean up the allocated memory

delete obj1;

delete char1;

return 0;

}

Q-11: Create a virtual base class Animal with pure virtual functions displayInfo() and makeSound(). Implement derived classes Lion, Elephant, and Giraffe that inherit from Animal. In a zoo management system, use these classes to display information and make sounds of different animals.

Sample test case:

|  |
| --- |
| Input:  Lion lion("Simba");  animal->displayInfo();  animal->makeSound();  Output:  Name: Simba  Species: Lion  The lion roars! |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class representing an Animal

class Animal {

protected:

string name;

string species;

public:

// Constructor for the Animal class

Animal(string \_name, string \_species) : name(\_name), species(\_species) {}

// Pure virtual function to display information about the animal

virtual void displayInfo() const = 0;

// Pure virtual function to make sound of the animal

virtual void makeSound() const = 0;

};

// Derived class representing a Lion, inheriting from Animal

class Lion : public Animal {

public:

// Constructor for the Lion class, which also calls the constructor of the base class (Animal)

Lion(string \_name) : Animal(\_name, "Lion") {}

// Implementing the displayInfo method to display information about the lion

void displayInfo() const override {

cout << "Name: " << name << "\nSpecies: " << species << endl;

}

// Implementing the makeSound method to make a lion sound

void makeSound() const override {

cout << "The lion roars!" << endl;

}

};

// Derived class representing an Elephant, inheriting from Animal

class Elephant : public Animal {

public:

// Constructor for the Elephant class, which also calls the constructor of the base class (Animal)

Elephant(string \_name) : Animal(\_name, "Elephant") {}

// Implementing the displayInfo method to display information about the elephant

void displayInfo() const override {

cout << "Name: " << name << "\nSpecies: " << species << endl;

}

// Implementing the makeSound method to make an elephant sound

void makeSound() const override {

cout << "The elephant trumpets!" << endl;

}

};

// Derived class representing a Giraffe, inheriting from Animal

class Giraffe : public Animal {

public:

// Constructor for the Giraffe class, which also calls the constructor of the base class (Animal)

Giraffe(string \_name) : Animal(\_name, "Giraffe") {}

// Implementing the displayInfo method to display information about the giraffe

void displayInfo() const override {

cout << "Name: " << name << "\nSpecies: " << species << endl;

}

// Implementing the makeSound method to make a giraffe sound

void makeSound() const override {

cout << "The giraffe bleats!" << endl;

}

};

int main() {

// Creating objects of the derived classes (Lion, Elephant, and Giraffe)

Lion lion("Simba");

Elephant elephant("Dumbo");

Giraffe giraffe("Melman");

// Displaying information and making sounds of different animals using the base class pointer (Animal\*)

Animal\* animals[] = {&lion, &elephant, &giraffe};

for (const auto animal : animals) {

animal->displayInfo();

animal->makeSound();

cout << endl;

}

return 0;

}

Q-12: Create a virtual base class Travel with pure virtual functions bookTicket() and calculateFare(). Implement derived classes Flight, Train, and Bus that inherit from Travel. In a travel booking system, use these classes to book tickets for different modes of travel and calculate the fare accordingly.

Sample test case:

|  |
| --- |
| Input:  Travel\* flight = new Flight("Flight", 2, 300.0);  flight->bookTicket();  flight->calculateFare();  Output:  Booking 2 flight ticket(s) for Flight mode.  Total fare for 2 passengers in Flight mode: $600  Input:  Travel\* train = new Train("Train", 3, 50.0);  train->bookTicket();  train->calculateFare();  Output:  Booking 3 train ticket(s) for Train mode.  Total fare for 3 passengers in Train mode: $150  Input:  Travel\* bus = new Bus("Bus", 4, 20.0);  bus->bookTicket();  bus->calculateFare();  Output:  Booking 4 bus ticket(s) for Bus mode.  Total fare for 4 passengers in Bus mode: $80 |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class Travel

class Travel {

protected:

string mode;

int passengers;

public:

Travel(string \_mode, int \_passengers) : mode(\_mode), passengers(\_passengers) {}

virtual void bookTicket() = 0;

virtual void calculateFare() = 0;

};

// Derived class Flight from Travel

class Flight : public Travel {

private:

double baseFare;

public:

Flight(string \_mode, int \_passengers, double \_baseFare) : Travel(\_mode, \_passengers), baseFare(\_baseFare) {}

// Override the base class function to book a flight ticket

void bookTicket() override {

cout << "Booking " << passengers << " flight ticket(s) for " << mode << " mode." << endl;

}

// Override the base class function to calculate flight fare

void calculateFare() override {

double totalFare = baseFare \* passengers;

cout << "Total fare for " << passengers << " passengers in " << mode << " mode: $" << totalFare << endl;

}

};

// Derived class Train from Travel

class Train : public Travel {

private:

double ticketPrice;

public:

Train(string \_mode, int \_passengers, double \_ticketPrice) : Travel(\_mode, \_passengers), ticketPrice(\_ticketPrice) {}

// Override the base class function to book a train ticket

void bookTicket() override {

cout << "Booking " << passengers << " train ticket(s) for " << mode << " mode." << endl;

}

// Override the base class function to calculate train fare

void calculateFare() override {

double totalFare = ticketPrice \* passengers;

cout << "Total fare for " << passengers << " passengers in " << mode << " mode: $" << totalFare << endl;

}

};

// Derived class Bus from Travel

class Bus : public Travel {

private:

double farePerPerson;

public:

Bus(string \_mode, int \_passengers, double \_farePerPerson) : Travel(\_mode, \_passengers), farePerPerson(\_farePerPerson) {}

// Override the base class function to book a bus ticket

void bookTicket() override {

cout << "Booking " << passengers << " bus ticket(s) for " << mode << " mode." << endl;

}

// Override the base class function to calculate bus fare

void calculateFare() override {

double totalFare = farePerPerson \* passengers;

cout << "Total fare for " << passengers << " passengers in " << mode << " mode: $" << totalFare << endl;

}

};

int main() {

// Create instances of derived classes and book tickets for different modes of travel

Travel\* flight = new Flight("Flight", 2, 300.0);

Travel\* train = new Train("Train", 3, 50.0);

Travel\* bus = new Bus("Bus", 4, 20.0);

flight->bookTicket();

flight->calculateFare();

train->bookTicket();

train->calculateFare();

bus->bookTicket();

bus->calculateFare();

// Free the memory allocated for each instance

delete flight;

delete train;

delete bus;

return 0;

}