**4.1**

A team of engineers was tasked with developing a program to calculate and manage the areas ofmultiple circles for a design project. To achieve this, they devised a solution using a structured, object-oriented approach. At the foundation of their solution was a base class that represented a genericshape, responsible for storing and managing the radius of the circle. Building upon this, a specializedclass for circles was created to extend functionality by introducing a method for calculating the areaof a circle based on its radius. Using this framework, the team designed a system to handle multiplecircles, ensuring that the process of storing, calculating, and displaying the areas was efficient andadaptable. They explored two different approaches for managing the collection of circles—onefocusing on flexibility and dynamic handling, while the other used a more static structure. Byimplementing and comparing these methods, the engineers gained insights into the benefits of usingefficient techniques for organizing and processing geometric data, enhancing their problem solving capabilities.

**CODE**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

class Shape {

protected:

double radius;

public:

Shape(double r = 0.0) : radius(r) {}

void setRadius(double r) {

radius = r;

}

double getRadius() const {

return radius;

}

};

class Circle : public Shape {

public:

Circle(double r = 0.0) : Shape(r) {}

double area() const {

return M\_PI \* radius \* radius;

}

};

void dynamicCircleHandling() {

cout << "\n--- Dynamic Circle Handling ---\n";

vector<Circle> circles;

vector<double> radii = {3.0, 5.0, 7.0};

for (double r : radii) {

circles.push\_back(Circle(r));

}

for (size\_t i = 0; i < circles.size(); ++i) {

cout << "Circle " << i + 1

<< ": Radius = " << circles[i].getRadius()

<< ", Area = " << circles[i].area()

<< endl;

}

}

int main() {

cout << "Circle Area Calculation Program\n";

staticCircleHandling();

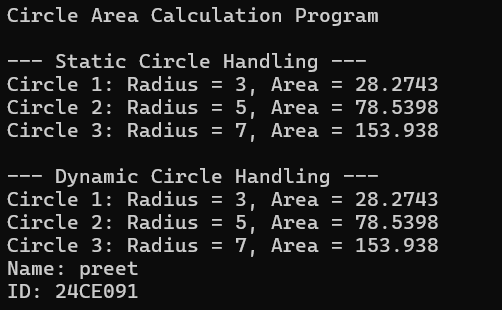
dynamicCircleHandling();

cout << "Name: preet" << endl << "ID: 24CE091" << endl;

return 0;

}

**OUTPUT**

****

**4.2**

A growing organization wanted to develop a system to manage its hierarchy and represent itsstructure in a programmatic way. To achieve this, a multilevel approach was designed, reflecting the natural progression of roles within the organization. At the foundation, a class was created to represent a person, capturing the basic details such as name and age. Building on this, an intermediate level was introduced to represent employees, adding a unique identifier for each. Finally, at thetopmost level, a class for managers was created, which included additional details such as thedepartment they oversee.

The system needed to handle the initialization of all these attributes through constructors at eachlevel, ensuring the proper propagation of information across the hierarchy. Additionally, the functionality to display details at every level was included to provide clear insights into the organization's structure. Two approaches were explored for managing multiple managers: one reliedon an efficient method for retrieval and organization based on employee identifiers, while the other used a straightforward and static method for storage.

**Code**

**#include <iostream>**

**#include <vector>**

**#include <unordered\_map>**

**using namespace std;**

**class Person {**

**protected:**

**string name;**

**int age;**

**public:**

**Person(string n, int a) : name(n), age(a) {}**

**void displayPerson() const {**

**cout << "Name: " << name << ", Age: " << age << endl;**

**}**

**};**

**class Employee : public Person {**

**protected:**

**int employeeID;**

**public:**

**Employee(string n, int a, int id) : Person(n, a), employeeID(id) {}**

**void displayEmployee() const {**

**displayPerson();**

**cout << "Employee ID: " << employeeID << endl;**

**}**

**int getID() const {**

**return employeeID;**

**}**

**};**

**class Manager : public Employee {**

**private:**

**string department;**

**public:**

**Manager(string n, int a, int id, string dept) : Employee(n, a, id), department(dept) {}**

**void displayManager() const {**

**displayEmployee();**

**cout << "Department: " << department << endl;**

**}**

**};**

**void staticManagerHandling() {**

**cout << "\n--- Static Manager Handling ---\n";**

**Manager m1("PREET", 19, 91, "HR");**

**Manager m2("PAL", 18, 92, "Finance");**

**m1.displayManager();**

**cout << endl;**

**m2.displayManager();**

**}**

**void dynamicManagerHandling() {**

**cout << "\n--- Dynamic Manager Handling ---\n";**

**vector<Manager> managerList = {**

**Manager("PREET", 19, 24-01, "CE"),**

**Manager("David", 45, 202, "Marketing")**

**};**

**unordered\_map<int, Manager\*> managerMap;**

**for (auto& m : managerList) {**

**managerMap[m.getID()] = &m;**

**}**

**int searchID = 24-91;**

**if (managerMap.count(searchID)) {**

**managerMap[searchID]->displayManager();**

**}**

**}**

**int main() {**

**cout << "Organization Hierarchy Program\n";**

**staticManagerHandling();**

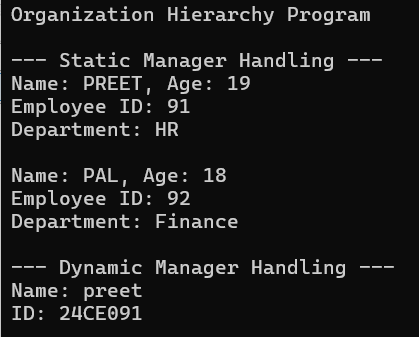
**dynamicManagerHandling();**

**cout << "Name: preet" << endl << "ID: 24CE091" << endl;**

**return 0;**

**}**

**OUTPUT**

****

**4.3**

A vehicle manufacturing company sought to create a robust system to organize and manage the details of various cars produced under its brand. To accomplish this, a hierarchical structure was conceptualized, reflecting the essential components of a vehicle. At the foundation, a class was designed to represent the type of fuel a vehicle uses. Another class was created to capture the brand name of the vehicle. These two foundational elements were then combined into a derived class specifically representing cars, integrating both fuel type and brand information.

Constructors were used at each level to ensure proper initialization of attributes, allowing seamless integration of all details. Additionally, the ability to display complete information about a car, including its fuel type and brand, was incorporated into the system. To simulate a real-world scenario such as a service queue, multiple cars were organized and processed sequentially using a structured approach. This not only streamlined the handling of cars but also provided an opportunity to compare different methods of managing the collection and processing of vehicle data.

**CODE**

#include <iostream>

#include <queue>

#include <string>

using namespace std;

class Fuel {

protected:

string fuelType;

public:

Fuel(string type) : fuelType(type) {}

};

class Brand {

protected:

string brandName;

public:

Brand(string name) : brandName(name) {}

};

class Car : public Fuel, public Brand {

public:

Car(string type, string name) : Fuel(type), Brand(name) {}

void display() {

cout << "Brand: " << brandName << ", Fuel Type: " << fuelType << endl;

}

};

int main() {

queue<Car> serviceQueue;

serviceQueue.push(Car("Petrol", "Toyota"));

serviceQueue.push(Car("Diesel", "Ford"));

serviceQueue.push(Car("Electric", "Tesla"));

while (!serviceQueue.empty()) {

Car c = serviceQueue.front();

c.display();

serviceQueue.pop();

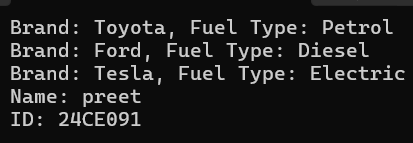
}

cout << "Name: preet" << endl << "ID: 24CE091" << endl;

return 0;

}

**OUTPUT**

****

**4.4**

In a bid to design an efficient and user-friendly banking system, a structure was proposed that mirrors the real-world operations of various account types. The foundation of the system is a base class representing a generic bank account, encapsulating essential details such as account number and balance. Building on this foundation, two specialized account types were created: a savings account, which includes an interest rate as an additional feature, and a current account, which allows an overdraft limit to accommodate specific customer needs.

To ensure proper initialization and cleanup of account objects, constructors and destructors were implemented. Essential banking operations such as deposits and withdrawals were made available for both account types, allowing users to perform and manage their transactions effectively. The system also accounted for the need to track and manage transaction history, enabling operations such as undoing the last transaction. This was achieved by simulating a mechanism to store a sequence of transactions for each account type, providing insight into different ways of managing and organizing data.

**CODE**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

class BankAccount {

protected:

string accountNumber;

double balance;

vector<string> transactionHistory;

public:

BankAccount(string accNo, double bal) : accountNumber(accNo), balance(bal) {}

virtual ~BankAccount() {}

virtual void deposit(double amount) {

balance += amount;

transactionHistory.push\_back("Deposit: " + to\_string(amount));

}

virtual void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

transactionHistory.push\_back("Withdraw: " + to\_string(amount));

}

}

virtual void undoLastTransaction() {

if (!transactionHistory.empty()) {

string last = transactionHistory.back();

size\_t pos = last.find(": ");

string type = last.substr(0, pos);

double amount = stod(last.substr(pos + 2));

if (type == "Deposit") balance -= amount;

else if (type == "Withdraw") balance += amount;

transactionHistory.pop\_back();

}

}

virtual void display() {

cout << "Account: " << accountNumber << ", Balance: " << balance << endl;

}

};

class SavingsAccount : public BankAccount {

double interestRate;

public:

SavingsAccount(string accNo, double bal, double rate) : BankAccount(accNo, bal), interestRate(rate) {}

void display() override {

BankAccount::display();

cout << "Interest Rate: " << interestRate << "%" << endl;

}

};

class CurrentAccount : public BankAccount {

double overdraftLimit;

public:

CurrentAccount(string accNo, double bal, double limit) : BankAccount(accNo, bal), overdraftLimit(limit) {}

void withdraw(double amount) override {

if (balance + overdraftLimit >= amount) {

balance -= amount;

transactionHistory.push\_back("Withdraw: " + to\_string(amount));

}

}

void display() override {

BankAccount::display();

cout << "Overdraft Limit: " << overdraftLimit << endl;

}

};

int main() {

SavingsAccount sa("S123", 1000.0, 3.5);

CurrentAccount ca("C456", 2000.0, 500.0);

sa.deposit(200.0);

sa.withdraw(150.0);

sa.undoLastTransaction();

sa.display();

ca.deposit(500.0);

ca.withdraw(2700.0);

ca.undoLastTransaction();

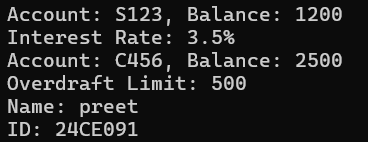
ca.display();

cout << "Name: PREET " << endl << "ID: 24CE091" << endl;

return 0;

}

**OUTPUT**

****

**4.5**

In an educational setting, an advanced grading system was conceptualized to accommodate the diverse evaluation criteria for students at different academic levels. At the heart of the system is an abstract base class that defines the grading framework. This class includes a protected member to store marks and declares a pure virtual function for computing grades, ensuring that specific grading logic is implemented by derived classes.

Two distinct derived classes were introduced to handle the unique grading needs of undergraduate and postgraduate students. Each class defines its own implementation of the grade computation method, reflecting the varying academic expectations for these groups. The system enables users to input student data, compute grades based on the respective criteria, and manage a collection of student records.

**CODE**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

class Student {

protected:

string name;

int marks;

public:

Student(string n, int m) : name(n), marks(m) {}

virtual ~Student() {}

virtual char computeGrade() = 0;

virtual void display() {

cout << "Name: " << name << ", Marks: " << marks << ", Grade: " << computeGrade() << endl;

}

};

class Undergraduate : public Student {

public:

Undergraduate(string n, int m) : Student(n, m) {}

char computeGrade() override {

if (marks >= 85) return 'A';

else if (marks >= 70) return 'B';

else if (marks >= 50) return 'C';

else return 'F';

}

};

class Postgraduate : public Student {

public:

Postgraduate(string n, int m) : Student(n, m) {}

char computeGrade() override {

if (marks >= 90) return 'A';

else if (marks >= 75) return 'B';

else if (marks >= 60) return 'C';

else return 'F';

}

};

int main() {

vector<Student\*> students;

students.push\_back(new Undergraduate("Alice", 78));

students.push\_back(new Undergraduate("Bob", 45));

students.push\_back(new Postgraduate("Charlie", 88));

students.push\_back(new Postgraduate("Diana", 92));

for (Student\* s : students) {

s->display();

delete s;

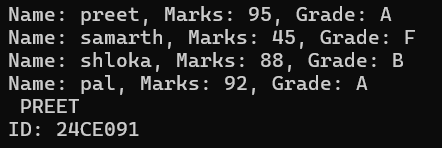
}

cout << "Name: PREET " << endl << "ID: 24CE091" << endl;

return 0;

}

**OUTPUT**

****

**Thank you**