1. Create a class that behaves like a simple list in Python. It should support adding, removing, displaying, and getting the size of the list. Explain why you used constructors and any operator overloading (if used).

#include <iostream>

using namespace std;

class list {

private:

int\* arr; // array to hold the elements

int capacity; // total slots in the array

int length; // number of used slots

public:

//set up an empty list

list() {

capacity = 5;

length = 0;

arr = new int[capacity];

}

~list() {

delete[] arr;

}

// add an item at the end

void append(int value) {

if (length == capacity) {

// make the array bigger

int ncap = capacity \* 2;

int\* narr = new int[ncap];

for (int i = 0; i < length; ++i)

narr[i] = arr[i];

delete[] arr;

arr = narr;

capacity = ncap;

}

arr[length] = value;

length++;

}

// remove item at a position

void remove\_at(int index) {

if (index < 0 || index >= length) {

cout << "invalid index!" << endl;

return;

}

for (int i = index; i < length - 1; ++i)

arr[i] = arr[i + 1];

--length;

}

// show all elements

void display() {

cout << "[ ";

for (int i = 0; i < length; ++i)

cout << arr[i] << " ";

cout << "]" << endl;

}

// return current size

int size() {

return length;

}

};

int main() {

list new\_list;

int choice, value, index;

while (true) {

cout << "\n1. Add element"

<< "\n2. Remove element"

<< "\n3. Display list"

<< "\n4. Show size"

<< "\n0. Exit"

<< "\nEnter choice: ";

cin >> choice;

if (choice == 0) break;

switch (choice) {

case 1:

cout << "Enter value: ";

cin >> value;

new\_list.append(value);

break;

case 2:

cout << "Enter index to remove: ";

cin >> index;

new\_list.remove\_at(index);

break;

case 3:

new\_list.display();

break;

case 4:

cout << "List size: " << new\_list.size() << endl;

break;

default:

cout << "Invalid option!" << endl;

}

}

return 0;

}

{

cout << "1: append 2: remove\_at 3: display 4: size 0: exit\n";

cin >> choice;

}

2. Design a class that mimics a dictionary or map functionality — where you can store key-value pairs, search for a key, and display all pairs. Explain why encapsulation is important in this implementation.

#include <iostream>

using namespace std;

class dictionary {

private:

string keys[100]; // to store keys

int values[100]; // to store values

int count; // current number of key-value pairs

public:

// set up an empty dictionary

dictionary() {

count= 0;

}

// add a key-value pair

void add(string key, int value) {

// check if key already exists

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

if (keys[i] == key) {

values[i] = value; // update existing value

cout << "Updated existing key.\n";

return;

}

}

keys[count] = key;

values[count] = value;

count++;

}

// search for a key

void search(string key) {

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

if (keys[i] == key) {

cout << "Value for '" << key << "' = " << values[i] << endl;

return;

}

}

cout << "Key not found.\n";

}

// display all pairs

void display() {

cout << "{ ";

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

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

if (i < count - 1) cout << ", ";

}

cout << " }" << endl;

}

// show how many pairs exist

int size() {

return count;

}

};

int main() {

dictionary my\_dict;

int choice, value;

string key;

while (true) {

cout << "\n1. Add/Update key-value pair"

<< "\n2. Search for key"

<< "\n3. Display all pairs"

<< "\n4. Show size"

<< "\n0. Exit"

<< "\nEnter choice: ";

cin >> choice;

if (choice == 0) break;

switch (choice) {

case 1:

cout << "Enter key (string): ";

cin >> key;

cout << "Enter value (integer): ";

cin >> value;

my\_dict.add(key, value);

break;

case 2:

cout << "Enter key to search: ";

cin >> key;

my\_dict.search(key);

break;

case 3:

my\_dict.display();

break;

case 4:

cout << "Dictionary size: " << my\_dict.size() << endl;

break;

default:

cout << "Invalid option!" << endl;

}

}

return 0;

}

3. Write a program to demonstrate single inheritance. Create a base class Shape and a derived class Rectangle. Include methods to calculate area and perimeter. Explain why inheritance is preferred over writing separate classes for each shape.

#include <iostream>

using namespace std;

class shape {

protected:

double width;

double height;

public:

shape(double w = 0, double h = 0) {

width = w;

height = h;

}

void show\_dimensions() {

cout << "Width: " << width << ", Height: " << height << endl;

}

};

class rectangle : public shape {

public:

rectangle(double w = 0, double h = 0) : shape(w, h) {}

// calculate area

double area() {

return width \* height;

}

// calculate perimeter

double perimeter() {

return 2 \* (width + height);

}

};

int main() {

double w, h;

cout << "Enter width of rectangle: ";

cin >> w;

cout << "Enter height of rectangle: ";

cin >> h;

rectangle rect(w, h);

cout << "\nRectangle Details:\n";

rect.show\_dimensions();

cout << "Area: " << rect.area() << endl;

cout << "Perimeter: " << rect.perimeter() << endl;

return 0;

}

4. Implement a program demonstrating multilevel inheritance. For example: Person→ Employee → Manager. Show how data and functions are passed through the inheritance chain. Explain why access specifiers matter here.

#include <iostream>

using namespace std;

class person {

protected:

string name;

int age;

public:

person(string n = "", int a = 0) {

name = n;

age = a;

}

void show\_person() {

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

}

};

class employee : public person {

protected:

int emp\_id;

double salary;

public:

employee(string n = "", int a = 0, int id = 0, double sal = 0.0)

: person(n, a) {

emp\_id = id;

salary = sal;

}

void show\_employee() {

show\_person();

cout << "Employee ID: " << emp\_id << ", Salary: " << salary << endl;

}

};

class manager : public employee {

private:

string department;

public:

// constructor

manager(string n = "", int a = 0, int id = 0, double sal = 0.0, string dep = "")

: employee(n, a, id, sal) {

department = dep;

}

void show\_manager() {

show\_employee();

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

}

};

int main() {

string name, dept;

int age, id;

double salary;

cout << "Enter manager's name: ";

cin >> name;

cout << "Enter age: ";

cin >> age;

cout << "Enter employee ID: ";

cin >> id;

cout << "Enter salary: ";

cin >> salary;

cout << "Enter department: ";

cin >> dept;

manager m1(name, age, id, salary, dept);

cout << "\nManager Details:\n";

m1.show\_manager();

return 0;

}

5. Create a program that demonstrates multiple inheritance. Example: A class SportsPerson inherits from both Person and Athlete. Explain what ambiguity arises and how you resolved it using scope resolution or virtual base classes.

#include <iostream>

using namespace std;

class person {

protected:

string name;

int age;

public:

person(string n = "", int a = 0) {

name = n;

age = a;

}

void show\_person() {

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

}

};

class athlete {

protected:

string sport;

int medals;

public:

athlete(string s = "", int m = 0) {

sport = s;

medals = m;

}

void show\_athlete() {

cout << "Sport: " << sport << ", Medals: " << medals << endl;

}

};

class sportsperson : public person, public athlete {

public:

sportsperson(string n, int a, string s, int m)

: person(n, a), athlete(s, m) {}

void show\_sportsperson() {

show\_person();

show\_athlete();

}

};

int main() {

string name, sport;

int age, medals;

cout << "Enter name: ";

cin >> name;

cout << "Enter age: ";

cin >> age;

cout << "Enter sport: ";

cin >> sport;

cout << "Enter number of medals: ";

cin >> medals;

sportsperson sp(name, age, sport, medals);

cout << "\nSportsPerson Details:\n";

sp.show\_sportsperson();

return 0;

}

6. Write a program where two different class objects share data using a friend function. For example, a class Student and a class Sports share marks and scores. Explain why the friend function was needed instead of a member function.

#include <iostream>

using namespace std;

// must declare the class

class sports;

class student {

private:

string name;

int marks;

public:

student(string n = "", int m = 0) {

name = n;

marks = m;

}

void show\_student() {

cout << "Student Name: " << name << ", Marks: " << marks << endl;

}

friend void show\_total\_score(student s, sports sp);

};

class sports {

private:

int score;

public:

sports(int sc = 0) {

score = sc;

}

void show\_sports() {

cout << "Sports Score: " << score << endl;

}

friend void show\_total\_score(student s, sports sp);

};

void show\_total\_score(student s, sports sp) {

cout << "\n--- Combined Scores ---\n";

cout << "Student Name: " << s.name << endl;

cout << "Marks: " << s.marks << ", Sports Score: " << sp.score << endl;

cout << "Total Score: " << s.marks + sp.score << endl;

}

int main() {

string name;

int marks, score;

cout << "Enter student name: ";

cin >> name;

cout << "Enter academic marks: ";

cin >> marks;

cout << "Enter sports score: ";

cin >> score;

student s1(name, marks);

sports sp1(score);

s1.show\_student();

sp1.show\_sports();

show\_total\_score(s1, sp1);

return 0;

}

7. Implement all sorting algorithm (Bubble Sort, Insertion Sort, or Selection Sort) using OOP concepts. Use a class and methods for sorting, displaying, and inputting data. Explain why object-oriented design improves modularity here.

#include <iostream>

using namespace std;

class sorter {

private:

int arr[100]; // array to hold elements

int len; // number of elements

public:

sorter() {

len = 0;

}

void input() {

cout << "Enter number of elements: ";

cin >> len;

cout << "Enter " <<len << " elements:\n";

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

cin >> arr[i];

}

}

void display() {

cout << "[ ";

for (int i = 0; i < len; i++)

cout << arr[i] << " ";

cout << "]" << endl;

}

// bubble sort algorithm

void bubble\_sort() {

for (int i = 0; i < len - 1; i++) {

for (int j = 0; j < len - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

cout << "Array sorted using Bubble Sort.\n";

}

};

int main() {

sorter s;

int choice;

while (true) {

cout << "\n1. Input elements"

<< "\n2. Display elements"

<< "\n3. Bubble sort"

<< "\n0. Exit"

<< "\nEnter choice: ";

cin >> choice;

if (choice == 0) break;

switch (choice) {

case 1:

s.input();

break;

case 2:

s.display();

break;

case 3:

s.bubble\_sort();

break;

default:

cout << "Invalid option!\n";

}

}

return 0;

}

8. Create a program to demonstrate function overloading and overriding. Show how compile-time and run-time polymorphism differ. Explain when each is useful in real-world applications.

#include <iostream>

using namespace std;

class base {

public:

//overriding

virtual void greet() {

cout << "Hello from base class!\n";

}

// overloading same name, different parameters

void display(int x) {

cout << "Integer: " << x << endl;

}

void display(string s) {

cout << "String: " << s << endl;

}

};

class derived : public base {

public:

// overriding base class greet

void greet() override {

cout << "Hello from derived class!\n";

}

};

int main() {

base b;

derived d;

int choice;

while (true) {

cout << "\n1. Base greet"

<< "\n2. Derived greet"

<< "\n3. Display integer"

<< "\n4. Display string"

<< "\n0. Exit"

<< "\nEnter choice: ";

cin >> choice;

if (choice == 0) break;

switch (choice) {

case 1:

b.greet();

break;

case 2:

d.greet();

break;

case 3:

{

int num;

cout << "Enter integer: ";

cin >> num;

b.display(num); // overloading

break;

}

case 4:

{

string text;

cout << "Enter string: ";

cin >> text;

b.display(text); // overloading

break;

}

default:

cout << "Invalid option!\n";

}

}

return 0;

}

9. Write a class BankAccount with features like deposit(), withdraw(), and display(). Use constructor initialization and show how encapsulation prevents invalid access. Explain why data hiding is essential in banking systems.

#include <iostream>

using namespace std;

class bankaccount {

private:

string account\_holder;

int account\_number;

double balance;

public:

bankaccount(string name = "", int acc\_num = 0, double bal = 0.0) {

account\_holder = name;

account\_number = acc\_num;

balance = bal;

}

void deposit(double amount) {

if (amount > 0) {

balance += amount;

cout << "Deposited: " << amount << endl;

} else {

cout << "Invalid deposit amount!\n";

}

}

void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

cout << "Withdrawn: " << amount << endl;

} else {

cout << "Invalid withdrawal amount or insufficient balance!\n";

}

}

void display() {

cout << "Account Holder: " << account\_holder << endl;

cout << "Account Number: " << account\_number << endl;

cout << "Balance: " << balance << endl;

}

};

int main() {

string name;

int acc\_num;

double bal;

int choice;

double amount;

cout << "Enter account holder name: ";

cin >> name;

cout << "Enter account number: ";

cin >> acc\_num;

cout << "Enter initial balance: ";

cin >> bal;

bankaccount acc(name, acc\_num, bal);

while (true) {

cout << "\n1. Deposit"

<< "\n2. Withdraw"

<< "\n3. Display account"

<< "\n0. Exit"

<< "\nEnter choice: ";

cin >> choice;

if (choice == 0) break;

switch (choice) {

case 1:

cout << "Enter deposit amount: ";

cin >> amount;

acc.deposit(amount);

break;

case 2:

cout << "Enter withdrawal amount: ";

cin >> amount;

acc.withdraw(amount);

break;

case 3:

acc.display();

break;

default:

cout << "Invalid option!\n";

}

}

return 0;

}

10. Design a simple example that demonstrates abstract classes and pure virtual functions. Create a base class Shape with a pure virtual function area() and derive Circle and Square classes. Explain why abstract classes are useful in large-scale software design.

#include <iostream>

using namespace std;

// abstract base class

class shape {

public:

// pure virtual function

virtual double area() = 0;

virtual void show\_area() {

cout << "Area: " << area() << endl;

}

};

class circle : public shape {

private:

double radius;

public:

circle(double r = 0) {

radius = r;

}

double area() override {

return 3.14159 \* radius \* radius;

}

};

class square : public shape {

private:

double side;

public:

square(double s = 0) {

side = s;

}

double area() override {

return side \* side;

}

};

int main() {

double r, s;

int choice;

while (true) {

cout << "\n1. Circle Area"

<< "\n2. Square Area"

<< "\n0. Exit"

<< "\nEnter choice: ";

cin >> choice;

if (choice == 0) break;

switch (choice) {

case 1:

cout << "Enter radius of circle: ";

cin >> r;

{

circle c(r);

c.show\_area();

}

break;

case 2:

cout << "Enter side of square: ";

cin >> s;

{

square sq(s);

sq.show\_area();

}

break;

default:

cout << "Invalid option!\n";

}

}

return 0;

}