**Problem .1 :List Operations**

**Description:**

**Write a program that uses the std::list container to manage a collection of integers. Your program should perform the following operations:**

**Insert elements at the front and back of the list.**

**Remove elements from the front and back of the list.**

**Sort the list in ascending and descending order.**

**Reverse the list.**

**Display the elements of the list.**

#include <iostream>

#include <list>

#include <algorithm> // For std::sort

void displayList(const std::list<int>& lst) {

for (int num : lst) {

std::cout << num << " ";

}

std::cout << std::endl;

}

int main() {

std::list<int> myList;

myList.push\_back(10);

myList.push\_back(20);

myList.push\_front(5);

myList.push\_front(1);

std::cout << "List after inserting : ";

displayList(myList);

myList.pop\_front();

myList.pop\_back();

std::cout << "List after removing : ";

displayList(myList);

myList.sort();

std::cout << "List sorted in ascending order: ";

displayList(myList);

myList.sort(std::greater<int>());

std::cout << "List sorted in descending order: ";

displayList(myList);

myList.reverse();

std::cout << "List after reversing: ";

displayList(myList);

return 0;

}

OUTPUT:

List after inserting : 1 5 10 20

List after removing : 5 10

List sorted in ascending order: 5 10

List sorted in descending order: 10 5

List after reversing: 5 10

**Problem 2: Vector Manipulation**

**Description:**

**Create a program that uses the std::vector container to store a collection of floating-point numbers. The program should:**

**Add elements to the vector.**

**Remove elements from a specified position.**

**Find the maximum and minimum elements in the vector.**

**Calculate the average of the elements.**

**Display the elements of the vector.**

#include <iostream>

#include <vector>

#include <algorithm>

#include <numeric> // For std::accumulate

// Function to display the elements of the vector

void displayVector(const std::vector<float>& vec) {

for (float num : vec) {

std::cout << num << " ";

}

std::cout << std::endl;

}

// Function to find the maximum element in the vector

float findMax(const std::vector<float>& vec) {

return \*std::max\_element(vec.begin(), vec.end());

}

// Function to find the minimum element in the vector

float findMin(const std::vector<float>& vec) {

return \*std::min\_element(vec.begin(), vec.end());

}

// Function to calculate the average of the elements in the vector

float calculateAverage(const std::vector<float>& vec) {

if (vec.empty()) {

return 0.0;

}

float sum = std::accumulate(vec.begin(), vec.end(), 0.0f);

return sum / vec.size();

}

int main() {

std::vector<float> myVector;

// Add elements to the vector

myVector.push\_back(1.2f);

myVector.push\_back(2.3f);

myVector.push\_back(3.4f);

myVector.push\_back(4.5f);

myVector.push\_back(5.6f);

std::cout << "Vector after adding elements: ";

displayVector(myVector);

// Remove element from a specified position (e.g., position 2)

if (myVector.size() > 2) {

myVector.erase(myVector.begin() + 2);

}

std::cout << "Vector after removing element at position 2: ";

displayVector(myVector);

// Find the maximum element in the vector

float maxElement = findMax(myVector);

std::cout << "Maximum element in the vector: " << maxElement << std::endl;

// Find the minimum element in the vector

float minElement = findMin(myVector);

std::cout << "Minimum element in the vector: " << minElement << std::endl;

// Calculate the average of the elements in the vector

float average = calculateAverage(myVector);

std::cout << "Average of the elements in the vector: " << average << std::endl;

return 0;

}

OUTPUT:

Vector after adding elements: 1.2 2.3 3.4 4.5 5.6

Vector after removing element at position 2: 1.2 2.3 4.5 5.6

Maximum element in the vector: 5.6

Minimum element in the vector: 1.2

Average of the elements in the vector: 3.4

**Problem 3: Queue Simulation**

**Description:**

**Implement a program using the std::queue container to simulate a ticketing system. The program should:**

**Add customers to the queue.**

**Serve customers (remove from front of the queue).**

**Display the current queue.**

**Display the number of customers served.**

#include <iostream>

#include <queue>

#include <string>

// Function to display the current queue

void displayQueue(const std::queue<std::string>& q) {

std::queue<std::string> copy = q;

while (!copy.empty()) {

std::cout << copy.front() << " ";

copy.pop();

}

std::cout << std::endl;

}

int main() {

std::queue<std::string> ticketQueue;

int customersServed = 0;

// Add customers to the queue

ticketQueue.push("Customer 1");

ticketQueue.push("Customer 2");

ticketQueue.push("Customer 3");

ticketQueue.push("Customer 4");

std::cout << "Current queue: ";

displayQueue(ticketQueue);

// Serve customers (remove from front of the queue)

while (!ticketQueue.empty()) {

std::cout << "Serving: " << ticketQueue.front() << std::endl;

ticketQueue.pop();

customersServed++;

std::cout << "Current queue: ";

displayQueue(ticketQueue);

}

// Display the number of customers served

std::cout << "Number of customers served: " << customersServed << std::endl;

return 0;

}

OUTPUT:

Current queue: Customer 1 Customer 2 Customer 3 Customer 4

Serving: Customer 1

Current queue: Customer 2 Customer 3 Customer 4

Serving: Customer 2

Current queue: Customer 3 Customer 4

Serving: Customer 3

Current queue: Customer 4

Serving: Customer 4

Current queue:

Number of customers served: 4

**Problem 4: Stack Operations**

**Description:**

**Write a program using the std::stack container to evaluate a postfix expression. The program should:**

**Read a postfix expression.**

**Use a stack to evaluate the expression.**

**Display the result of the evaluation.**

// C++ Program to illustrate how we can use the stack data

// structure to evaluate the value of a postfix expression

#include <iostream>

#include <stack>

#include <string>

using namespace std;

// Function to perform an operation based on the operator

// and return the result

int performOperation(int operand1, int operand2, char operation) {

switch (operation) {

case '+':

return operand1 + operand2;

case '-':

return operand1 - operand2;

case '\*':

return operand1 \* operand2;

case '/':

return operand1 / operand2;

default:

return 0;

}

}

// Function to evaluate the postfix expression

int evaluatePostfixExpression(const string& expression) {

stack<int> stack;

for (char c : expression) {

if (isdigit(c)) {

// Convert char digit to int and push onto the stack

stack.push(c - '0');

} else {

// Pop the top two elements for the operation

int operand2 = stack.top();

stack.pop();

int operand1 = stack.top();

stack.pop();

// Perform operation and push the result back onto the stack

int result = performOperation(operand1, operand2, c);

stack.push(result);

}

}

// The final result should be the only item left in the stack

return stack.top();

}

int main() {

string expression;

// Prompt the user to enter a postfix expression

cout << "Enter a postfix expression: ";

cin >> expression;

// Evaluate the postfix expression

int result = evaluatePostfixExpression(expression);

cout << "Result of Postfix Expression \"" << expression << "\" is: " << result << endl;

return 0;

}

OUTPUT:

Enter a postfix expression: 73\*4+

Result of Postfix Expression "73\*4+" is: 25

**Classes:**

**Shape: Base class representing a generic shape.**

**Rectangle: Derived class representing a rectangle with length and width.**

**Circle: Derived class representing a circle with radius.**

#include <iostream>

#include <cmath>

class Shape {

public:

virtual float area() const = 0;

virtual float perimeter() const = 0;

virtual void print() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float perimeter() const override {

return 2 \* (length + width);

}

void print() const override {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float perimeter() const override {

return 2 \* M\_PI \* radius;

}

void print() const override {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << perimeter() << std::endl;

}

};

int main() {

Rectangle rect(5, 3);

Circle circle(4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: Length = 5, Width = 3, Area = 15, Perimeter = 16

Circle: Radius = 4, Area = 50.2655, Circumference = 25.1327

**Constructors and Destructors:**

**Define a default constructor for Shape to initialize common properties.**

**Overload constructors for Rectangle and Circle to take specific dimensions as input during object creation.**

**Implement destructors for all classes to handle memory cleanup (if applicable).**

#include <iostream>

#include <cmath>

class Shape {

protected:

std::string name;

public:

Shape() : name("Shape") {}

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

virtual float area() const = 0;

virtual float perimeter() const = 0;

virtual void print() const = 0;

virtual ~Shape() {

std::cout << "Shape destructor called for " << name << std::endl;

}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

Rectangle() : length(0), width(0) {}

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

float area() const override {

return length \* width;

}

float perimeter() const override {

return 2 \* (length + width);

}

void print() const override {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

~Rectangle() {

std::cout << "Rectangle destructor called" << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

Circle() : radius(0) {}

Circle(float r) : radius(r), Shape("Circle") {}

float area() const override {

return M\_PI \* radius \* radius;

}

float perimeter() const override {

return 2 \* M\_PI \* radius;

}

void print() const override {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << perimeter() << std::endl;

}

~Circle() {

std::cout << "Circle destructor called" << std::endl;

}

};

int main() {

Rectangle rect(5, 3);

Circle circle(4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: Length = 5, Width = 3, Area = 15, Perimeter = 16

Circle: Radius = 4, Area = 50.2655, Circumference = 25.1327

Circle destructor called

Shape destructor called for Circle

Rectangle destructor called

Shape destructor called for Rectangle

**Overriding:**

**Override the area() function in Rectangle and Circle to calculate their respective areas using appropriate formulas. The base class Shape can have a pure virtual area() function to enforce implementation in derived classes.**

#include <iostream>

#include <cmath>

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float perimeter() const {

return 2 \* (length + width);

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float circumference() const {

return 2 \* M\_PI \* radius;

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << circumference() << std::endl;

}

};

int main() {

Rectangle rect(5, 3);

Circle circle(4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: Length = 5, Width = 3, Area = 15, Perimeter = 16

Circle: Radius = 4, Area = 50.2655, Circumference = 25.1327

**Operator Overloading:**

**Overload the == operator for Shape to compare shapes based on a chosen criterion (e.g., area for simplicity).**

**Consider overloading other operators (like +) for specific shapes if applicable (e.g., combining rectangles)**

#include <iostream>

#include <cmath>

class Shape {

public:

virtual float area() const = 0;

bool operator==(const Shape& other) const {

return this->area() == other.area();

}

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float perimeter() const {

return 2 \* (length + width);

}

bool operator==(const Rectangle& other) const {

return this->area() == other.area();

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float circumference() const {

return 2 \* M\_PI \* radius;

}

bool operator==(const Circle& other) const {

return this->area() == other.area();

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << circumference() << std::endl;

}

};

int main() {

Rectangle rect1(5, 3);

Rectangle rect2(4, 4);

Circle circle1(4);

Circle circle2(5);

std::cout << "Comparison using operator == :" << std::endl;

std::cout << "Rect1 == Rect2: " << (rect1 == rect2) << std::endl;

std::cout << "Circle1 == Circle2: " << (circle1 == circle2) << std::endl;

return 0;

}

OUTPUT:

Comparison using operator == :

Rect1 == Rect2: 0

Circle1 == Circle2: 0

**Friend Function:**

**Define a friend function totalArea outside the class hierarchy that takes an array of Shape pointers and calculates the total area of all shapes. This function needs access to private member variables of Shape and its derived classes.**

#include <iostream>

#include <cmath>

class Shape;

class Rectangle;

class Circle;

float totalArea(const Shape\* shapes[], int size);

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float getLength() const {

return length;

}

float getWidth() const {

return width;

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << std::endl;

}

friend float totalArea(const Shape\* shapes[], int size);

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float getRadius() const {

return radius;

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << std::endl;

}

friend float totalArea(const Shape\* shapes[], int size);

};

float totalArea(const Shape\* shapes[], int size) {

float total = 0.0;

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

total += shapes[i]->area();

}

return total;

}

int main() {

Rectangle rect(5, 3);

Circle circle(4);

const Shape\* shapes[] = { &rect, &circle };

float total = totalArea(shapes, 2);

std::cout << "Total area of all shapes: " << total << std::endl;

return 0;

}

OUTPUT:

Total area of all shapes: 65.2655

**Template (Optional)**

**(Optional) Create a template class Point to represent a point in 2D space with x and y coordinates. Use this template class within the Shape hierarchy if needed.**

**Implementation:**

#include <iostream>

template<typename T>

class Point {

private:

T x;

T y;

public:

Point(T x = 0, T y = 0) : x(x), y(y) {}

T getX() const { return x; }

T getY() const { return y; }

void setX(T newX) { x = newX; }

void setY(T newY) { y = newY; }

void print() const {

std::cout << "(" << x << ", " << y << ")";

}

};

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

Point<float> topLeft;

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

void print() const {

std::cout << "Rectangle: TopLeft = ";

topLeft.print();

std::cout << ", Length = " << length << ", Width = " << width

<< ", Area = " << area() << std::endl;

}

};

class Circle : public Shape {

private:

Point<float> center;

float radius;

public:

Circle(float x, float y, float r) : center(x, y), radius(r) {}

float area() const override {

return 3.14159f \* radius \* radius;

}

void print() const {

std::cout << "Circle: Center = ";

center.print();

std::cout << ", Radius = " << radius

<< ", Area = " << area() << std::endl;

}

};

int main() {

Rectangle rect(1, 2, 5, 3);

Circle circle(0, 0, 4);

rect.print();

circle.print();

return 0;

}

OUTPUT:  
Rectangle: TopLeft = (1, 2), Length = 5, Width = 3, Area = 15

Circle: Center = (0, 0), Radius = 4, Area = 50.2654

**Implementation:**

**Design the Shape class with appropriate member variables and functions, including a pure virtual area() function.**

**Implement derived classes Rectangle and Circle with constructors, destructors, overridden area() functions, and potentially overloaded operators.**

**Define a friend function totalArea that takes an array of Shape pointers and calculates the total area.**

**(Optional) Implement a template class Point for representing points**

#include <iostream>

#include <cmath> // For M\_PI

// Template class representing a point in 2D space

template<typename T>

class Point {

private:

T x;

T y;

public:

// Constructor with default values

Point(T x = 0, T y = 0) : x(x), y(y) {}

// Getter for x coordinate

T getX() const { return x; }

// Getter for y coordinate

T getY() const { return y; }

// Setter for x coordinate

void setX(T newX) { x = newX; }

// Setter for y coordinate

void setY(T newY) { y = newY; }

// Function to print the coordinates

void print() const {

std::cout << "(" << x << ", " << y << ")";

}

};

// Base class representing a generic shape

class Shape {

public:

// Pure virtual function for area calculation

virtual float area() const = 0;

// Virtual destructor

virtual ~Shape() {}

};

// Derived class representing a rectangle with length and width

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

// Override area calculation for rectangle

float area() const override {

return length \* width;

}

// Optional: Destructor

~Rectangle() {}

// Optional: Overload equality operator for comparing rectangles by area

bool operator==(const Rectangle& other) const {

return this->area() == other.area();

}

// Optional: Overload addition operator to combine rectangles (example)

Rectangle operator+(const Rectangle& other) const {

float newLength = this->length + other.length;

float newWidth = this->width + other.width;

return Rectangle(newLength, newWidth);

}

// Function to print details of the rectangle

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << std::endl;

}

};

// Derived class representing a circle with radius

class Circle : public Shape {

private:

float radius;

public:

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

// Override area calculation for circle

float area() const override {

return M\_PI \* radius \* radius;

}

// Optional: Destructor

~Circle() {}

// Optional: Overload equality operator for comparing circles by area

bool operator==(const Circle& other) const {

return this->area() == other.area();

}

// Function to print details of the circle

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << std::endl;

}

};

// Friend function declaration for totalArea

float totalArea(const Shape\* shapes[], int size);

// Friend function definition to calculate total area of shapes

float totalArea(const Shape\* shapes[], int size) {

float total = 0.0;

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

total += shapes[i]->area();

}

return total;

}

int main() {

// Example usage of the Shape classes with Point template

Rectangle rect(5, 3);

Circle circle(4);

// Array of Shape pointers

const Shape\* shapes[] = { &rect, &circle };

// Calculate total area using friend function

float total = totalArea(shapes, 2);

std::cout << "Total area of all shapes: " << total << std::endl;

// Optional: Demonstrating operator overloading (uncomment to use)

// Rectangle rect1(3, 2);

// Rectangle rect2(4, 5);

// Rectangle rectSum = rect1 + rect2;

// std::cout << "Sum of rectangles: ";

// rectSum.print();

return 0;

}

OUTPUT:

Total area of all shapes: 65.2655