Smart Pointers: Define a smart pointer class MySmartPtr that overloads the dereference operator (\*) and arrow operator (->) for memory management and safe access to the pointed-to object.

#include <iostream>

template <typename T>

class MySmartPtr {

private:

T\* ptr;

public:

MySmartPtr(T\* p = nullptr) : ptr(p) {}

~MySmartPtr() {

delete ptr;

}

T& operator\*() {

return \*ptr;

}

T\* operator->() {

return ptr;

}

// Disable copy constructor and copy assignment

MySmartPtr(const MySmartPtr&) = delete;

MySmartPtr& operator=(const MySmartPtr&) = delete;

// Move constructor and move assignment

MySmartPtr(MySmartPtr&& other) : ptr(other.ptr) {

other.ptr = nullptr;

}

MySmartPtr& operator=(MySmartPtr&& other) {

if (this != &other) {

delete ptr;

ptr = other.ptr;

other.ptr = nullptr;

}

return \*this;

}

};

class Test {

public:

void display() const {

std::cout << "Test class display function" << std::endl;

}

};

int main() {

MySmartPtr<Test> ptr(new Test());

ptr->display(); // Output: Test class display function

(\*ptr).display(); // Output: Test class display function

return 0;

}

Template Class (Vector): Implement a template class Vector that can store elements of any data type and overload operators (+, -, []) to work with vectors of different types.

#include <iostream>

#include <vector>

template <typename T>

class Vector {

private:

std::vector<T> data;

public:

Vector() {}

void push\_back(T value) {

data.push\_back(value);

}

Vector operator+(const Vector& other) const {

if (data.size() != other.data.size()) {

throw std::invalid\_argument("Vectors must have the same size for addition");

}

Vector result;

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

result.push\_back(data[i] + other.data[i]);

}

return result;

}

Vector operator-(const Vector& other) const {

if (data.size() != other.data.size()) {

throw std::invalid\_argument("Vectors must have the same size for subtraction");

}

Vector result;

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

result.push\_back(data[i] - other.data[i]);

}

return result;

}

T& operator[](size\_t index) {

if (index >= data.size()) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

const T& operator[](size\_t index) const {

if (index >= data.size()) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

void print() const {

for (const auto& elem : data) {

std::cout << elem << " ";

}

std::cout << std::endl;

}

};

int main() {

Vector<int> v1;

v1.push\_back(1);

v1.push\_back(2);

v1.push\_back(3);

Vector<int> v2;

v2.push\_back(4);

v2.push\_back(5);

v2.push\_back(6);

Vector<int> v3 = v1 + v2;

v3.print(); // Output: 5 7 9

Vector<int> v4 = v1 - v2;

v4.print(); // Output: -3 -3 -3

std::cout << v1[1] << std::endl; // Output: 2

return 0;

}

Matrix Operations (Challenge): Create a class Matrix to store a 2D array and overload arithmetic operators (+, -, \*) for matrix addition, subtraction, and multiplication (considering matrix dimensions).

#include <iostream>

#include <vector>

#include <stdexcept>

class Matrix {

private:

std::vector<std::vector<int>> data;

int rows;

int cols;

public:

Matrix(int r, int c) : rows(r), cols(c), data(r, std::vector<int>(c, 0)) {}

void setElement(int r, int c, int value) {

if (r >= 0 && r < rows && c >= 0 && c < cols) {

data[r][c] = value;

}

}

Matrix operator+(const Matrix& other) const {

if (rows != other.rows || cols != other.cols) {

throw std::invalid\_argument("Matrices must have the same dimensions for addition");

}

Matrix result(rows, cols);

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

for (int j = 0; j < cols; ++j) {

result.data[i][j] = data[i][j] + other.data[i][j];

}

}

return result;

}

Matrix operator-(const Matrix& other) const {

if (rows != other.rows || cols != other.cols) {

throw std::invalid\_argument("Matrices must have the same dimensions for subtraction");

}

Matrix result(rows, cols);

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

for (int j = 0; j < cols; ++j) {

result.data[i][j] = data[i][j] - other.data[i][j];

}

}

return result;

}

Matrix operator\*(const Matrix& other) const {

if (cols != other.rows) {

throw std::invalid\_argument("Matrices must have compatible dimensions for multiplication");

}

Matrix result(rows, other.cols);

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

for (int j = 0; j < other.cols; ++j) {

for (int k = 0; k < cols; ++k) {

result.data[i][j] += data[i][k] \* other.data[k][j];

}

}

}

return result;

}

void print() const {

for (const auto& row : data) {

for (const auto& elem : row) {

std::cout << elem << " ";

}

std::cout << std::endl;

}

}

};

int main() {

Matrix m1(2, 3);

m1.setElement(0, 0, 1);

m1.setElement(0, 1, 2);

m1.setElement(0, 2, 3);

m1.setElement(1, 0, 4);

m1.setElement(1, 1, 5);

m1.setElement(1, 2, 6);

Matrix m2(2, 3);

m2.setElement(0, 0, 6);

m2.setElement(0, 1, 5);

m2.setElement(0, 2, 4);

m2.setElement(1, 0, 3);

m2.setElement(1, 1, 2);

m2.setElement(1, 2, 1);

Matrix m3 = m1 + m2;

m3.print();

// Output:

// 7 7 7

// 7 7 7

Matrix m4(3, 2);

m4.setElement(0, 0, 1);

m4.setElement(0, 1, 2);

m4.setElement(1, 0, 3);

m4.setElement(1, 1, 4);

m4.setElement(2, 0, 5);

m4.setElement(2, 1, 6);

Matrix m5 = m1 \* m4;

m5.print();

// Output:

// 22 28

// 49 64

return 0;

}

Custom Container: Design a class CustomList that behaves like a list but overloads the subscript operator ([]) to perform boundary checking and prevent out-of-bounds access.

#include <iostream>

#include <vector>

#include <stdexcept>

template <typename T>

class CustomList {

private:

std::vector<T> data;

public:

void push\_back(const T& value) {

data.push\_back(value);

}

T& operator[](size\_t index) {

if (index >= data.size()) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

const T& operator[](size\_t index) const {

if (index >= data.size()) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

size\_t size() const {

return data.size();

}

void print() const {

for (const auto& elem : data) {

std::cout << elem << " ";

}

std::cout << std::endl;

}

};

int main() {

CustomList<int> list;

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

list.print(); // Output: 1 2 3

try {

std::cout << list[2] << std::endl; // Output: 3

std::cout << list[3] << std::endl; // Throws out\_of\_range exception

} catch (const std::out\_of\_range& e) {

std::cerr << e.what() << std::endl; // Output: Index out of range

}

return 0;

}

Inventory Management: Implement a class Item with properties like name, price, and quantity. Overload the << operator for easy printing of item details to the console

#include <iostream>

#include <string>

class Item {

private:

std::string name;

double price;

int quantity;

public:

Item(std::string n, double p, int q) : name(n), price(p), quantity(q) {}

friend std::ostream& operator<<(std::ostream& os, const Item& item) {

os << "Name: " << item.name << ", Price: " << item.price << ", Quantity: " << item.quantity;

return os;

}

};

int main() {

Item item("Laptop", 999.99, 10);

std::cout << item << std::endl; // Output: Name: Laptop, Price: 999.99, Quantity: 10

return 0;

}

Shape Hierarchy: Create a base class Shape with an abstract method getArea. Derive classes like Circle, Rectangle, and Square from Shape and implement the getArea method in each derived class.

#include <iostream>

#include <cmath>

class Shape {

public:

virtual double getArea() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

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

double getArea() const override {

return M\_PI \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double length, width;

public:

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

double getArea() const override {

return length \* width;

}

};

class Square : public Shape {

private:

double side;

public:

Square(double s) : side(s) {}

double getArea() const override {

return side \* side;

}

};

int main() {

Circle c(5);

Rectangle r(4, 6);

Square s(3);

std::cout << "Circle Area: " << c.getArea() << std::endl;

std::cout << "Rectangle Area: " << r.getArea() << std::endl;

std::cout << "Square Area: " << s.getArea() << std::endl;

return 0;

}

Math Functions: Design overloaded functions factorial and power that can handle integer and floating-point input for calculating factorials and raising a number to a power.

#include <iostream>

#include <stdex>

#include <cmath>

int factorial(int n) {

if (n < 0) {

throw std::invalid\_argument("Factorial is not defined for negative numbers.");

}

if (n == 0 || n == 1) {

return 1;

}

int result = 1;

for (int i = 2; i <= n; ++i) {

result \*= i;

}

return result;

}

double factorial(double n) {

throw std::invalid\_argument("Factorial is not defined for floating-point numbers.");

}

int power(int base, int exponent) {

return std::pow(base, exponent);

}

double power(double base, double exponent) {

return std::pow(base, exponent);

}

int main() {

try {

int n = 5;

std::cout << "Factorial of " << n << " is " << factorial(n) << std::endl;

int base = 2;

int exponent = 3;

std::cout << base << " raised to the power of " << exponent << " is " << power(base, exponent) << std::endl;

double base\_f = 2.5;

double exponent\_f = 3.0;

std::cout << base\_f << " raised to the power of " << exponent\_f << " is " << power(base\_f, exponent\_f) << std::endl;

double n\_f = 5.0;

std::cout << "Factorial of " << n\_f << " is " << factorial(n\_f) << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

return 0;

}

Polynomial Addition: Implement a class Polynomial to represent polynomials with terms (coefficient and exponent). Overload the + operator to add two Polynomial objects and return a new Polynomial with the combined terms.

#include <iostream>

#include <map>

class Polynomial {

private:

std::map<int, double> terms; // key: exponent, value: coefficient

public:

Polynomial() {}

void addTerm(double coefficient, int exponent) {

terms[exponent] += coefficient;

}

Polynomial operator+(const Polynomial& other) const {

Polynomial result = \*this;

for (const auto& term : other.terms) {

result.terms[term.first] += term.second;

}

return result;

}

void print() const {

for (const auto& term : terms) {

std::cout << term.second << "x^" << term.first << " ";

}

std::cout << std::endl;

}

};

int main() {

Polynomial p1;

p1.addTerm(2, 2);

p1.addTerm(3, 1);

Polynomial p2;

p2.addTerm(1, 2);

p2.addTerm(4, 0);

Polynomial p3 = p1 + p2;

p3.print(); // Output: 3x^2 3x^1 4x^0

return 0;

}

Money Class: Design a class Money to store currency amount and type (e.g., USD, EUR). Overload the comparison operators (==, !=, <, >, <=, >=) for Money objects, considering currency types and exchange rates.

#include <iostream>

#include <string>

class Money {

private:

double amount;

std::string currency;

public:

Money(double amt, std::string curr) : amount(amt), currency(curr) {}

Money operator+(const Money& other) const {

if (currency != other.currency) {

throw std::invalid\_argument("Currency types do not match");

}

return Money(amount + other.amount, currency);

}

Money operator-(const Money& other) const {

if (currency != other.currency) {

throw std::invalid\_argument("Currency types do not match");

}

return Money(amount - other.amount, currency);

}

Money operator\*(double multiplier) const {

return Money(amount \* multiplier, currency);

}

Money operator/(double divisor) const {

if (divisor == 0) {

throw std::invalid\_argument("Division by zero");

}

return Money(amount / divisor, currency);

}

void print() const {

std::cout << amount << " " << currency << std::endl;

}

};

int main() {

Money m1(100, "USD");

Money m2(200, "USD");

Money m3 = m1 + m2;

m3.print(); // Output: 300 USD

Money m4 = m1 - m2;

m4.print(); // Output: -100 USD

Money m5 = m1 \* 2;

m5.print(); // Output: 200 USD

Money m6 = m2 / 2;

m6.print(); // Output: 100 USD

return 0;

}

String Stream Insertion: Overload the stream insertion operator (<<) for a custom String class to allow easy printing of strings to standard output.

#include <iostream>

#include <string>

class String {

private:

std::string data;

public:

String(const std::string& str) : data(str) {}

friend std::ostream& operator<<(std::ostream& os, const String& str) {

os << str.data;

return os;

}

};

int main() {

String str("Hello, World!");

std::cout << str << std::endl; // Output: Hello, World!

return 0;

}

Minimum and Maximum: Create overloaded functions min and max that can handle different data types (e.g., int, double) and return the minimum or maximum value.

#include <iostream>

int min(int a, int b) {

return (a < b) ? a : b;

}

double min(double a, double b) {

return (a < b) ? a : b;

}

double min(int a, double b) {

return (a < b) ? a : b;

}

double min(double a, int b) {

return (a < b) ? a : b;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

double max(double a, double b) {

return (a > b) ? a : b;

}

double max(int a, double b) {

return (a > b) ? a : b;

}

double max(double a, int b) {

return (a > b) ? a : b;

}

int main() {

int int\_a = 5, int\_b = 10;

std::cout << "min(" << int\_a << ", " << int\_b << ") = " << min(int\_a, int\_b) << std::endl;

std::cout << "max(" << int\_a << ", " << int\_b << ") = " << max(int\_a, int\_b) << std::endl;

double double\_a = 5.5, double\_b = 10.1;

std::cout << "min(" << double\_a << ", " << double\_b << ") = " << min(double\_a, double\_b) << std::endl;

std::cout << "max(" << double\_a << ", " << double\_b << ") = " << max(double\_a, double\_b) << std::endl;

std::cout << "min(" << int\_a << ", " << double\_b << ") = " << min(int\_a, double\_b) << std::endl;

std::cout << "max(" << int\_a << ", " << double\_b << ") = " << max(int\_a, double\_b) << std::endl;

std::cout << "min(" << double\_a << ", " << int\_b << ") = " << min(double\_a, int\_b) << std::endl;

std::cout << "max(" << double\_a << ", " << int\_b << ") = " << max(double\_a, int\_b) << std::endl;

return 0;

}

Array Statistics: Implement overloaded functions average, minimum, and maximum that can take an array of integers or doubles as input, depending on the function call.

#include <iostream>

#include <stdexcept>

// Average function for integer array

double average(const int arr[], int size) {

if (size <= 0) {

throw std::invalid\_argument("Array size must be greater than 0.");

}

int sum = 0;

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

sum += arr[i];

}

return static\_cast<double>(sum) / size;

}

// Average function for double array

double average(const double arr[], int size) {

if (size <= 0) {

throw std::invalid\_argument("Array size must be greater than 0.");

}

double sum = 0.0;

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

sum += arr[i];

}

return sum / size;

}

// Minimum function for integer array

int minimum(const int arr[], int size) {

if (size <= 0) {

throw std::invalid\_argument("Array size must be greater than 0.");

}

int minValue = arr[0];

for (int i = 1; i < size; ++i) {

if (arr[i] < minValue) {

minValue = arr[i];

}

}

return minValue;

}

// Minimum function for double array

double minimum(const double arr[], int size) {

if (size <= 0) {

throw std::invalid\_argument("Array size must be greater than 0.");

}

double minValue = arr[0];

for (int i = 1; i < size; ++i) {

if (arr[i] < minValue) {

minValue = arr[i];

}

return minValue;

}

int maximum(const int arr[], int size) {

if (size <= 0) {

throw std::invalid\_argument("Array size must be greater than 0.");

}

int maxValue = arr[0];

for (int i = 1; i < size; ++i) {

if (arr[i] > maxValue) {

maxValue = arr[i];

}

}

return maxValue;

}

double maximum(const double arr[], int size) {

if (size <= 0) {

throw std::invalid\_argument("Array size must be greater than 0.");

}

double maxValue = arr[0];

for (int i = 1; i < size; ++i) {

if (arr[i] > maxValue) {

maxValue = arr[i];

}

}

return maxValue;

}

int main() {

int intArray[] = {1, 2, 3, 4, 5};

double doubleArray[] = {1.1, 2.2, 3.3, 4.4, 5.5};

int intSize = sizeof(intArray) / sizeof(intArray[0]);

int doubleSize = sizeof(doubleArray) / sizeof(doubleArray[0]);

std::cout << "Average of int array: " << average(intArray, intSize) << std::endl;

std::cout << "Average of double array: " << average(doubleArray, doubleSize) << std::endl;

std::cout << "Minimum of int array: " << minimum(intArray, intSize) << std::endl;

std::cout << "Minimum of double array: " << minimum(doubleArray, doubleSize) << std::endl;

std::cout << "Maximum of int array: " << maximum(intArray, intSize) << std::endl;

std::cout << "Maximum of double array: " << maximum(doubleArray, doubleSize) << std::endl;

return 0;

}