Question 1:

What are the benefits and drawbacks of operator overloading?

Benefits:

Operator overloading can make code that uses complex data types more readable and intuitive.

It allows user-defined types to behave like built-in types. This can make it easier to learn and use new types, as they can leverage familiar operators.

Operator overloading can contribute to polymorphism. Different objects can respond differently to the same operator, allowing for more flexible and extensible code.

Drawbacks:

Operator overloading can introduce complexity, making the code harder to understand and debug.

Overloaded operators can sometimes obscure the actual operations being performed, especially for developers unfamiliar with the codebase.

The compiler or interpreter needs to resolve overloaded operators at runtime, which can sometimes be less efficient than built-in operators.

Question 2:

Can you overload the assignment operator (=) in C++? If so, how would you ensure proper behavior?

Before performing any operations, check if the object is being assigned to itself. If this is equal to &other, simply return \*this.

Before allocating new resources, release any existing resources to avoid memory leaks. This is done by deleting the existing data.

Allocate new memory and copy the contents from the source object to the target object. This ensures that each object manages its own copy of the data.

The assignment operator should return a reference to the current object (\*this) to allow for chained assignments (e.g., a = b = c).

Ensure that the copy constructor and assignment operator perform similar tasks, both providing deep copies of the object's data.

Question 3:

Explain the difference between member function and non-member (friend) function overloading for operators.

Member Function Overloading:

In C++, operator overloading can be achieved through both member functions and non-member (friend) functions. Each approach has its own use cases and implications.

A member function is a function that is a member of a class and has access to the class's private and protected members. The operator is overloaded as a method inside the class.

Member functions have access to the private and protected members of the class.

The this pointer is implicitly passed, simplifying the syntax for accessing class members.

Suitable for operators that are naturally used with a class instance as the left-hand operand (e.g., obj1 + obj2).

Non-Member (Friend) Function Overloading:

A non-member function is a function that is not a member of the class. It can be a friend of the class, allowing it access to the class's private and protected members.

The operator is overloaded outside the class, but declared as a friend within the class to grant access to private members.

Suitable for operators that are naturally used with operands that may not be instances of the class (e.g., obj + 5 where 5 is not an object of the class).

Best for operators where the left operand is naturally an object of the class (e.g., assignment = or compound assignment +=).

Question 4:

Design a class Vector2D and overload the arithmetic operators (+, -, \*, /) for vector addition, subtraction, scalar multiplication, and division (by a scalar).

#include <iostream>

using namespace std;

class Vector2D {

private:

double x, y;

public:

// Constructor

Vector2D(double x=0, double y=0) : x(x), y(y) {}

// Addition

Vector2D operator+(const Vector2D& other) const {

return Vector2D(this->x + other.x, this->y + other.y);

}

// Subtraction

Vector2D operator-(const Vector2D& other) const {

return Vector2D(this->x - other.x, this->y - other.y);

}

// Scalar Multiplication

Vector2D operator\*(double scalar) const {

return Vector2D(this->x \* scalar, this->y \* scalar);

}

// Scalar Division

Vector2D operator/(double scalar) const {

if (scalar == 0) {

throw std::invalid\_argument("Division by zero is not allowed");

}

return Vector2D(this->x / scalar, this->y / scalar);

}

// Overload << operator for easy printing

friend std::ostream& operator<<(std::ostream& os, const Vector2D& vec) {

os << "Vector2D(" << vec.x << ", " << vec.y << ")";

return os;

}

};

int main() {

Vector2D v1(2, 3);

Vector2D v2(1, 1);

std::cout << v1 + v2 << std::endl; // Output: Vector2D(3, 4)

std::cout << v1 - v2 << std::endl; // Output: Vector2D(1, 2)

std::cout << v1 \* 2 << std::endl; // Output: Vector2D(4, 6)

std::cout << v1 / 2 << std::endl; // Output: Vector2D(1, 1.5)

return 0;

}

Question 5:

Is it possible to overload the comparison operators (==, !=, <, >, <=, >=) for custom classes? If so, what considerations should be taken into account?

In C++, the comparison operators for custom classes (==,!=, \, >, \=, >=) can indeed be overloaded.   
Make sure the overworked operators are in sync with one another. For instance, b > a should also be true if a < b is. In the same way, a == b ought to indicate that both a >= b and a <= b are true. To preserve symmetry, simultaneously overload the equality (==) and inequality (!=) operators.

Question 6:

\*Can you overload the stream insertion (<<) and extraction (>>) operators for your Vector2D class to allow easy printing and reading from streams?\*

Overloading the stream insertion (<<) and extraction (>>) operators for a Vector2D class allows for convenient printing to and reading from streams, such as std::cout and std::cin in C++. Outputs the Vector2D object in the format (x, y) to the output stream (std::ostream). Reads input of the form (x, y) from the input stream (std::istream) and assigns it to the Vector2D object.

Question 7:

Describe a scenario where overloading the logical operators (&&, ||, !) for a custom class might be useful.

Overloading the logical operators (&&, ||, !) for a custom class can be particularly useful in scenarios where the class represents a complex state or condition that can be logically evaluated.

&& (Logical AND): Checks if the user has all specified permissions.

|| (Logical OR): Checks if the user has at least one of the specified permissions.

! (Logical NOT): Checks if the user has no permissions at all.

Question 8:

Discuss the potential ambiguity that could arise when overloading the subscript operator ([]) for a class. How can this ambiguity be resolved?

The subscript operator is commonly used to access elements in containers like arrays, vectors, and maps. When you overload [] in a custom class, it's crucial to ensure that it is clear whether it is intended for element access or is being used to implement some other functionality. When overloading the subscript operator, you need to differentiate between returning a non-const reference (modifiable) and a const reference (read-only). Failing to do so can lead to ambiguities, especially when dealing with const objects.

Ensure that you provide separate overloads for const and non-const versions of the subscript operator. This distinction helps in avoiding ambiguities and makes it clear whether the operation is read-only or modifiable.

Returns a reference to allow modification of the element. Throws an exception if the index is out of bounds.

Question 9:

Can operator overloading be used to implement the concept of immutability (unchanging state) for a class? Explain your answer.

Yes, operator overloading can be used to implement the concept of immutability (unchanging state) for a class in C++. Immutability refers to the property where an object's state cannot be modified after it is created. Operator overloading can indeed be utilized to enforce immutability in C++ classes by carefully designing how operators are overloaded and ensuring that operations do not modify the internal state of objects. By returning const references or new objects instead of modifying the existing state directly, you can effectively create immutable classes that enhance code safety, reliability, and maintainability.

Question 10:

\*When overloading operators, what are some best practices to ensure code clarity and maintainability?\*

Make sure that the precedence, associativity, and expected behavior of the overloaded operator are all in line with those of the regular operators.  
Unexpected side effects or usage patterns that differ from normal should not occur in overloaded operators.  
Clearly state in comments or documentation what you want the overloaded operator to do.  
Unit tests should be used to confirm that your operator overloads behave as planned and correctly in a variety of scenarios and edge cases.  
By returning an element by value, it ensures const-correctness and can be used with const instances of Vector.  
operator+: Concatenates the data of two vectors to create a new vector without changing the original objects.  
operator\\: To ensure clarity and usability, overloads the output stream operator to print the vector in a readable format.

Function Overloading

Question: 1

What is the core concept behind function overloading?

The Core Concept behind the function overloading in programming language in c++ is providing the multiple functions with the same name but different parameter lists with in the same scope. Function overloading is allows to defining the several functions with the same name but different parameter types.

Functions that are overloaded must having the same name but it is different types of parameters. Function overloading is allows to write the code by logically related data in under in one name. it is improving the code readability and code usability.

By using the same function name for related operations and code become more readable and give answer and it is easy to understandable.

Question 2:

How does the compiler differentiate between overloaded functions with the same name?

Functions can be overloaded if they have a different number of parameters and Functions can be overloaded if they have parameters of different types. Functions can be overloaded if the order of parameters is different, even if the types are the same. When the compiler encounters a call to an overloaded function, it uses these criteria to determine which version of the function to call based on the arguments provided. This process is known as function signature matching or function resolution. The compiler ensures that the correct function is called by matching the function signature (the combination of the function's name and its parameter types) with the arguments used in the function call.

Question 3:

Can functions with different return types be overloaded? Explain your reasoning.

No, functions with only different return types cannot be overloaded. The compiler uses the function name and the parameters (number, types, and order) to differentiate between overloaded functions. The return type is not considered when resolving function calls because it is not part of the function's signature that the compiler uses for overload resolution.

Question 4:

Design a function printValue that can handle different data types (e.g., int, double,

#include <iostream>

#include <string>

// Function to print an integer

void printValue(int value) {

std::cout << "Integer: " << value << std::endl;

}

// Function to print a double

void printValue(double value) {

std::cout << "Double: " << value << std::endl;

}

// Function to print a string

void printValue(const std::string& value) {

std::cout << "String: " << value << std::endl;

}

int main() {

int intValue = 42;

double doubleValue = 3.14;

std::string stringValue = "Hello, World!";

printValue(intValue); // Calls the function to print an integer

printValue(doubleValue); // Calls the function to print a double

printValue(stringValue); // Calls the function to print a string

return 0;

}

Question 5:

Advantages

**Code Simplification**: Default arguments can reduce the number of overloads needed. This makes the code simpler and easier to read. For instance, if you have a function for configuring network settings, you can provide default values for optional parameters instead of creating multiple overloaded versions of the function.

**Maintenance Ease**: With fewer overloaded functions, maintaining the code becomes easier. Any changes to the function's core logic need to be made in only one place, reducing the risk of errors.

**Flexibility**: Default arguments provide flexibility to the function caller. They can choose to use the default values or provide specific ones, depending on the situation.

Dis advantages:

**Ambiguity in Overloading**: Combining overloaded functions with default arguments can lead to ambiguities, making it difficult for the compiler to decide which function to call. This can result in compilation errors or unintended behavior.

**Complex Debugging**: Default arguments can complicate debugging because the source of an issue may not be immediately clear. When debugging network configurations or data management logic, tracking down where and why a default argument was used can add to the complexity.

Question 6:

In the context of function overloading, explain the concept of argument promotion and implicit type conversion.

Argument Promotion

Argument promotion is a type of implicit conversion where smaller or less precise data types are automatically converted to larger or more precise types when passed to a function. This is done to ensure that operations within the function are performed using a consistent and sufficiently precise type.

### Implicit Type Conversion

Implicit type conversion (also known as type coercion) occurs when the compiler automatically converts one data type to another to match the type expected by a function. This can happen between different types that have a defined conversion relationship, including both built-in types and user-defined types with appropriate conversion operators or constructors.

Question 7:

When might it be a better idea to use separate functions with descriptive names instead of overloading a single function?

Using separate functions with descriptive names instead of overloading a single function might be a better idea in the following situation

**Distinct Functionality**: When the functions perform distinct operations or have significantly different purposes, using separate names can make the code more readable and easier to understand.

**Avoiding Ambiguity**: Overloading can sometimes lead to confusion or ambiguity, especially if the parameter differences are subtle. Clear, descriptive names can prevent this confusion.

**Simplifying Maintenance**: When each function has a clear, distinct purpose, maintaining and updating the code becomes easier.

**Specialized Behavior**: When functions need to handle specific cases or specialized behavior, using separate names can make these specializations explicit.

Using separate functions with descriptive names can enhance clarity, readability, maintainability, and specificity, making your code easier to understand and work with. Overloading is useful for closely related functionalities, but when functions have distinct purposes or involve complex parameter sets, descriptive names are often the better choice.

Question 8:

Compare and contrast function overloading with virtual functions in C++ inheritance. Which approach is more suitable for specific use cases?

**Definition**: Function overloading is the process of defining multiple functions with the same name but different parameter lists (number, types, or order of parameters) within the same scope.

 Compile-Time Polymorphism: The decision about which overloaded function to call is made at compile time.

 Static Binding: The function to be executed is determined based on the function signature during compilation.

Example:

class Base {

public:

virtual void display() const {

std::cout << "Display Base" << std::endl;

}

};

class Derived : public Base {

public:

void display() const override {

std::cout << "Display Derived" << std::endl;

}

};

void show(const Base& obj) {

obj.display();

}

int main() {

Base b;

Derived d;

show(b); // Outputs: Display Base

show(d); // Outputs: Display Derived

}

Programs:

Complex Numbers (C++) - Define a class Complex to represent complex numbers with member variables for real and imaginary parts. Overload the +, -, and \* operators for complex number addition, subtraction, and multiplication.

#include <iostream>

class Complex {

private:

double real;

double imag;

public:

// Constructor

complex(double r = 0, double i = 0) : real(r), imag(i) {}

// Overload the + operator

Complex operator+(const Complex& other) const {

return Complex(real + other.real, imag + other.imag);

}

// Overload the - operator

Complex operator-(const Complex& other) const {

return Complex(real - other.real, imag - other.imag);

}

// Overload the \* operator

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

return Complex(

real \* other.real - imag \* other.imag,

real \* other.imag + imag \* other.real

);

}

// Friend function to overload << for printing

friend std::ostream& operator<<(std::ostream& out, const Complex& c) {

out << c.real << " + " << c.imag << "i";

return out;

}

};

int main() {

Complex c1(3.0, 2.0);

Complex c2(1.5, 4.5);

Complex sum = c1 + c2;

Complex diff = c1 - c2;

Complex prod = c1 \* c2;

std::cout << "c1: " << c1 << std::endl;

std::cout << "c2: " << c2 << std::endl;

std::cout << "Sum: " << sum << std::endl;

std::cout << "Difference: " << diff << std::endl;

std::cout << "Product: " << prod << std::endl;

return 0;

}

Question 2:

- Create a class Point2D with x and y coordinates. Overload the + operator to return a new Point2D object representing the sum of two points.

#include <iostream>

class Point2D {

private:

double x;

double y;

public:

// Constructor

Point2D(double x = 0, double y = 0) : x(x), y(y) {}

// Overload the + operator

Point2D operator+(const Point2D& other) const {

return Point2D(x + other.x, y + other.y);

}

// Getter for x

double getX() const {

return x;

}

// Getter for y

double getY() const {

return y;

}

// Friend function to overload << for printing

friend std::ostream& operator<<(std::ostream& out, const Point2D& point) {

out << "Point2D(" << point.x << ", " << point.y << ")";

return out;

}

};

int main() {

Point2D p1(1.5, 2.5);

Point2D p2(3.0, 4.0);

Point2D sum = p1 + p2;

std::cout << "p1: " << p1 << std::endl;

std::cout << "p2: " << p2 << std::endl;

std::cout << "Sum: " << sum << std::endl;

return 0;

}

Question 3:

Design a class Time to store hours, minutes, and seconds. Overload the + operator to add two Time objects and return a new Time object with the combined duration.

#include <iostream>

class Time {

private:

int hours;

int minutes;

int seconds;

public:

// Constructor

Time(int h = 0, int m = 0, int s = 0) : hours(h), minutes(m), seconds(s) {}

// Overload the + operator

Time operator+(const Time& other) const {

int totalSeconds = seconds + other.seconds;

int carryMinutes = totalSeconds / 60;

totalSeconds %= 60;

int totalMinutes = minutes + other.minutes + carryMinutes;

int carryHours = totalMinutes / 60;

totalMinutes %= 60;

int totalHours = hours + other.hours + carryHours;

return Time(totalHours, totalMinutes, totalSeconds);

}

// Getter for hours

int getHours() const {

return hours;

}

// Getter for minutes

int getMinutes() const {

return minutes;

}

// Getter for seconds

int getSeconds() const {

return seconds;

}

// Friend function to overload << for printing

friend std::ostream& operator<<(std::ostream& out, const Time& time) {

out << "Time: " << time.hours << "h " << time.minutes << "m " << time.seconds << "s";

return out;

}

};

int main() {

Time t1(3, 45, 30);

Time t2(2, 15, 45);

Time sum = t1 + t2;

std::cout << "t1: " << t1 << std::endl;

std::cout << "t2: " << t2 << std::endl;

std::cout << "Sum: " << sum << std::endl;

return 0;

}

Question 4:

Implement a class Date with year, month, and day. Overload the comparison operators (== and !=) to compare two Date objects.

#include <iostream>

class Date {

private:

int year;

int month;

int day;

public:

// Constructor

Date(int y = 0, int m = 0, int d = 0) : year(y), month(m), day(d) {}

// Overload the equality operator (==)

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

return (year == other.year && month == other.month && day == other.day);

}

// Overload the inequality operator (!=)

bool operator!=(const Date& other) const {

return !(\*this == other);

}

// Getter for year

int getYear() const {

return year;

}

// Getter for month

int getMonth() const {

return month;

}

// Getter for day

int getDay() const {

return day;

}

};

int main() {

Date d1(2023, 6, 27);

Date d2(2023, 6, 27);

Date d3(2023, 6, 28);

// Test equality operator

if (d1 == d2) {

std::cout << d1.getYear() << "/" << d1.getMonth() << "/" << d1.getDay() << " is equal to "

<< d2.getYear() << "/" << d2.getMonth() << "/" << d2.getDay() << std::endl;

} else {

std::cout << "Dates are not equal." << std::endl;

}

// Test inequality operator

if (d1 != d3) {

std::cout << d1.getYear() << "/" << d1.getMonth() << "/" << d1.getDay() << " is not equal to "

<< d3.getYear() << "/" << d3.getMonth() << "/" << d3.getDay() << std::endl;

} else {

std::cout << "Dates are equal." << std::endl;

}

return 0;

}

Question 5:

String Equality (C++) - Overload the equality operator (==) for a custom String class to compare string contents (not just memory addresses).

#include <iostream>

#include <cstring> // for strlen and strcmp

class String {

private:

char\* str;

public:

// Constructor

String(const char\* s = nullptr) {

if (s) {

str = new char[strlen(s) + 1];

strcpy(str, s);

} else {

str = nullptr;

}

}

// Destructor

~String() {

delete[] str;

}

// Copy constructor

String(const String& other) {

if (other.str) {

str = new char[strlen(other.str) + 1];

strcpy(str, other.str);

} else {

str = nullptr;

}

}

// Assignment operator

String& operator=(const String& other) {

if (this != &other) {

delete[] str;

if (other.str) {

str = new char[strlen(other.str) + 1];

strcpy(str, other.str);

} else {

str = nullptr;

}

}

return \*this;

}

// Overload the equality operator (==)

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

return (strcmp(str, other.str) == 0);

}

// Getter for string

const char\* getStr() const {

return str;

}

};

int main() {

String s1("Hello");

String s2("Hello");

String s3("World");

// Test equality operator

if (s1 == s2) {

std::cout << "s1: " << s1.getStr() << " is equal to s2: " << s2.getStr() << std::endl;

} else {

std::cout << "s1: " << s1.getStr() << " is not equal to s2: " << s2.getStr() << std::endl;

}

if (s1 == s3) {

std::cout << "s1: " << s1.getStr() << " is equal to s3: " << s3.getStr() << std::endl;

} else {

std::cout << "s1: " << s1.getStr() << " is not equal to s3: " << s3.getStr() << std::endl;

}

return 0;

}

Question 6:

Create a function calculateArea that can handle different shapes (e.g., rectangle, circle) by overloading it with parameters like width, height, or radius.

#include <iostream>

#include <cmath> // for M\_PI and pow

// Function to calculate the area of a rectangle

double calculateArea(double width, double height) {

return width \* height;

}

// Function to calculate the area of a circle

double calculateArea(double radius) {

return M\_PI \* pow(radius, 2);

}

int main() {

// Calculate area of a rectangle

double rectWidth = 5.0;

double rectHeight = 3.0;

double rectArea = calculateArea(rectWidth, rectHeight);

std::cout << "Area of rectangle with width " << rectWidth << " and height " << rectHeight << " is: " << rectArea << std::endl;

// Calculate area of a circle

double circleRadius = 2.5;

double circleArea = calculateArea(circleRadius);

std::cout << "Area of circle with radius " << circleRadius << " is: " << circleArea << std::endl;

return 0;

}

Question 7:

Design a function convert that takes a value and a unit (e.g., meters, feet, Celsius, Fahrenheit) and converts it to another unit using appropriate conversion factors.

#include <iostream>

// Constants for conversion factors

const double METERS\_TO\_FEET = 3.28084;

const double CELSIUS\_TO\_FAHRENHEIT = 1.8;

const double FAHRENHEIT\_TO\_CELSIUS = 1 / CELSIUS\_TO\_FAHRENHEIT;

// Function to convert meters to feet

double convert(double meters, const std::string& toUnit) {

if (toUnit == "feet") {

return meters \* METERS\_TO\_FEET;

} else {

std::cerr << "Unsupported conversion to " << toUnit << std::endl;

return 0.0; // or handle error appropriately

}

}

// Function to convert feet to meters

double convert(int feet, const std::string& toUnit) {

if (toUnit == "meters") {

return feet / METERS\_TO\_FEET;

} else {

std::cerr << "Unsupported conversion to " << toUnit << std::endl;

return 0.0; // or handle error appropriately

}

}

// Function to convert Celsius to Fahrenheit

double convert(double celsius, const std::string& toUnit) {

if (toUnit == "Fahrenheit") {

return celsius \* CELSIUS\_TO\_FAHRENHEIT + 32.0;

} else {

std::cerr << "Unsupported conversion to " << toUnit << std::endl;

return 0.0; // or handle error appropriately

}

}

// Function to convert Fahrenheit to Celsius

double convert(int fahrenheit, const std::string& toUnit) {

if (toUnit == "Celsius") {

return (fahrenheit - 32.0) \* FAHRENHEIT\_TO\_CELSIUS;

} else {

std::cerr << "Unsupported conversion to " << toUnit << std::endl;

return 0.0; // or handle error appropriately

}

}

int main() {

double meters = 10.0;

std::cout << meters << " meters is equal to " << convert(meters, "feet") << " feet." << std::endl;

int feet = 30;

std::cout << feet << " feet is equal to " << convert(feet, "meters") << " meters." << std::endl;

double celsius = 25.0;

std::cout << celsius << " Celsius is equal to " << convert(celsius, "Fahrenheit") << " Fahrenheit." << std::endl;

int fahrenheit = 98;

std::cout << fahrenheit << " Fahrenheit is equal to " << convert(fahrenheit, "Celsius") << " Celsius." << std::endl;

return 0;

}

Question 8:

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

#include <iostream>

#include <limits> // for numeric\_limits

// Function template to calculate average of an array

template <typename T>

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

if (size <= 0) {

std::cerr << "Error: Array size must be greater than zero." << std::endl;

return 0.0; // or handle error appropriately

}

double sum = 0.0;

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

sum += static\_cast<double>(arr[i]); // convert to double for accurate average

}

return sum / size;

}

// Function template to find minimum value in an array

template <typename T>

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

if (size <= 0) {

std::cerr << "Error: Array size must be greater than zero." << std::endl;

return T(); // return default value for type T, or handle error appropriately

}

T minVal = arr[0];

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

if (arr[i] < minVal) {

minVal = arr[i];

}

}

return minVal;

}

// Function template to find maximum value in an array

template <typename T>

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

if (size <= 0) {

std::cerr << "Error: Array size must be greater than zero." << std::endl;

return T(); // return default value for type T, or handle error appropriately

}

T maxVal = arr[0];

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

if (arr[i] > maxVal) {

maxVal = arr[i];

}

}

return maxVal;

}

int main() {

// Example with integers

int intArray[] = {5, 3, 8, 1, 6};

int intSize = sizeof(intArray) / sizeof(int);

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

std::cout << "Minimum integer: " << minimum(intArray, intSize) << std::endl;

std::cout << "Maximum integer: " << maximum(intArray, intSize) << std::endl;

// Example with doubles

double doubleArray[] = {3.5, 1.2, 6.7, 2.1, 5.4};

int doubleSize = sizeof(doubleArray) / sizeof(double);

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

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

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

return 0;

}

Question 9:

Write overloaded functions formatString that can take a format string and different data types (e.g., int, double, string) to create formatted output strings

#include <iostream>

#include <sstream>

#include <iomanip> // for std::setw, std::setprecision

// Function to format string for integer

std::string formatString(const std::string& format, int value) {

std::stringstream ss;

ss << std::setfill('0') << std::setw(5) << value; // example formatting: zero-padded to 5 digits

return ss.str();

}

// Function to format string for double

std::string formatString(const std::string& format, double value) {

std::stringstream ss;

ss << std::fixed << std::setprecision(2) << value; // example formatting: fixed-point with 2 decimal places

return ss.str();

}

// Function to format string for string

std::string formatString(const std::string& format, const std::string& value) {

// example formatting: no special formatting needed for string

return value;

}

int main() {

int intValue = 123;

double doubleValue = 456.789;

std::string stringValue = "Hello";

std::cout << "Formatted integer: " << formatString("int", intValue) << std::endl;

std::cout << "Formatted double: " << formatString("double", doubleValue) << std::endl;

std::cout << "Formatted string: " << formatString("string", stringValue) << std::endl;

return 0;

}

Question 10:

Create 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 <cmath> // for pow function

// Function to calculate factorial of an integer

unsigned long long factorial(int n) {

if (n < 0) {

std::cerr << "Error: Factorial of negative number is undefined." << std::endl;

return 0; // or handle error appropriately

}

unsigned long long result = 1;

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

result \*= i;

}

return result;

}

// Overloaded function to calculate factorial of a double (approximation)

double factorial(double n) {

if (n < 0) {

std::cerr << "Error: Factorial of negative number is undefined." << std::endl;

return 0.0; // or handle error appropriately

}

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

return 1.0;

}

double result = 1.0;

for (double i = 2.0; i <= n; ++i) {

result \*= i;

}

return result;

}

// Function to calculate power of an integer

long long power(int base, int exponent) {

return static\_cast<long long>(std::pow(base, exponent));

}

// Overloaded function to calculate power of a double

double power(double base, double exponent) {

return std::pow(base, exponent);

}

int main() {

int intBase = 2;

int intExponent = 5;

double doubleBase = 2.5;

double doubleExponent = 3.0;

// Factorial examples

std::cout << "Factorial of " << intBase << " (integer): " << factorial(intBase) << std::endl;

std::cout << "Factorial of " << doubleBase << " (double): " << factorial(doubleBase) << std::endl;

// Power examples

std::cout << intBase << " raised to the power of " << intExponent << " (integer): " << power(intBase, intExponent) << std::endl;

std::cout << doubleBase << " raised to the power of " << doubleExponent << " (double): " << power(doubleBase, doubleExponent) << std::endl;

return 0;

}

Question 11:

Define 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 <vector>

#include <algorithm> // for std::max

// Structure to represent a term in the polynomial

struct Term {

double coefficient;

int exponent;

// Constructor to initialize coefficient and exponent

Term(double coeff = 0.0, int exp = 0) : coefficient(coeff), exponent(exp) {}

};

// Class to represent a Polynomial

class Polynomial {

private:

std::vector<Term> terms; // Vector to store polynomial terms

public:

// Constructor to initialize Polynomial with a single term

Polynomial(double coeff = 0.0, int exp = 0) {

terms.emplace\_back(coeff, exp);

}

// Constructor to initialize Polynomial with a vector of terms

Polynomial(const std::vector<Term>& termsVec) : terms(termsVec) {}

// Overloaded + operator to add two Polynomials

Polynomial operator+(const Polynomial& other) const {

std::vector<Term> resultTerms;

// Merge terms from both polynomials

auto mergeTerm = [&](const Term& term) {

auto it = std::find\_if(resultTerms.begin(), resultTerms.end(),

[&](const Term& t) { return t.exponent == term.exponent; });

if (it != resultTerms.end()) {

it->coefficient += term.coefficient;

} else {

resultTerms.push\_back(term);

}

};

// Add terms from this polynomial

for (const auto& term : terms) {

mergeTerm(term);

}

// Add terms from the other polynomial

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

mergeTerm(term);

}

// Sort terms by exponent in descending order

std::sort(resultTerms.begin(), resultTerms.end(),

[](const Term& a, const Term& b) { return a.exponent > b.exponent; });

return Polynomial(resultTerms);

}

// Function to print the polynomial

void print() const {

bool firstTerm = true;

for (const auto& term : terms) {

if (term.coefficient != 0.0) {

if (!firstTerm && term.coefficient > 0) {

std::cout << "+ ";

}

std::cout << term.coefficient;

if (term.exponent > 0) {

std::cout << "x^" << term.exponent << " ";

}

firstTerm = false;

}

}

std::cout << std::endl;

}

};

int main() {

// Example usage of Polynomial class and operator overloading

// Create first polynomial: 3x^2 + 2x + 1

Polynomial poly1({{3.0, 2}, {2.0, 1}, {1.0, 0}});

std::cout << "Polynomial 1: ";

poly1.print();

// Create second polynomial: 4x^3 - 2x^2 + 5

Polynomial poly2({{4.0, 3}, {-2.0, 2}, {0.0, 1}, {5.0, 0}});

std::cout << "Polynomial 2: ";

poly2.print();

// Add polynomials poly1 and poly2

Polynomial result = poly1 + poly2;

std::cout << "Sum of Polynomial 1 and Polynomial 2: ";

result.print();

return 0;

}

Question 12:

Design a class Money to store currency amount and type (e.g., USD, EUR). Overload the arithmetic operators (+, -, \*, /) for money objects, considering appropriate type conversions and potential rounding issues.

#include <iostream>

#include <string>

#include <cmath> // for std::round

// Enum for currency types

enum class Currency {

USD,

EUR

};

// Class to represent Money with currency amount and type

class Money {

private:

double amount;

Currency currency;

public:

// Constructor

Money(double amt = 0.0, Currency curr = Currency::USD) : amount(amt), currency(curr) {}

// Overloaded arithmetic operators

Money operator+(const Money& other) const {

if (currency != other.currency) {

std::cerr << "Error: Cannot add money amounts of different currencies." << std::endl;

return Money(); // or handle error appropriately

}

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

}

Money operator-(const Money& other) const {

if (currency != other.currency) {

std::cerr << "Error: Cannot subtract money amounts of different currencies." << std::endl;

return Money(); // or handle error appropriately

}

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

}

Money operator\*(double multiplier) const {

return Money(amount \* multiplier, currency);

}

Money operator/(double divisor) const {

if (divisor == 0.0) {

std::cerr << "Error: Division by zero." << std::endl;

return Money(); // or handle error appropriately

}

return Money(amount / divisor, currency);

}

// Function to print the Money object

void print() const {

std::string currStr = (currency == Currency::USD) ? "USD" : "EUR";

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

}

};

int main() {

// Example usage of Money class and operator overloading

Money money1(100.0, Currency::USD);

Money money2(50.0, Currency::USD);

Money money3(75.0, Currency::EUR);

// Arithmetic operations

Money sumUSD = money1 + money2;

Money differenceUSD = money1 - money2;

Money scaledMoney = money3 \* 1.5;

Money dividedMoney = money1 / 4.0;

// Print results

std::cout << "Sum of money1 and money2: ";

sumUSD.print();

std::cout << "Difference of money1 and money2: ";

differenceUSD.print();

std::cout << "Scaled money3 by 1.5: ";

scaledMoney.print();

std::cout << "Money1 divided by 4: ";

dividedMoney.print();

return 0;

}

Question 13:

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>

// Base class Shape with pure virtual function getArea

class Shape {

public:

// Pure virtual function to calculate area (abstract method)

virtual double getArea() const = 0;

// Virtual destructor (to allow proper cleanup of derived classes)

virtual ~Shape() {}

};

// Derived class Circle from Shape

class Circle : public Shape {

private:

double radius;

public:

// Constructor

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

// Implementation of getArea for Circle

double getArea() const override {

return 3.14159 \* radius \* radius;

}

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

private:

double width;

double height;

public:

// Constructor

Rectangle(double w, double h) : width(w), height(h) {}

// Implementation of getArea for Rectangle

double getArea() const override {

return width \* height;

}

};

// Derived class Square from Shape

class Square : public Shape {

private:

double side;

public:

// Constructor

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

// Implementation of getArea for Square

double getArea() const override {

return side \* side;

}

};

// Function to calculate and print area of any Shape object

void printArea(const Shape& shape) {

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

}

int main() {

// Example usage of Shape and its derived classes

Circle circle(5.0);

Rectangle rectangle(4.0, 3.0);

Square square(2.5);

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

printArea(circle);

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

printArea(rectangle);

std::cout << "Square:" << std::endl;

printArea(square);

return 0;

}

Question 14:

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 to represent an item with name, price, and quantity

class Item {

private:

std::string name;

double price;

int quantity;

public:

// Constructor

Item(const std::string& n = "", double p = 0.0, int q = 0)

: name(n), price(p), quantity(q) {}

// Getter methods

std::string getName() const { return name; }

double getPrice() const { return price; }

int getQuantity() const { return quantity; }

// Overloading << operator for easy printing of item details

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

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

return os;

}

};

int main() {

// Example usage of Item class and << operator overloading

Item item1("Laptop", 999.99, 2);

Item item2("Headphones", 149.95, 3);

std::cout << "Item 1 details: " << item1 << std::endl;

std::cout << "Item 2 details: " << item2 << std::endl;

return 0;

}

Question 15:

Design a class FileReaderWriter with overloaded functions read and write that can handle different file types (e.g., text, CSV) based on the file extension provided as input.

#include <iostream>

#include <fstream>

#include <string>

// Class FileReaderWriter to handle different file types

class FileReaderWriter {

public:

// Function to read from a file based on file extension

void read(const std::string& filename) {

std::string extension = getFileExtension(filename);

if (extension == "txt") {

readTextFile(filename);

} else if (extension == "csv") {

readCsvFile(filename);

} else {

std::cerr << "Unsupported file type." << std::endl;

}

}

// Function to write to a file based on file extension

void write(const std::string& filename, const std::string& content) {

std::string extension = getFileExtension(filename);

if (extension == "txt") {

writeTextFile(filename, content);

} else if (extension == "csv") {

writeCsvFile(filename, content);

} else {

std::cerr << "Unsupported file type." << std::endl;

}

}

private:

// Helper function to extract file extension from filename

std::string getFileExtension(const std::string& filename) const {

size\_t dotPos = filename.find\_last\_of(".");

if (dotPos == std::string::npos) {

return ""; // No extension found

}

return filename.substr(dotPos + 1);

}

// Function to read from a text file

void readTextFile(const std::string& filename) {

std::ifstream file(filename);

if (!file) {

std::cerr << "Error opening file: " << filename << std::endl;

return;

}

std::string line;

while (std::getline(file, line)) {

std::cout << line << std::endl;

}

file.close();

}

// Function to read from a CSV file (dummy implementation)

void readCsvFile(const std::string& filename) {

std::cout << "Reading CSV file: " << filename << std::endl;

// Implement CSV file reading logic as needed

}

// Function to write to a text file

void writeTextFile(const std::string& filename, const std::string& content) {

std::ofstream file(filename);

if (!file) {

std::cerr << "Error opening file: " << filename << std::endl;

return;

}

file << content;

file.close();

}

// Function to write to a CSV file (dummy implementation)

void writeCsvFile(const std::string& filename, const std::string& content) {

std::cout << "Writing to CSV file: " << filename << std::endl;

// Implement CSV file writing logic as needed

}

};

int main() {

// Example usage of FileReaderWriter class

FileReaderWriter fileRW;

// Reading from files

std::cout << "Reading text file:" << std::endl;

fileRW.read("example.txt");

std::cout << "\nReading CSV file:" << std::endl;

fileRW.read("data.csv");

// Writing to files

std::cout << "\nWriting to text file:" << std::endl;

fileRW.write("output.txt", "Hello, world!");

std::cout << "\nWriting to CSV file:" << std::endl;

fileRW.write("output.csv", "Name, Age, City\nJohn, 25, New York\nJane, 30, Los Angeles");

return 0;

}

Question 16:

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>

// Class Matrix to store a 2D array and perform matrix operations

class Matrix {

private:

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

size\_t rows;

size\_t cols;

public:

// Constructor to initialize matrix dimensions and data

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

// Getter methods

size\_t getRows() const { return rows; }

size\_t getCols() const { return cols; }

// Overloading operator[] to access elements of the matrix

std::vector<int>& operator[](size\_t idx) {

return data[idx];

}

const std::vector<int>& operator[](size\_t idx) const {

return data[idx];

}

// Overloading addition operator (+) for matrix addition

Matrix operator+(const Matrix& other) const {

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

throw std::invalid\_argument("Matrix dimensions must be the same for addition.");

}

Matrix result(rows, cols);

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

for (size\_t j = 0; j < cols; ++j) {

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

}

}

return result;

}

// Overloading subtraction operator (-) for matrix subtraction

Matrix operator-(const Matrix& other) const {

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

throw std::invalid\_argument("Matrix dimensions must be the same for subtraction.");

}

Matrix result(rows, cols);

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

for (size\_t j = 0; j < cols; ++j) {

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

}

}

return result;

}

// Overloading multiplication operator (\*) for matrix multiplication

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

if (cols != other.rows) {

throw std::invalid\_argument("Matrix dimensions must be compatible for multiplication.");

}

size\_t resultRows = rows;

size\_t resultCols = other.cols;

Matrix result(resultRows, resultCols);

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

for (size\_t j = 0; j < resultCols; ++j) {

int sum = 0;

for (size\_t k = 0; k < cols; ++k) {

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

}

result[i][j] = sum;

}

}

return result;

}

// Function to print the matrix

void print() const {

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

for (size\_t j = 0; j < cols; ++j) {

std::cout << data[i][j] << " ";

}

std::cout << std::endl;

}

}

};

int main() {

// Example usage of Matrix class and operator overloading

Matrix A(2, 3);

A[0][0] = 1; A[0][1] = 2; A[0][2] = 3;

A[1][0] = 4; A[1][1] = 5; A[1][2] = 6;

Matrix B(3, 2);

B[0][0] = 7; B[0][1] = 8;

B[1][0] = 9; B[1][1] = 10;

B[2][0] = 11; B[2][1] = 12;

std::cout << "Matrix A:" << std::endl;

A.print();

std::cout << "\nMatrix B:" << std::endl;

B.print();

// Matrix addition

std::cout << "\nMatrix A + B:" << std::endl;

Matrix C = A + B;

C.print();

// Matrix subtraction

std::cout << "\nMatrix A - B:" << std::endl;

Matrix D = A - B;

D.print();

// Matrix multiplication

std::cout << "\nMatrix A \* B:" << std::endl;

Matrix E = A \* B;

E.print();

return 0;

}

Question 17:

Implement 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> // for std::out\_of\_range

// Class CustomList behaving like a list with subscript operator overloading

template <typename T>

class CustomList {

private:

std::vector<T> data;

public:

// Constructor to initialize the list

CustomList() {}

// Function to add elements to the list

void add(const T& element) {

data.push\_back(element);

}

// Overloaded subscript operator for element access with boundary checking

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

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

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

}

return data[index];

}

// Const version of subscript operator for element access with boundary checking

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

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

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

}

return data[index];

}

// Function to get the size of the list

size\_t size() const {

return data.size();

}

};

int main() {

// Example usage of CustomList class with subscript operator overloading

CustomList<int> list;

list.add(10);

list.add(20);

list.add(30);

// Access elements using subscript operator with boundary checking

try {

std::cout << "Element at index 0: " << list[0] << std::endl;

std::cout << "Element at index 1: " << list[1] << std::endl;

std::cout << "Element at index 2: " << list[2] << std::endl;

// Uncomment to test out-of-bounds access:

// std::cout << "Element at index 3: " << list[3] << std::endl;

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

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

}

return 0;

}

Question 18:

Design 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; // Pointer to the object

public:

// Constructor

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

// Destructor

~MySmartPtr() {

delete ptr; // Release memory when MySmartPtr goes out of scope

}

// Overloaded dereference operator (\*)

T& operator\*() const {

return \*ptr; // Return reference to the pointed-to object

}

// Overloaded arrow operator (->)

T\* operator->() const {

return ptr; // Return pointer to the object

}

// Function to release ownership of the pointed object

void release() {

ptr = nullptr; // Set pointer to null, no longer owns the object

}

// Function to get the raw pointer

T\* get() const {

return ptr;

}

};

// Example class

class Test {

public:

void print() {

std::cout << "Hello from Test!" << std::endl;

}

};

int main() {

// Example usage of MySmartPtr with Test class

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

ptr->print(); // Using arrow operator to call Test's member function

(\*ptr).print(); // Using dereference operator to call Test's member function

ptr.release(); // Release ownership

return 0;

}

Question 19:

Create a class CharacterStream that overloads the << operator to provide a custom way of writing characters to a stream, applying transformations or filters if needed.

#include <iostream>

#include <sstream>

#include <cctype> // for std::toupper

class CharacterStream {

private:

std::ostream& output; // Reference to output stream

public:

// Constructor initializes with a reference to an output stream

explicit CharacterStream(std::ostream& out) : output(out) {}

// Overloaded << operator to write characters with transformation/filtering

CharacterStream& operator<<(char ch) {

if (std::islower(ch)) {

ch = std::toupper(ch); // Transform lowercase to uppercase

}

output << ch; // Write transformed character to output stream

return \*this; // Return reference to the CharacterStream object

}

};

int main() {

// Example usage of CharacterStream class with custom transformation/filtering

std::ostringstream oss; // Using a stringstream as the output stream

CharacterStream cs(oss);

// Writing characters to CharacterStream with custom transformation/filtering

cs << 'a' << 'b' << 'c' << 'X' << 'y' << 'z';

std::cout << "Transformed output: " << oss.str() << std::endl;

return 0;

}

Question 20:

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>

#include <stdexcept> // for std::out\_of\_range

template <typename T>

class Vector {

private:

std::vector<T> elements;

public:

// Default constructor

Vector() = default;

// Constructor to initialize with an optional list of elements

Vector(std::initializer\_list<T> initList) : elements(initList) {}

// Function to add elements to the vector

void add(const T& element) {

elements.push\_back(element);

}

// Overloaded addition operator (+)

Vector<T> operator+(const Vector<T>& other) const {

Vector<T> result;

size\_t minSize = std::min(elements.size(), other.elements.size());

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

result.add(elements[i] + other.elements[i]);

}

// Add remaining elements from the longer vector

for (size\_t i = minSize; i < elements.size(); ++i) {

result.add(elements[i]);

}

for (size\_t i = minSize; i < other.elements.size(); ++i) {

result.add(other.elements[i]);

}

return result;

}

// Overloaded subtraction operator (-)

Vector<T> operator-(const Vector<T>& other) const {

Vector<T> result;

size\_t minSize = std::min(elements.size(), other.elements.size());

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

result.add(elements[i] - other.elements[i]);

}

// Add remaining elements from the longer vector

for (size\_t i = minSize; i < elements.size(); ++i) {

result.add(elements[i]);

}

for (size\_t i = minSize; i < other.elements.size(); ++i) {

result.add(-other.elements[i]);

}

return result;

}

// Overloaded subscript operator ([])

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

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

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

}

return elements[index];

}

// Const version of subscript operator ([]) for const objects

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

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

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

}

return elements[index];

}

// Function to get the size of the vector

size\_t size() const {

return elements.size();

}

};

int main() {

// Example usage of Vector class with operator overloading

Vector<int> v1{1, 2, 3}; // Corrected initialization with initializer list

Vector<int> v2{4, 5, 6}; // Corrected initialization with initializer list

// Vector addition using operator+

Vector<int> v3 = v1 + v2;

// Vector subtraction using operator-

Vector<int> v4 = v1 - v2;

// Output results

std::cout << "v1: ";

for (size\_t i = 0; i < v1.size(); ++i) {

std::cout << v1[i] << " ";

}

std::cout << std::endl;

std::cout << "v2: ";

for (size\_t i = 0; i < v2.size(); ++i) {

std::cout << v2[i] << " ";

}

std::cout << std::endl;

std::cout << "v1 + v2: ";

for (size\_t i = 0; i < v3.size(); ++i) {

std::cout << v3[i] << " ";

}

std::cout << std::endl;

std::cout << "v1 - v2: ";

for (size\_t i = 0; i < v4.size(); ++i) {

std::cout << v4[i] << " ";

}

std::cout << std::endl;

return 0;

}

Programs:

Define a class Rectangle with member variables for width and height. Overload the + operator to return a new Rectangle object representing the sum of the areas of two rectangles.

#include <iostream>

using namespace std;

class Rectangle {

private:

double width;

double height;

public:

Rectangle(double w, double h) : width(w), height(h) {}

double getWidth() const { return width; }

double getHeight() const { return height; }

double area() const {

return width \* height;

}

Rectangle operator+(const Rectangle& other) const {

double sumOfAreas = this->area() + other.area();

double newDimension = sqrt(sumOfAreas);

return Rectangle(newDimension, newDimension);

}

};

int main() {

Rectangle r1(3.0, 4.0);

Rectangle r2(2.0, 6.0);

Rectangle sumRect = r1 + r2;

std::cout << "Dimensions of summed rectangle: " << sumRect.getWidth() << " x " << sumRect.getHeight() << std::endl;

std::cout << "Area of summed rectangle: " << sumRect.area() << std::endl;

return 0;

}

Create a class Fraction with numerator and denominator. Overload the arithmetic operators (+, -, \*, /) for fraction addition, subtraction, multiplication, and division.

#include <iostream>

using namespace std;

class Fraction {

private:

int numerator;

int denominator;

public:

Fraction(int num = 0, int denom = 1) : numerator(num), denominator(denom) {

if (denominator == 0) {

std::cerr << "Error: Denominator cannot be zero. Setting denominator to 1." << std::endl;

denominator = 1;

}

simplify();

}

int getNumerator() const { return numerator; }

int getDenominator() const { return denominator; }

void simplify() {

int gcdVal = gcd(numerator, denominator);

numerator /= gcdVal;

denominator /= gcdVal;

if (denominator < 0) {

numerator = -numerator;

denominator = -denominator;

}

}

int gcd(int a, int b) const {

return b == 0 ? a : gcd(b, a % b);

}

Fraction operator+(const Fraction& other) const {

int newNumerator = numerator \* other.denominator + other.numerator \* denominator;

int newDenominator = denominator \* other.denominator;

return Fraction(newNumerator, newDenominator);

}

Fraction operator-(const Fraction& other) const {

int newNumerator = numerator \* other.denominator - other.numerator \* denominator;

int newDenominator = denominator \* other.denominator;

return Fraction(newNumerator, newDenominator);

}

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

int newNumerator = numerator \* other.numerator;

int newDenominator = denominator \* other.denominator;

return Fraction(newNumerator, newDenominator);

}

Fraction operator/(const Fraction& other) const {

int newNumerator = numerator \* other.denominator;

int newDenominator = denominator \* other.numerator;

return Fraction(newNumerator, newDenominator);

}

void display() const {

std::cout << numerator << "/" << denominator;

}

};

int main() {

Fraction f1(2, 3);

Fraction f2(3, 4);

Fraction sum = f1 + f2;

std::cout << "Sum: ";

sum.display();

std::cout << std::endl;

Fraction diff = f1 - f2;

std::cout << "Difference: ";

diff.display();

std::cout << std::endl;

Fraction prod = f1 \* f2;

std::cout << "Product: ";

prod.display();

std::cout << std::endl;

Fraction div = f1 / f2;

std::cout << "Division: ";

div.display();

std::cout << std::endl;

return 0;

}

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>

#include <map>

using namespace std;

class Money {

private:

double amount;

std::string currencyType;

static std::map<std::string, double> exchangeRates;

public:

Money(double amt = 0.0, const std::string& currency = "USD")

: amount(amt), currencyType(currency) {}

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

return convertToUSD() == other.convertToUSD();

}

bool operator!=(const Money& other) const {

return !(\*this == other);

}

bool operator<(const Money& other) const {

return convertToUSD() < other.convertToUSD();

}

bool operator>(const Money& other) const {

return convertToUSD() > other.convertToUSD();

}

bool operator<=(const Money& other) const {

return convertToUSD() <= other.convertToUSD();

}

bool operator>=(const Money& other) const {

return convertToUSD() >= other.convertToUSD();

}

double convertToUSD() const {

if (currencyType == "USD") {

return amount;

} else if (exchangeRates.find(currencyType) != exchangeRates.end()) {

return amount / exchangeRates.at(currencyType);

} else {

std::cerr << "Error: Exchange rate not found for currency " << currencyType << std::endl;

return 0.0;

}

}

static void setExchangeRate(const std::string& currency, double rate) {

exchangeRates[currency] = rate;

}

};

std::map<std::string, double> Money::exchangeRates = {

{"USD", 1.0},

{"EUR", 0.85},

{"GBP", 0.75}

};

int main() {

Money m1(100.0, "USD");

Money m2(85.0, "EUR");

if (m1 == m2) {

std::cout << "m1 equals m2" << std::endl;

}

if (m1 != m2) {

std::cout << "m1 does not equal m2" << std::endl;

}

if (m1 < m2) {

std::cout << "m1 is less than m2" << std::endl;

}

if (m1 > m2) {

std::cout << "m1 is greater than m2" << std::endl;

}

if (m1 <= m2) {

std::cout << "m1 is less than or equal to m2" << std::endl;

}

if (m1 >= m2) {

std::cout << "m1 is greater than or equal to m2" << std::endl;

}

return 0;

}

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

#include <iostream>

#include <cstring>

using namespace std;

class String {

private:

char\* buffer;

public:

String() : buffer(nullptr) {}

String(const char\* str) {

if (str) {

buffer = new char[strlen(str) + 1];

strcpy(buffer, str);

} else {

buffer = nullptr;

}

}

String(const String& other) {

if (other.buffer) {

buffer = new char[strlen(other.buffer) + 1];

strcpy(buffer, other.buffer);

} else {

buffer = nullptr;

}

}

~String() {

delete[] buffer;

}

String& operator=(const String& other) {

if (this != &other) {

delete[] buffer;

if (other.buffer) {

buffer = new char[strlen(other.buffer) + 1];

strcpy(buffer, other.buffer);

} else {

buffer = nullptr;

}

}

return \*this;

}

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

if (str.buffer) {

os << str.buffer;

}

return os;

}

};

int main() {

String s1("Hello, world!");

String s2 = "This is a custom String class.";

std::cout << "s1: " << s1 << std::endl;

std::cout << "s2: " << s2 << std::endl;

return 0;

}

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 <vector>

using namespace std;

class Polynomial {

private:

std::vector<std::pair<double, int>> terms;

public:

Polynomial() {}

void addTerm(double coeff, int exp) {

terms.push\_back(std::make\_pair(coeff, exp));

}

Polynomial operator+(const Polynomial& other) const {

Polynomial result;

for (auto& term : terms) {

result.addTerm(term.first, term.second);

}

for (auto& term : other.terms) {

result.addTerm(term.first, term.second);

}

result.simplify();

return result;

}

void simplify() {

std::vector<std::pair<double, int>> newTerms;

std::vector<int> processedExponents;

for (auto& term : terms) {

int exp = term.second;

if (std::find(processedExponents.begin(), processedExponents.end(), exp) == processedExponents.end()) {

double coeffSum = term.first;

for (auto& otherTerm : terms) {

if (otherTerm.second == exp && &otherTerm != &term) {

coeffSum += otherTerm.first;

}

}

newTerms.push\_back(std::make\_pair(coeffSum, exp));

processedExponents.push\_back(exp);

}

}

terms = newTerms;

}

void display() const {

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

if (i > 0) {

std::cout << " + ";

}

std::cout << terms[i].first << "x^" << terms[i].second;

}

std::cout << std::endl;

}

};

int main() {

Polynomial p1, p2, p3;

p1.addTerm(3.0, 2);

p1.addTerm(2.0, 1);

p1.addTerm(1.0, 0);

p2.addTerm(5.0, 3);

p2.addTerm(4.0, 2);

p2.addTerm(-2.0, 1);

p2.addTerm(7.0, 0);

std::cout << "p1: ";

p1.display();

std::cout << "p2: ";

p2.display();

p3 = p1 + p2;

std::cout << "p3 (p1 + p2): ";

p3.display();

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>

using namespace std;

template <typename T>

T min(T a, T b) {

return (a < b) ? a : b;

}

template <typename T>

T max(T a, T b) {

return (a > b) ? a : b;

}

int main() {

int minInt = min(3, 7);

int maxInt = max(3, 7);

std::cout << "Minimum of 3 and 7 is: " << minInt << std::endl;

std::cout << "Maximum of 3 and 7 is: " << maxInt << std::endl;

double minDouble = min(3.5, 7.2);

double maxDouble = max(3.5, 7.2);

std::cout << "Minimum of 3.5 and 7.2 is: " << minDouble << std::endl;

std::cout << "Maximum of 3.5 and 7.2 is: " << maxDouble << std::endl;

return 0;

}

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>

using namespace std;

template <typename T>

double average(const T\* array, int size) {

if (size == 0) {

std::cerr << "Error: Array size is zero." << std::endl;

return 0.0;

}

double sum = 0.0;

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

sum += static\_cast<double>(array[i]);

}

return sum / size;

}

template <typename T>

T minimum(const T\* array, int size) {

if (size == 0) {

std::cerr << "Error: Array size is zero." << std::endl;

}

T minVal = array[0];

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

if (array[i] < minVal) {

minVal = array[i];

}

}

return minVal;

}

template <typename T>

T maximum(const T\* array, int size) {

if (size == 0) {

std::cerr << "Error: Array size is zero." << std::endl;

return T();

}

T maxVal = array[0];

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

if (array[i] > maxVal) {

maxVal = array[i];

}

}

return maxVal;

}

int main() {

int intArray[] = {3, 7, 1, 5, 2};

int intSize = sizeof(intArray) / sizeof(int);

std::cout << "Integer Array: ";

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

std::cout << intArray[i] << " ";

}

std::cout << std::endl;

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

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

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

double doubleArray[] = {3.5, 7.2, 1.8, 5.9, 2.4};

int doubleSize = sizeof(doubleArray) / sizeof(double);

std::cout << "\nDouble Array: ";

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

std::cout << doubleArray[i] << " ";

}

std::cout << std::endl;

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

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

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

return 0;

}

Write overloaded functions formatString that can take a format string and different data types (e.g., int, double, string) to create formatted output strings.

#include <iostream>

#include <sstream>

#include <iomanip>

#include <string>

using namespace std;

template <typename T>

std::string formatString(const std::string& format, const T& value) {

std::ostringstream oss;

oss << std::setprecision(2) << std::fixed;

oss << std::setw(10) << std::left << format << ": " << value;

return oss.str();

}

template <>

std::string formatString<std::string>(const std::string& format, const std::string& value) {

std::ostringstream oss;

oss << std::setw(10) << std::left << format << ": " << value;

return oss.str();

}

int main() {

int intValue = 123;

double doubleValue = 456.789;

std::string stringValue = "Hello, world!";

std::cout << formatString("Int", intValue) << std::endl;

std::cout << formatString("Double", doubleValue) << std::endl;

std::cout << formatString("String", stringValue) << std::endl;

return 0;

}

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 <cmath>

using namespace std;

unsigned long long factorial(int n) {

if (n < 0) {

std::cerr << "Error: Factorial is defined only for non-negative integers." << std::endl;

return 0;

}

unsigned long long result = 1;

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

result \*= i;

}

return result;

}

template <typename T>

T power(T base, int exponent) {

return std::pow(base, exponent);

}

template <typename T>

T power(T base, double exponent) {

return std::pow(base, exponent);

}

int main() {

int n = 5;

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

int baseInt = 2;

double baseDouble = 2.5;

int exponentInt = 3;

double exponentDouble = 2.0;

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

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

return 0;

}

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.

Combined Concepts

#include <iostream>

#include <cmath>

// Base class Shape with an abstract method getArea

class Shape {

public:

// Pure virtual function (abstract method) to calculate area

virtual double getArea() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

// Constructor

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

// Implementation of getArea for Circle

virtual double getArea() const override {

return M\_PI \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

// Constructor

Rectangle(double w, double h) : width(w), height(h) {}

// Implementation of getArea for Rectangle

virtual double getArea() const override {

return width \* height;

}

};

// Derived class Square from Shape

class Square : public Shape {

private:

double side;

public:

// Constructor

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

// Implementation of getArea for Square

virtual double getArea() const override {

return side \* side;

}

};

int main() {

// Create instances of different shapes

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Square square(3.0);

// Polymorphic behavior: using Shape pointers to access getArea()

Shape\* shapes[] = { &circle, &rectangle, &square };

// Calculate and display areas

for (const auto& shape : shapes) {

std::cout << "Area: " << shape->getArea() << std::endl;

}

return 0;

}

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:

// Constructor

Item(const std::string& n, double p, int q)

: name(n), price(p), quantity(q) {}

// Overloaded stream insertion operator

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

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

return os;

}

};

int main() {

// Create an Item object

Item item1("Laptop", 1200.50, 3);

// Print item details using overloaded operator<<

std::cout << item1 << std::endl;

return 0;

}

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 <stdexcept> // For std::out\_of\_range exception

template <typename T>

class CustomList {

private:

T\* data; // Pointer to dynamically allocated array

int size; // Current size of the list

int capacity; // Capacity of the underlying array

public:

// Constructor

CustomList(int initialCapacity = 10) : size(0), capacity(initialCapacity) {

data = new T[capacity];

}

// Destructor

~CustomList() {

delete[] data;

}

// Copy constructor

CustomList(const CustomList& other) : size(other.size), capacity(other.capacity) {

data = new T[capacity];

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

data[i] = other.data[i];

}

}

// Assignment operator

CustomList& operator=(const CustomList& other) {

if (this != &other) {

delete[] data;

size = other.size;

capacity = other.capacity;

data = new T[capacity];

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

data[i] = other.data[i];

}

}

return \*this;

}

// Subscript operator overload for element access

T& operator[](int index) {

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

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

}

return data[index];

}

// Const version of subscript operator overload

const T& operator[](int index) const {

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

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

}

return data[index];

}

// Method to add an element to the list

void add(const T& element) {

if (size >= capacity) {

// Increase capacity if needed

int newCapacity = capacity \* 2;

T\* newData = new T[newCapacity];

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

newData[i] = data[i];

}

delete[] data;

data = newData;

capacity = newCapacity;

}

data[size++] = element;

}

// Method to get the current size of the list

int getSize() const {

return size;

}

};

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; // Raw pointer to the object

public:

// Constructor: Initialize with nullptr or a pointer to an object

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

// Destructor: Clean up allocated memory

~MySmartPtr() {

delete ptr;

}

// Overloaded dereference operator (\*)

T& operator\*() const {

return \*ptr;

}

// Overloaded arrow operator (->)

T\* operator->() const {

return ptr;

}

// Utility method to get the raw pointer

T\* get() const {

return ptr;

}

};

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>

#include <stdexcept> // for std::out\_of\_range

template <typename T>

class Vector {

private:

std::vector<T> elements;

public:

// Default constructor

Vector() {}

// Constructor with initializer list

Vector(std::initializer\_list<T> init) : elements(init) {}

// Destructor

~Vector() {}

// Overloaded operator+: Vector addition

Vector<T> operator+(const Vector<T>& other) const {

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

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

}

Vector<T> result;

result.elements.reserve(size());

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

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

}

return result;

}

// Overloaded operator-: Vector subtraction

Vector<T> operator-(const Vector<T>& other) const {

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

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

}

Vector<T> result;

result.elements.reserve(size());

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

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

}

return result;

}

// Overloaded operator[]: Access element by index

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

if (index >= size()) {

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

}

return elements[index];

}

// Const version of operator[] for const objects

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

if (index >= size()) {

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

}

return elements[index];

}

// Method to get the size of the vector

size\_t size() const {

return elements.size();

}

};

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> // for std::invalid\_argument

class Matrix {

private:

std::vector<std::vector<int>> mat;

size\_t rows;

size\_t cols;

public:

// Constructor to initialize matrix with dimensions and values

Matrix(size\_t m, size\_t n, const std::vector<std::vector<int>>& values)

: rows(m), cols(n), mat(values) {

if (values.size() != m || (values.size() > 0 && values[0].size() != n)) {

throw std::invalid\_argument("Matrix dimensions do not match provided values");

}

}

// Default constructor

Matrix() : rows(0), cols(0) {}

// Getter methods for rows and columns

size\_t getRows() const { return rows; }

size\_t getCols() const { return cols; }

// Overloaded addition operator (+)

Matrix operator+(const Matrix& other) const {

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

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

}

Matrix result(rows, cols, std::vector<std::vector<int>>(rows, std::vector<int>(cols, 0)));

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

for (size\_t j = 0; j < cols; ++j) {

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

}

}

return result;

}

// Overloaded subtraction operator (-)

Matrix operator-(const Matrix& other) const {

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

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

}

Matrix result(rows, cols, std::vector<std::vector<int>>(rows, std::vector<int>(cols, 0)));

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

for (size\_t j = 0; j < cols; ++j) {

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

}

}

return result;

}

// Overloaded multiplication operator (\*)

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

if (cols != other.rows) {

throw std::invalid\_argument("Number of columns in the first matrix must match number of rows in the second matrix for multiplication");

}

size\_t resultRows = rows;

size\_t resultCols = other.cols;

Matrix result(resultRows, resultCols, std::vector<std::vector<int>>(resultRows, std::vector<int>(resultCols, 0)));

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

for (size\_t j = 0; j < resultCols; ++j) {

for (size\_t k = 0; k < cols; ++k) {

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

}

}

}

return result;

}

// Overloaded output operator (<<) to print matrix

friend std::ostream& operator<<(std::ostream& os, const Matrix& matrix) {

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

for (size\_t j = 0; j < matrix.cols; ++j) {

os << matrix.mat[i][j] << ' ';

}

os << std::endl;

}

return os;

}

};