Operator Overloading

1. Rectangle Class: 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>


class Rectangle {
public:
    double width, height;
    Rectangle(double w = 0, double h = 0) : width(w), height(h) {}
    double area() const {
        return width * height;
    }
    Rectangle operator+(const Rectangle &other) {
        return Rectangle(0, (this->area() + other.area()));
    }
    friend std::ostream& operator<<(std::ostream &out, const Rectangle &rect) {
        out << "Rectangle(width: " << rect.width << ", height: " << rect.height << ")";
        return out;
    }
};
int main() {
    Rectangle r1(3, 4), r2(5, 6);
    Rectangle r3 = r1 + r2;
    std::cout << r3 << " with area: " << r3.area() << std::endl;
    return 0;
}
```

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

```
#include <iostream>

class Fraction {

public:

    int numerator, denominator;

    Fraction(int n = 0, int d = 1) : numerator(n), denominator(d) {

        if (denominator == 0) {

            throw std::invalid_argument("Denominator cannot be zero");

        } }

    Fraction operator+(const Fraction &other) {

        return Fraction(numerator * other.denominator + other.numerator * denominator, denominator *
other.denominator);

    }

    Fraction operator-(const Fraction &other) {

        return Fraction(numerator * other.denominator - other.numerator * denominator, denominator *
other.denominator);

    }

    Fraction operator*(const Fraction &other) {

        return Fraction(numerator * other.numerator, denominator * other.denominator);

    }

    Fraction operator/(const Fraction &other) {

        return Fraction(numerator * other.denominator, denominator * other.numerator);

    }

    friend std::ostream& operator<<(std::ostream &out, const Fraction &frac) {

        out << frac.numerator << "/" << frac.denominator;

        return out;
```

```
    }
};
int main() {
    Fraction f1(1, 2), f2(3, 4);
    std::cout << "Sum: " << f1 + f2 << std::endl;
    std::cout << "Difference: " << f1 - f2 << std::endl;
    std::cout << "Product: " << f1 * f2 << std::endl;
    std::cout << "Quotient: " << f1 / f2 << std::endl;
    return 0;
}
```

3. 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>
#include <unordered_map>

class Money {
public:
    double amount;
    std::string currency;

    Money(double amt = 0, const std::string &cur = "USD") : amount(amt), currency(cur) {}

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

    double toUSD() const {
```

```
        return amount / exchangeRates[currency];
    }

    bool operator==(const Money &other) const {
        return this->toUSD() == other.toUSD();
    }
    bool operator!=(const Money &other) const {
        return !(*this == other);
    }
    bool operator<(const Money &other) const {
        return this->toUSD() < other.toUSD();
    }
    bool operator<=(const Money &other) const {
        return this->toUSD() <= other.toUSD();
    }
    bool operator>(const Money &other) const {
        return this->toUSD() > other.toUSD();
    }
    bool operator>=(const Money &other) const {
        return this->toUSD() >= other.toUSD();
    }
    friend std::ostream& operator<<(std::ostream &out, const Money &money) {
        out << money.amount << " " << money.currency;
        return out;
    }
};
```

```
std::unordered_map<std::string, double> Money::exchangeRates = {
    {"USD", 1.0},
    {"EUR", 1.1},
    {"GBP", 1.3}
};
int main() {
    Money m1(100, "USD"), m2(100, "EUR");
    std::cout << (m1 == m2) << std::endl;
    std::cout << (m1 < m2) << std::endl;
    return 0;
}
```

4. 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 {
public:
    std::string str;
    String(const std::string &s = "") : str(s) {}
    friend std::ostream& operator<<(std::ostream &out, const String &s) {
        out << s.str;
        return out;
    } };
int main() {
    String s("Hello, World!");
    std::cout << s << std::endl;
    return 0;
```

}

5. 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 <vector>
#include <algorithm>
class Polynomial {
public:
    std::vector<std::pair<int, int>> terms; // (coefficient, exponent)
    Polynomial() {}
    void addTerm(int coeff, int exp) {
        terms.push_back({coeff, exp});
    }
    Polynomial operator+(const Polynomial &other) {
        Polynomial result;
        result.terms = this->terms;

        for (const auto &term : other.terms) {
            bool found = false;
            for (auto &rterm : result.terms) {
                if (rterm.second == term.second) {
                    rterm.first += term.first;
                    found = true;
                    break;
                }
            }
```

```
            if (!found) {
                result.terms.push_back(term);
            }
        }
        return result;
    }
    friend std::ostream& operator<<(std::ostream &out, const Polynomial &poly) {
        for (const auto &term : poly.terms) {
            out << term.first << "x^" << term.second << " ";
        }
        return out;
    }
};
int main() {
    Polynomial p1, p2;
    p1.addTerm(3, 2);
    p1.addTerm(4, 1);
    p2.addTerm(1, 2);
    p2.addTerm(2, 0);
    Polynomial p3 = p1 + p2;
    std::cout << p3 << std::endl;
    return 0;
}
```

Function Overloading

6. 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>
```

```
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() {
    std::cout << "Min(3, 5): " << min(3, 5) << std::endl;
    std::cout << "Max(3.5, 2.1): " << max(3.5, 2.1) << std::endl;
    return 0;
}
```

7. 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 <algorithm>
#include <numeric>
template <typename T>
T average(const T arr[], int size) {
    return std::accumulate(arr, arr + size, T(0)) / size;
}
template <typename T>
T minimum(const T arr[], int size) {
    return *std::min_element(arr, arr + size);
}
template <typename T>
```

```
T maximum(const T arr[], int size) {
    return *std::max_element(arr, arr + size);
}
int main() {
    int intArr[] = {1, 2, 3, 4, 5};
    double doubleArr[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    std::cout << "Average(int): " << average(intArr, 5) << std::endl;
    std::cout << "Minimum(int): " << minimum(intArr, 5) << std::endl;
    std::cout << "Maximum(int): " << maximum(intArr, 5) << std::endl;
    std::cout << "Average(double): " << average(doubleArr, 5) << std::endl;
    std::cout << "Minimum(double): " << minimum(doubleArr, 5) << std::endl;
    std::cout << "Maximum(double): " << maximum(doubleArr, 5) << std::endl;

    return 0;
}
```

8. String Formatting: 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 <string>
std::string formatString(const std::string &fmt, int value) {
    std::ostringstream oss;
    oss << fmt << value;
    return oss.str();
}
std::string formatString(const std::string &fmt, double value) {
    std::ostringstream oss;
```

```
    oss << fmt << value;
    return oss.str();
}
std::string formatString(const std::string &fmt, const std::string &value) {
    std::ostringstream oss;
    oss << fmt << value;
    return oss.str();
}
int main() {
    std::cout << formatString("Value: ", 42) << std::endl;
    std::cout << formatString("Value: ", 3.14) << std::endl;
    std::cout << formatString("Value: ", "Hello") << std::endl
```

9. 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 <cmath>

// Factorial function for integer inputs
int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
// Factorial function for floating-point inputs (gamma function approximation)
double factorial(double n) {
    return std::tgamma(n + 1);
}
// Power function for integer inputs
int power(int base, int exp) {
```

```
    return std::pow(base, exp);
}
// Power function for floating-point inputs
double power(double base, double exp) {
    return std::pow(base, exp);
}
int main() {
    std::cout << "Factorial(5): " << factorial(5) << std::endl;
    std::cout << "Factorial(5.5): " << factorial(5.5) << std::endl;
    std::cout << "Power(2, 3): " << power(2, 3) << std::endl;
    std::cout << "Power(2.5, 3.5): " << power(2.5, 3.5) << std::endl;
    return 0;
}
```

10. 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. Combined Concepts.

```
#include <iostream>
#include <cmath>

class Shape {
public:
    virtual double getArea() const = 0; // Pure virtual function
    virtual ~Shape() {}
};

class Circle : public Shape {
    double radius;
```

```
public:
    Circle(double r) : radius(r) {}
    double getArea() const override {
        return M_PI * radius * radius;
    }
};
class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() const override {
        return width * height;
    }
};
class Square : public Shape {
    double side;
public:
    Square(double s) : side(s) {}
    double getArea() const override {
        return side * side;
    }
};
int main() {
    Shape *shapes[] = {
        new Circle(5),
        new Rectangle(4, 6),
```

```
        new Square(3)
    };
    for (Shape *shape : shapes) {
        std::cout << "Area: " << shape->getArea() << std::endl;
        delete shape;
    }
    return 0;
}
```

11. 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 {
public:
    std::string name;
    double price;
    int quantity;
    Item(const std::string &n, double p, int q) : name(n), price(p), quantity(q) {}
    friend std::ostream& operator<<(std::ostream &out, const Item &item) {
        out << "Item(name: " << item.name << ", price: " << item.price << ", quantity: " << item.quantity
<< ")";
        return out;
    }
};
int main() {
    Item item("Laptop", 999.99, 10);
    std::cout << item << std::endl;
```

```
    return 0;
}
```

12. 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 {
    std::vector<T> data;
public:
    void add(const T &value) {
        data.push_back(value);
    }
    T& operator[](std::size_t index) {
        if (index >= data.size()) {
            throw std::out_of_range("Index out of bounds");
        }
        return data[index];
    }
    std::size_t size() const {
        return data.size();
    }
};
int main() {
    CustomList<int> list;
    list.add(10);
```

```
    list.add(20);
    try {
        std::cout << list[0] << std::endl;
        std::cout << list[2] << std::endl; // This will throw an exception
    } catch (const std::out_of_range &e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

13. 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 {
    T *ptr;
public:
    explicit MySmartPtr(T *p = nullptr) : ptr(p) {}
    ~MySmartPtr() { delete ptr; }
    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
};
class Test {
public:
    void show() { std::cout << "Test::show()" << std::endl; }
};
```

```
int main() {
    MySmartPtr<Test> ptr(new Test());
    ptr->show();
    return 0;
}
```

14. 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 {
    std::vector<T> data;
public:
    void add(const T &value) {
        data.push_back(value);
    }
    T& operator[](std::size_t index) {
        return data[index];
    }
    Vector operator+(const Vector &other) {
        Vector result;
        for (std::size_t i = 0; i < data.size(); ++i) {
            result.add(data[i] + other.data[i]);
        }
        return result;
    }
    Vector operator-(const Vector &other) {
```

```
        Vector result;
        for (std::size_t i = 0; i < data.size(); ++i) {
            result.add(data[i] - other.data[i]);
        }
        return result;
    }

    std::size_t size() const {
        return data.size();
    }
};
int main() {
    Vector<int> v1, v2;
    v1.add(1);
    v1.add(2);
    v2.add(3);
    v2.add(4);
    Vector<int> v3 = v1 + v2;
    std::cout << "v3[0]: " << v3[0] << ", v3[1]: " << v3[1] << std::endl;
    return 0;
}
```

15. 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 {
    std::vector<std::vector<int>> data;
    std::size_t rows, cols;
public:
    Matrix(std::size_t r, std::size_t c) : rows(r), cols(c) {
        data.resize(rows, std::vector<int>(cols, 0));
    }

    std::vector<int>& operator[](std::size_t index) {
        return data[index];
    }
    const std::vector<int>& operator[](std::size_t index) const {
        return data[index];
    }
    Matrix operator+(const Matrix &other) {
        if (rows != other.rows || cols != other.cols) {
            throw std::invalid_argument("Matrix dimensions must match for addition");
        }
        Matrix result(rows, cols);
        for (std::size_t i = 0; i < rows; ++i) {
            for (std::size_t j = 0; j < cols; ++j) {
                result[i][j] = data[i][j] + other[i][j];
            }
        }
        return result;
```

```
}
Matrix operator-(const Matrix &other) {
    if (rows != other.rows || cols != other.cols) {
        throw std::invalid_argument("Matrix dimensions must match for subtraction");
    }
    Matrix result(rows, cols);
    for (std::size_t i = 0; i < rows; ++i) {
        for (std::size_t j = 0; j < cols; ++j) {
            result[i][j] = data[i][j] - other[i][j];
        }
    }
    return result;
}
Matrix operator*(const Matrix &other) {
    if (cols != other.rows) {
        throw std::invalid_argument("Matrix dimensions must match for multiplication");
    }
    Matrix result(rows, other.cols);
    for (std::size_t i = 0; i < rows; ++i) {
        for (std::size_t j = 0; j < other.cols; ++j) {
            for (std::size_t k = 0; k < cols; ++k) {
                result[i][j] += data[i][k] * other[k][j];
            }
        }
    }
    return result;
```

```
    }
    friend std::ostream& operator<<(std::ostream &out, const Matrix &matrix) {
        for (std::size_t i = 0; i < matrix.rows; ++i) {
            for (std::size_t j = 0; j < matrix.cols; ++j) {
                out << matrix[i][j] << " ";
            }
            out << std::endl;
        }
        return out;
    }
};
int main() {
    Matrix m1(2, 2), m2(2, 2);
    m1[0][0] = 1; m1[0][1] = 2;
    m1[1][0] = 3; m1[1][1] = 4;
    m2[0][0] = 5; m2[0][1] = 6;
    m2[1][0] = 7; m2[1][1] = 8;
    Matrix m3 = m1 + m2;
    Matrix m4 = m1 - m2;
    Matrix m5 = m1 * m2;
    std::cout << "Matrix m3 (Addition):\n" << m3;
    std::cout << "Matrix m4 (Subtraction):\n" << m4;
    std::cout << "Matrix m5 (Multiplication):\n" << m5;

    return 0;
}
```