Sunny kumar

Operator overloading

Q.What are the benefits and drawbacks of operator overloading?

Ans: benefits of operator overloading:-

1. Natural syntax(making code more readable)
2. Easy to use (due to class and primitive data types)
3. Improving maintainability

Drawback of operator overloading:

1. complex to understand
2. If we not handle properly it would be ambiguity
3. Difficult to maintain
4. Operator overloading can hide the actual operations being performed.

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

Ans: Yes we can overload the assignment operator in C++.

ensure proper behavior:

1. Avoid unnecessary operations, make it as simple as you can.
2. Ensure that each object has its own copy of data while passing the value to the constructor.

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

Ans:difference between member function and non member function:

Member function overloading:

1. Member functions are defined in the class.
2. It is a member of class.
3. They are accessed using object instances.

Here is the example of member function:

```
class test
{
   private:
   int num;     //
   public:
   test(): num(8){}
   void operator ++(){
   num=num+2;
   }
   void print()
   {
      cout<<"the count is :"<< num<<endl;
   }
};
```

Non Member function overloading:

1. Non member function defined outside the class definition but declared as friends inside the class to access private members.
2. They are called using normal function.

Here is the example of Member function overloading:

```
class myclass{
   public:
   friend myclass operator+( myclass& a, myclass& b);
};
myclass operator+(myclass& a, myclass& b)
{
   // you can write your statement here
}
```

Q. What is the core concept behind function overloading?
Ans: The main core concept behind the function overloading is that it allow a feature where we can use a function with same name but the parameter must be different (bcz of ambiguity to compiler).

Function overloading improved readability of code.

Here is the example of function overloading:

```
#include <iostream>
using namespace std;
class Cal {
public:
   static int add(int a, int b) {
       return a + b;
   }

   static int add(int a, int b, int c) {
       return a + b + c;
   }
};
int main(void) {
   Cal C;
   cout << Cal::add(10, 20) << endl;
   cout << Cal::add(12, 20, 23) << endl;
   return 0;
}
```

Q. 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:
    Vector2D(double x = 0, double y = 0) : x(x), y(y) {}
    Vector2D operator+(const Vector2D& other)  {
        return Vector2D(x + other.x, y + other.y);
    }
    Vector2D operator-(const Vector2D& other)  {
        return Vector2D(x - other.x, y - other.y);
    }
    Vector2D operator*(double scalar)  {
        return Vector2D(x * scalar, y * scalar);
    }
    Vector2D operator/(double scalar)  {
        return Vector2D(x / scalar, y / scalar);
    }
    friend Vector2D operator*(double scalar, const Vector2D& vector) {
        return Vector2D(vector.x * scalar, vector.y * scalar);
    }

    friend ostream& operator<<(ostream& os, const Vector2D& vector) {
        os << "(" << vector.x << ", " << vector.y << ")";
        return os;
    }
};

int main() {
    Vector2D v1(1.0, 2.0);
    Vector2D v2(3.0, 4.0);

    Vector2D v3 = v1 + v2;
    Vector2D v4 = v1 - v2;
    Vector2D v5 = v1 * 2.0;
    Vector2D v6 = v1 / 2.0;
    Vector2D v7 = 2.0 * v2;

    cout << "v1 + v2 = " << v3 << endl;
    cout << "v1 - v2 = " << v4 << endl;
    cout << "v1 * 2.0 = " << v5 << endl;
    cout << "v1 / 2.0 = " << v6 << endl;
    cout << "2.0 * v2 = " << v7 << endl;

    return 0;
}
```

Q.Is it possible to overload the comparison operators (==, !=, <, >, <=, >=) for custom classes? If so, what considerations should be taken into account?
ans:
Yes, it is possible to overload comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) for custom classes in C++.

Considerations:-
1. Use of Const
2. Efficiency
3. Consistency
4. Operators uses safely

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

ANS: The compiler differentiates overloaded functions by their signatures, which include the function name and parameter types. Name mangling ensures that each function has a unique identifier internally. When calling an overloaded function, the compiler matches the provided arguments to the appropriate function based on the best match.

Q. Can functions with different return types be overloaded? Explain your reasoning.
ANS: In C++, functions cannot be overloaded solely based on their return types. The function signature used by the compiler for overloading resolution includes the function name and parameter types, but not the return type

1. **Ambiguity in Function Calls:** When a function is called, the return type is not part of the information provided to resolve which function to call. Only the function name and argument types are considered. Therefore, if two functions differ only in their return type, the compiler would not be able to decide which function to call based on a function call alone.
   For ex.
   int function(int a);
   double function(int a);

2. **Declaration and Definition:** When declaring or defining a function, the return type is part of the function's signature, but overloading resolution happens based on the function call, where only the function name and arguments are visible. The return type does not part in this implement process.

Q.Design a function printValue that can handle different data types (e.g., int, double, std::string) by overloading it with appropriate parameter lists.

Ans:

```cpp
#include <iostream>
#include <string>
using namespace std;

// Overload for int
void printValue(int value) {
   cout << "Integer: " << value << endl;
}

// Overload for double
void printValue(double value) {
   cout << "Double: " << value << endl;
}

// Overload for string
void printValue(const string& value) {
   cout << "String: " << value << endl;
}

// Overload for const char* (C-string)
void printValue(const char* value) {
   cout << "C-String: " << value << endl;
}

int main() {
   int intValue = 42;
   double doubleValue = 3.14;
   string stringValue = "sunny";
   const char* cStringValue = "C-Style String";

   printValue(intValue);
   printValue(doubleValue);
   printValue(stringValue);
   printValue(cStringValue);

   return 0;
}
```

Q. Discuss the advantages and disadvantages of using default arguments in overloaded functions.
Ans:

advantages of using default arguments in overloaded functions:

1. Simplifies Function Calls
2. Reduces Code Duplication
3. Improves Function Interface

disadvantages of using default arguments in overloaded functions:

1. Ambiguity with Overloaded Functions
2. Maintenance Complexity
3. Readability Issues
4. Order Dependency

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

Ans: Using separate functions with descriptive names instead of overloading a single function might be a better idea in several scenarios.

1. Clarity and Readability
2. Avoiding Ambiguity
3. Simplifying Function Interfaces
4. Easy Maintenance

Q.Can function overloading be used to achieve polymorphism (the ability to treat objects of different derived classes in a similar way)? Explain.
Ans: Function overloading itself does not achieve polymorphism in the object-oriented programming sense. Instead, polymorphism is typically achieved through the use of inheritance and virtual functions. However, function overloading and polymorphism can be used together in certain contexts to provide flexible and extensible designs.

Q.Describe a scenario where overloading a function with a variable number of arguments (varargs) could be beneficial.
Ans: Overloading a function with a variable number of arguments (varargs) can be beneficial in scenarios where the function needs to handle a varying number of parameters of the same type or related types. This approach provides flexibility in function invocation, allowing it to accommodate different numbers of arguments based on the specific needs of the caller.

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

Function overloading and virtual functions in C++ inheritance are two distinct mechanisms that serve different purposes and are suited for different use cases.

Function Overloading

- Purpose: Function overloading allows multiple functions with the same name but different parameter lists
- Use case: Providing multiple ways to interact with a function based on different argument types or numbers.
- Handling default parameter values to simplify function calls.
- Resolves function calls at compile time, leading to potentially faster execution.
- Simplicity: Easy to understand and implement

virtual functions

- Purpose**:** Virtual functions enable polymorphism, allowing functions to be overridden in derived classes to provide different implementations.
- Implementing interfaces and abstract classes.

- Providing a common interface for different derived classes while allowing each class to define its specific behavior.
- Facilitating code reuse and extensibility through inheritance.
- Supports adding new derived classes without modifying existing code.

Choosing Between Function Overloading and Virtual Functions

- **Function Overloading** is suitable when you need to provide multiple behaviors for a function based on different parameter types or numbers at compile time. It is straightforward and efficient for static scenarios where the behavior does not change dynamically.
- **Virtual Functions** are more suitable when you require polymorphic behavior, where the appropriate function to call is determined at runtime based on the actual object type. This approach supports extensibility, flexibility, and the ability to work with objects of different derived classes through a common interface.

Q.When overloading operators, what are some best practices tensure code clarity and maintainability?

Ans: operators are overloaded with clear semantics that align with typical usage.Member functions are marked const where appropriate to ensure const-correctness.Friend functions are used for non-member overloads to maintain symmetry and clarity.

Q.Complex Numbers (C++) - Define a class Complex to represent complex numbers with member variables for real and imaginary parts.

Ans:

```
/*Complex Numbers (C++) - Define a class Complex to represent complex
     numbers

with member variables for real and imaginary parts. */
```

```cpp
#include <iostream>

using namespace std;


class Complex {

private:

   double real;

   double imag;


public:

   Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}


   double getReal() const { return real; }

   double getImaginary() const { return imag; }


   void setReal(double r) { real = r; }

   void setImaginary(double i) { imag = i; }


   Complex operator+(const Complex& other) const {

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

   }


   Complex operator-(const Complex& other) const {

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

```cpp
    }

    Complex operator*(const Complex& other) const {
        return Complex(real * other.real - imag * other.imag,
                       real * other.imag + imag * other.real);
    }

    Complex operator/(const Complex& other) const {
        double denom = other.real * other.real + other.imag * other.imag;
        return Complex((real * other.real + imag * other.imag) / denom,
                       (imag * other.real - real * other.imag) / denom);
    }

    friend ostream& operator<<(ostream& os, const Complex& complex) {
        os << complex.real;
        if (complex.imag >= 0)
            os << " + " << complex.imag << "i";
        else
            os << " - " << -complex.imag << "i";
        return os;
    }
};
int main() {
    Complex c1(2.5, 3.5);

    Complex c2(1.2, -0.7);


    Complex sum = c1 + c2;

    Complex difference = c1 - c2;

    Complex product = c1 * c2;

    Complex quotient = c1 / c2;


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

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

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

    cout << "Difference: " << difference << endl;

    cout << "Product: " << product << endl;

    cout << "Quotient: " << quotient << endl;


    return 0;

}
```

Q.Overload the +, -, and * operators for complex number addition, subtraction, and multiplication.

```
/*Overload the +, -, and * operators for complex number addition,
      subtraction, and multiplication.*/
```

```cpp
#include <iostream>

using namespace std;


class Complex {

private:

    double real;

    double imag;


public:

    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}


    double getReal() const { return real; }

    double getImaginary() const { return imag; }


    void setReal(double r) { real = r; }

    void setImaginary(double i) { imag = i; }




    Complex operator+(const Complex& other) const {

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

    }


    Complex operator-(const Complex& other) const {
        return Complex(real - other.real, imag - other.imag);

    }



    Complex operator*(const Complex& other) const {

        return Complex(real * other.real - imag * other.imag,

                       real * other.imag + imag * other.real);

    }



    friend ostream& operator<<(ostream& os, const Complex& complex) {

        os << complex.real;

        if (complex.imag >= 0)

            os << " + " << complex.imag << "i";

        else

            os << " - " << -complex.imag << "i";

        return os;

    }

};

int main() {

    Complex c1(2.5, 3.5);

    Complex c2(1.2, -0.7);



    Complex sum = c1 + c2;

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

```
   Complex difference = c1 - c2;

   cout << "Difference: " << difference << endl;


   Complex product = c1 * c2;

   cout << "Product: " << product << endl;


   return 0;

}
```

Q.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>

using namespace std;


class Point2D {

private:

   double x;

   double y;


public:

   Point2D(double x = 0.0, double y = 0.0) : x(x), y(y) {}
```

```cpp
    double getX() const { return x; }
    double getY() const { return y; }

    Point2D operator+(const Point2D& other) const {
        return Point2D(x + other.x, y + other.y);
    }

    friend ostream& operator<<(ostream& os, const Point2D& point) {
        os << "(" << point.x << ", " << point.y << ")";
        return os;
    }
};
int main() {
    Point2D p1(2.5, 3.5);
    Point2D p2(1.2, -0.7);

    Point2D sum = p1 + p2;
    cout << "Sum: " << sum << endl;

    return 0;
}
```

Q.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>

using namespace std;

class Time {

private:

   int hours;

   int minutes;

   int seconds;


public:

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


   int getHours() const { return hours; }

   int getMinutes() const { return minutes; }

   int getSeconds() const { return seconds; }


   Time operator+(const Time& other) const {

       int totalSeconds = seconds + other.seconds;

       int additionalMinutes = totalSeconds / 60;

       totalSeconds %= 60;
```

```
        int totalMinutes = minutes + other.minutes + additionalMinutes;

        int additionalHours = totalMinutes / 60;

        totalMinutes %= 60;



        int totalHours = hours + other.hours + additionalHours;



        return Time(totalHours, totalMinutes, totalSeconds);

    }



    friend ostream& operator<<(ostream& os, const Time& time) {

        os << time.hours << " hours, " << time.minutes << " minutes, " <<
time.seconds << " seconds";

        return os;

    }

};

int main() {

    Time t1(1, 30, 45);

    Time t2(2, 15, 20);

    Time sum = t1 + t2;

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



    return 0;

}
```

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

```
#include <iostream>

using namespace std;

class Date {

private:

   int year;

   int month;

   int day;


public:

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

   int getYear() const { return year; }

   int getMonth() const { return month; }

   int getDay() const { return day; }

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

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

   }



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

       return !(*this == other);

   }

   friend ostream& operator<<(ostream& os, const Date& date) {
```

```cpp
        os << date.year << "-" << date.month << "-" << date.day;

        return os;

    }

};

int main() {

   Date d1(2023, 6, 27);

   Date d2(2023, 6, 27);

   Date d3(2023, 6, 28);

   if (d1 == d2) {

       cout << d1 << " is equal to " << d2 << endl;

   } else {

       cout << d1 << " is not equal to " << d2 << endl;

   }

   if (d1 != d3) {

       cout << d1 << " is not equal to " << d3 << endl;

   } else {

       cout << d1 << " is equal to " << d3 << endl;

   }


   return 0;

}
```

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

```
#include <iostream>

#include <cstring>

using namespace std;

class String {

private:

   char* buffer;

   int length;


public:

   String(const char* str = "") {

       length = strlen(str);

       buffer = new char[length + 1];

       strcpy(buffer, str);

   }


   ~String() {

       delete[] buffer;

   }

   String(const String& other) : length(other.length) {

       buffer = new char[length + 1];

       strcpy(buffer, other.buffer);
```

```
    }

    String& operator=(const String& other) {

        if (this != &other) {

            delete[] buffer;

            length = other.length;

            buffer = new char[length + 1];

            strcpy(buffer, other.buffer);

        }

        return *this;

    }

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

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

    }

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

        os << str.buffer;

        return os;

    }

};

int main() {

    String s1("sunny ");

    String s2("kumar");

    String s3("sharma");
```

```
   if (s1 == s2) {

       cout << s1 << " is equal to " << s2 << endl;

   } else {

       cout << s1 << " is not equal to " << s2 << endl;

   }


   if (s1 == s3) {

       cout << s1 << " is equal to " << s3 << endl;

   } else {

       cout << s1 << " is not equal to " << s3 << endl;

   }


   return 0;

}
```

Q.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>

using namespace std;

double calculateArea(double width, double height) {

   return width * height;
}

double calculateArea(double radius) {

   return M_PI * radius * radius;

}



int main() {

   double rectWidth = 5.0;

   double rectHeight = 3.0;

   double rectArea = calculateArea(rectWidth, rectHeight);

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

   double circleRadius = 2.5;

   double circleArea = calculateArea(circleRadius);

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



   return 0;

}
```

Q. 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>
```

```
using namespace std;

const double METER_TO_FEET = 3.28084;

const double FEET_TO_METER = 1 / METER_TO_FEET;

const double CELSIUS_TO_FAHRENHEIT = 9.0 / 5.0;

const double FAHRENHEIT_TO_CELSIUS = 1.0 / CELSIUS_TO_FAHRENHEIT;


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

   if (toUnit == "feet")

       return meters * METER_TO_FEET;

   else if (toUnit == "meters")

       return meters;

   else {

       cerr << "nhi hoga " << toUnit << endl;

       return 0.0;

   }

}


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

   if (toUnit == "meters")

       return feet * FEET_TO_METER;

   else if (toUnit == "feet")

       return feet;

   else {
```

```cpp
        cerr << "nhi hoga " << toUnit << endl;

        return 0.0;

    }

}


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

    if (toUnit == "fahrenheit")

        return celsius * CELSIUS_TO_FAHRENHEIT + 32.0;

    else if (toUnit == "celsius")

        return celsius;

    else {

        cerr << "nhi hoga " << toUnit << endl;

        return 0.0;

    }

}


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

    if (toUnit == "celsius")

        return (fahrenheit - 32.0) * FAHRENHEIT_TO_CELSIUS;

    else if (toUnit == "fahrenheit")

        return fahrenheit;

    else {

        cerr << "nhi hoga " << toUnit << endl;
        return 0.0;

    }

}



int main() {

    double meters = 10.0;

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



    int feet = 20;

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



    double celsius = 25.0;

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



    int fahrenheit = 68;

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



    return 0;

}
```

Q. 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 <climits>

#include <cfloat>

using namespace std;

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

   if (size == 0)

       return 0.0;


   int sum = 0;

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

       sum += arr[i];

   }

   return static_cast<double>(sum) / size;

}

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

   if (size == 0)

       return 0.0;


   double sum = 0.0;

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

       sum += arr[i];
```

```
    }

    return sum / size;

}

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

    if (size == 0)

        return INT_MIN;


    int minVal = arr[0];

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

        if (arr[i] < minVal) {

            minVal = arr[i];

        }

    }

    return minVal;

}

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

    if (size == 0)

        return DBL_MIN;


    double minVal = arr[0];

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

        if (arr[i] < minVal) {

            minVal = arr[i];
        }

    }

    return minVal;

}

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

    if (size == 0)

        return INT_MAX;


    int maxVal = arr[0];

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

        if (arr[i] > maxVal) {

            maxVal = arr[i];

        }

    }

    return maxVal;

}

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

    if (size == 0)

        return DBL_MAX;


    double maxVal = arr[0];

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

        if (arr[i] > maxVal) {
            maxVal = arr[i];

        }

    }

    return maxVal;

}



int main() {

    int intArr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};

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



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

    cout << "Minimum of integers: " << minimum(intArr, intSize) << endl;

    cout << "Maximum of integers: " << maximum(intArr, intSize) << endl;

    double doubleArr[] = {3.1, 1.5, 4.2, 1.8, 5.9, 2.4, 6.7, 5.2};

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



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

    cout << "Minimum of doubles: " << minimum(doubleArr, doubleSize) <<
endl;

    cout << "Maximum of doubles: " << maximum(doubleArr, doubleSize) <<
endl;



    return 0;
```

```
}
```

Q.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>

using namespace std;

unsigned long long factorial(int n) {

   if (n < 0) {

       cerr << "Factorial is not defined for negative numbers." << endl;

       return 0;

   }

   unsigned long long result = 1;

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

       result *= i;

   }

   return result;

}

double factorial(double x) {

   if (x < 0) {

       cerr << "Factorial is not defined for negative numbers." << endl;

       return 0.0;
```

```
    }

    return tgamma(x + 1);

}

long long power(int base, int exponent) {

    return static_cast<long long>(pow(base, exponent));

}

double power(double base, double exponent) {

    return pow(base, exponent);

}


int main() {

    int n = 5;

    double x = 4.5;


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

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

    int baseInt = 2;

    double baseDouble = 2.5;

    int expInt = 3;

    double expDouble = 1.5;


    cout << baseInt << " raised to the power " << expInt << " (integer): "
<< power(baseInt, expInt) << endl;
```

```
   cout << baseDouble << " raised to the power " << expDouble << "
(double): " << power(baseDouble, expDouble) << endl;


   return 0;

}
```

Q. 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>

using namespace std;

struct Term {

   double coefficient;

   int exponent;


   Term(double coeff = 0.0, int exp = 0)

       : coefficient(coeff), exponent(exp) {}

};


class Polynomial {

private:
   vector<Term> terms;


public:

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


   Polynomial operator+(const Polynomial& other) const {

       vector<Term> resultTerms;

       int i = 0, j = 0;

               while (i < terms.size() && j < other.terms.size()) {

           if (terms[i].exponent > other.terms[j].exponent) {

               resultTerms.push_back(terms[i++]);

           } else if (terms[i].exponent < other.terms[j].exponent) {

               resultTerms.push_back(other.terms[j++]);

           } else {

               double sumCoeff = terms[i].coefficient +
other.terms[j].coefficient;

               if (sumCoeff != 0.0) {

                   resultTerms.push_back(Term(sumCoeff,
terms[i].exponent));

               }

               i++;

               j++;

           }

       }
```

```cpp
    while (i < terms.size()) {
        resultTerms.push_back(terms[i++]);
    }
    while (j < other.terms.size()) {
        resultTerms.push_back(other.terms[j++]);
    }

    return Polynomial(resultTerms);
}

void print() const {
    bool firstTerm = true;
    for (const auto& term : terms) {
        if (!firstTerm && term.coefficient > 0) {
            cout << " + ";
        } else if (!firstTerm && term.coefficient < 0) {
            cout << " - ";
        }
        if (term.coefficient != 1.0 && term.coefficient != -1.0) {
            cout << term.coefficient;
        } else if (term.coefficient == -1.0) {
            cout << "-";
            }
            if (term.exponent != 0) {
                cout << "x";
                if (term.exponent != 1) {
                    cout << "^" << term.exponent;
                }
            }
            firstTerm = false;
        }
        cout << endl;
    }
};

int main() {
    Polynomial poly1({Term(3.0, 2), Term(-5.0, 1), Term(2.0, 0)});
    Polynomial poly2({Term(4.0, 3), Term(2.0, 1), Term(1.0, 0)});

    Polynomial sum = poly1 + poly2;

    cout << "Polynomial 1: ";
    poly1.print();
    cout << "Polynomial 2: ";
    poly2.print();
    cout << "Sum: ";

    sum.print();



    return 0;

}
```

Q.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>

using namespace std;

template<typename T>

class Vector {

private:

    vector<T> elements;



public:

    Vector(initializer_list<T> init) : elements(init) {}



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

        Vector<T> result = *this;

        for (size_t i = 0; i < elements.size(); ++i) {
            result[i] += other[i];

        }

        return result;

    }



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

        Vector<T> result = *this;

        for (size_t i = 0; i < elements.size(); ++i) {

            result[i] -= other[i];

        }

        return result;

    }

    T& operator[](size_t index) {

        return elements[index];

    }

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

        return elements[index];

    }



    void display() const {

        cout << "[ ";

        for (const auto& elem : elements) {

            cout << elem << " ";
        }

        cout << "]" << endl;

    }

};


int main() {

    Vector<int> v1 = {1, 2, 3, 4, 5};

    Vector<int> v2 = {2, 4, 6, 8, 10};


    Vector<int> sum = v1 + v2;

    Vector<int> diff = v1 - v2;


    cout << "v1: ";

    v1.display();

    cout << "v2: ";

    v2.display();

    cout << "Sum: ";

    sum.display();

    cout << "Difference: ";

    diff.display();


    return 0;

}
```

Q.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>

using namespace std;

template<typename T>

class Matrix {

private:

   vector<vector<T>> data;

   size_t rows;

   size_t cols;


public:

   Matrix(size_t rows, size_t cols, const T& defaultValue = T())

       : rows(rows), cols(cols), data(vector<vector<T>>(rows,
vector<T>(cols, defaultValue))) {}


   size_t numRows() const {

       return rows;

   }
   size_t numCols() const {

       return cols;

   }



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

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

           throw invalid_argument("Matrix dimensions must be the
same for addition.");

       }



       Matrix<T> result(rows, cols);

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

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

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

           }

       }

       return result;

   }



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

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

           throw invalid_argument("Matrix dimensions must be the
same for subtraction.");

       }
```

```
        Matrix<T> result(rows, cols);
        for (size_t i = 0; i < rows; ++i) {
            for (size_t j = 0; j < cols; ++j) {
                result.data[i][j] = data[i][j] - other.data[i][j];
            }
        }
        return result;
    }

    Matrix<T> operator*(const Matrix<T>& other) const {
        if (cols != other.rows) {
            throw invalid_argument("Number of columns in first matrix
must match number of rows in second matrix for multiplication.");
        }

        Matrix<T> result(rows, other.cols);
        for (size_t i = 0; i < rows; ++i) {
            for (size_t j = 0; j < other.cols; ++j) {
                result.data[i][j] = 0;
                for (size_t k = 0; k < cols; ++k) {
                    result.data[i][j] += data[i][k] *
other.data[k][j];
                }
            }
        }
        return result;
    }

    vector<T>& operator[](size_t index) {
        return data[index];
    }

    const vector<T>& operator[](size_t index) const {
        return data[index];
    }

    void display() const {
        for (size_t i = 0; i < rows; ++i) {
            for (size_t j = 0; j < cols; ++j) {
                cout << data[i][j] << " ";
            }
            cout << endl;
        }
    }
};
```

```cpp
int main() {

   Matrix<int> m1(2, 3, 1);

   Matrix<int> m2(2, 3, 2);



   Matrix<int> sum = m1 + m2;

   Matrix<int> diff = m1 - m2;



   cout << "Matrix 1:" << endl;

   m1.display();

   cout << "Matrix 2:" << endl;

   m2.display();

   cout << "Sum:" << endl;

   sum.display();

   cout << "Difference:" << endl;

   diff.display();



   Matrix<int> m3(2, 3, 1);

   Matrix<int> m4(3, 2, 2);



   Matrix<int> product = m3 * m4;



   cout << "Matrix 3:" << endl;

   m3.display();
```

```
    cout << "Matrix 4:" << endl;

    m4.display();

    cout << "Product:" << endl;

    product.display();



    return 0;

}
```