

Object Oriented Programming using C++

Chapter: Function and Operator Overloading

Coding Problems and Solutions

This set contains practical programming problems based on function overloading, inline and constant member functions, operator overloading (unary, binary, and using friend functions), and type conversion. Each problem includes a complete, compilable C++ solution along with a short explanation, useful for lab examinations, viva preparation, and সরকারি চাকরির পরীক্ষায় technical papers.

Problem 1: Overload a function to find the maximum of two values

Write a C++ program that overloads a function max() to return the greater of two integers, two floating-point numbers, and two characters.

Solution:

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

int max(int a, int b) {
    return (a > b) ? a : b;
}
float max(float a, float b) {
    return (a > b) ? a : b;
}
char max(char a, char b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max int: " << max(15, 42) << endl;
    cout << "Max float: " << max(3.7f, 2.9f) << endl;
    cout << "Max char: " << max('a', 'z') << endl;
    return 0;
}
```

Output:

```
Max int: 42
Max float: 3.7
Max char: z
```

Explanation:

Three separate functions share the name max but differ in the data type of their parameters. The compiler selects the correct version to call by matching the types of the arguments passed at each call site.

Problem 2: Inline member function to check even or odd

Write a C++ program with a class Number containing an inline member function isEven() that returns true if the stored number is even.

Solution:

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

class Number {
    int value;
public:
    Number(int v) { value = v; }
```

```

    bool isEven() {           // defined inside class: inline by default
        return (value % 2 == 0);
    }
};

int main() {
    Number n1(14), n2(27);
    cout << n1.isEven() << endl;    // 1 (true)
    cout << n2.isEven() << endl;    // 0 (false)
    return 0;
}

```

Output:

```

1
0

```

Explanation:

Since `isEven()` is defined inside the class body itself, the compiler treats it as an inline function by default and attempts to expand its code directly at each call, avoiding the overhead of a separate function call for this small check.

Problem 3: Constant member function in a BankAccount class

Write a program with a class `BankAccount` that stores a balance and provides a constant member function `getBalance()`. Demonstrate calling it on a const object.

Solution:

```

#include <iostream>
using namespace std;

class BankAccount {
    double balance;
public:
    BankAccount(double b) { balance = b; }

    double getBalance() const {    // constant member function
        return balance;
    }
};

int main() {
    const BankAccount acc(5400.50); // const object
    cout << "Balance: " << acc.getBalance() << endl;
    return 0;
}

```

Output:

```

Balance: 5400.5

```

Explanation:

`getBalance()` is declared `const` because it only reads the balance and never modifies it. This makes it callable on a `const BankAccount` object; a non-`const` member function could not be called on `acc` at all.

Problem 4: Overload the postfix increment (++) operator

Write a program to overload the postfix `++` operator for a class `Counter`, distinguishing it from the prefix version using a dummy `int` argument.

Solution:

```

#include <iostream>
using namespace std;

class Counter {
    int count;
public:
    Counter(int c = 0) { count = c; }

    Counter operator++() {           // prefix ++
        count++;
        return *this;
    }
    Counter operator++(int) {        // postfix ++ (dummy int marks postfix)
        Counter temp = *this;
        count++;
        return temp;
    }
    int getCount() const { return count; }
};

int main() {
    Counter c(5);
    Counter a = ++c;    // prefix: increments first, then assigns
    Counter b = c++;    // postfix: assigns first, then increments
    cout << "a = " << a.getCount() << endl;
    cout << "b = " << b.getCount() << endl;
    cout << "c = " << c.getCount() << endl;
    return 0;
}

```

Output:

```

a = 6
b = 6
c = 7

```

Explanation:

The prefix version `operator++()` takes no argument and returns the object after incrementing it. The postfix version `operator++(int)` takes an unused dummy int parameter (used only to tell the compiler which overload is postfix), saves the object's original state in temp, increments the actual object, and returns the saved original value.

Problem 5: Overload the unary minus operator for a 2x2 Matrix

Write a program to overload the unary minus (-) operator for a class Matrix representing a 2x2 matrix, so that it negates every element.

Solution:

```

#include <iostream>
using namespace std;

class Matrix {
    int m[2][2];
public:
    Matrix(int a, int b, int c, int d) {
        m[0][0] = a; m[0][1] = b; m[1][0] = c; m[1][1] = d;
    }
    Matrix operator-() {           // unary minus
        return Matrix(-m[0][0], -m[0][1], -m[1][0], -m[1][1]);
    }
    void display() {
        cout << m[0][0] << " " << m[0][1] << endl;
    }
}

```

```

        cout << m[1][0] << " " << m[1][1] << endl;
    }
};

int main() {
    Matrix m1(1, -2, 3, -4);
    Matrix m2 = -m1;    // calls operator-
    m2.display();
    return 0;
}

```

Output:

```

-1 2
-3 4

```

Explanation:

Since unary minus acts on a single operand, the overloaded member function operator-() takes no explicit argument. It builds and returns a new Matrix whose every element is the negative of the corresponding element of the calling object.

Problem 6: Overload the binary * operator for Complex number multiplication

Write a program to overload the binary * operator as a member function to multiply two Complex numbers.

Solution:

```

#include <iostream>
using namespace std;

class Complex {
    float real, imag;
public:
    Complex(float r = 0, float i = 0) { real = r; imag = i; }

    Complex operator*(Complex c) {    // binary * as member function
        float r = real * c.real - imag * c.imag;
        float i = real * c.imag + imag * c.real;
        return Complex(r, i);
    }
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2, 3), c2(1, 4);
    Complex c3 = c1 * c2;    // calls operator*
    c3.display();
    return 0;
}

```

Output:

```

-10 + 11i

```

Explanation:

As a member function, operator* takes one explicit argument (the right-hand operand); the left-hand operand is the calling object itself. The function applies the standard complex-number multiplication formula $(a+bi)(c+di) = (ac-bd) + (ad+bc)i$ and returns a new Complex object.

Problem 7: Overload the insertion operator (<<) using a friend function

Write a program to overload the << operator as a friend function so that a Point object can be printed directly using cout.

Solution:

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

class Point {
    int x, y;
    public:
        Point(int a, int b) { x = a; y = b; }

        friend ostream& operator<<(ostream &out, Point p); // friend function
};

ostream& operator<<(ostream &out, Point p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

int main() {
    Point p1(3, 7);
    cout << "Point: " << p1 << endl;    // calls friend operator<<
    return 0;
}
```

Output:

```
Point: (3, 7)
```

Explanation:

The << operator must have an ostream object (cout) as its left operand, so it cannot be overloaded as a member function of Point (which would require the Point object to be the left operand). It is therefore overloaded as a friend function taking the stream and the Point object explicitly, and returning the stream by reference so that output statements can be chained.

Problem 8: Overload the subscript ([]) operator for a custom Array class

Write a program to overload the [] operator for a class SafeArray so that individual elements can be accessed using array-style indexing.

Solution:

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

class SafeArray {
    int arr[5];
    public:
        SafeArray() {
            for (int i = 0; i < 5; i++) arr[i] = 0;
        }
        int& operator[](int index) {    // subscript operator, member function
only    return arr[index];
        }
};

int main() {
    SafeArray a;
    a[0] = 10;
    a[1] = 20;
}
```

```

a[2] = 30;
for (int i = 0; i < 3; i++)
    cout << a[i] << " ";
cout << endl;
return 0;
}

```

Output:

10 20 30

Explanation:

The [] operator can only be overloaded as a non-static member function, never as a friend function. It is written to return a reference to the internal array element, which allows it to be used both to read a value (a[i]) and to assign a new value to it (a[0] = 10).

Problem 9: Basic-to-class conversion: build a Fraction from an int and overload +

Write a program with a class Fraction that can be constructed directly from an int (basic-to-class conversion) and overloads + to add two fractions.

Solution:

```

#include <iostream>
using namespace std;

class Fraction {
    int num, den;
public:
    Fraction(int n, int d = 1) { num = n; den = d; }    // single-argument-
    capable constructor

    Fraction operator+(Fraction f) {
        return Fraction(num * f.den + f.num * den, den * f.den);
    }
    void display() {
        cout << num << "/" << den << endl;
    }
};

int main() {
    Fraction f1(1, 2);
    Fraction f2 = 3;        // int 3 converted to Fraction(3, 1)
    Fraction f3 = f1 + f2;   // calls operator+
    f3.display();
    return 0;
}

```

Output:

7/2

Explanation:

Because the constructor Fraction(int n, int d = 1) can be satisfied with a single argument (den defaults to 1), the compiler uses it as a conversion constructor: the plain int 3 is automatically converted into a Fraction object before being used in Fraction f2 = 3;. The overloaded + then adds the two fractions using the standard cross-multiplication rule.

Problem 10: Class-to-class conversion: Polar to Rectangular coordinates

Write a program that converts an object of class Polar (radius, angle) into an object of class Rectangular (x, y) using a constructor in the destination class.

Solution:

```
#include <iostream>
#include <cmath>
using namespace std;

class Polar {
public:
    float radius, angle;    // angle in radians
    Polar(float r, float a) { radius = r; angle = a; }
};

class Rectangular {
    float x, y;
public:
    Rectangular(Polar p) {           // constructor takes a Polar object
        x = p.radius * cos(p.angle);
        y = p.radius * sin(p.angle);
    }
    void display() {
        cout << "x = " << x << ", y = " << y << endl;
    }
};

int main() {
    Polar p1(5.0, 0.9273);    // approx. radius 5, angle for (3, 4)
    Rectangular r1 = p1;      // Polar object converted to Rectangular object
    r1.display();
    return 0;
}
```

Output:

```
x = 3, y = 4
```

Explanation:

The conversion from Polar to Rectangular is achieved through a constructor defined in the destination class (Rectangular) that accepts a single Polar argument. When a Polar object is assigned to a Rectangular variable, the compiler automatically invokes this constructor to perform the conversion using the standard polar-to-rectangular formulas $x = r \cos(\theta)$ and $y = r \sin(\theta)$.