

Object Oriented Programming using C++

Chapter: Function and Operator Overloading

Coding Problems and Solutions — Set 3

This is a third, all-new set of practical programming problems on function overloading, inline and constant member functions, and operator overloading — including relational, function-call, and scalar-multiplication operators not covered in the earlier sets — along with type conversion. Each problem includes a complete, compilable C++ solution with sample output and a short explanation, useful for lab examinations, viva preparation, and সরকারি চাকরির পরীক্ষায় technical papers.

Problem 1: Overload a function to print product details at varying detail levels

Write a C++ program that overloads a function printDetails() to display a Product's name only, name with price, or name with price and quantity, depending on how many arguments are passed.

Solution:

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

void printDetails(string name) {
    cout << "Product: " << name << endl;
}
void printDetails(string name, float price) {
    cout << "Product: " << name << ", Price: " << price << endl;
}
void printDetails(string name, float price, int qty) {
    cout << "Product: " << name << ", Price: " << price
        << ", Qty: " << qty << endl;
}

int main() {
    printDetails("Notebook");
    printDetails("Notebook", 45.0f);
    printDetails("Notebook", 45.0f, 10);
    return 0;
}
```

Output:

```
Product: Notebook
Product: Notebook, Price: 45
Product: Notebook, Price: 45, Qty: 10
```

Explanation:

All three functions share the name printDetails but differ in the number of parameters. The compiler matches each call to the version whose parameter count fits the arguments supplied, so the caller can print as much or as little detail as needed without using different function names.

Problem 2: Inline member function to compute the perimeter of a rectangle

Write a program with a class Rectangle that stores length and breadth, and provides an inline member function perimeter() to compute and return it.

Solution:

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

```

class Rectangle {
    float length, breadth;
public:
    Rectangle(float l, float b) { length = l; breadth = b; }

    float perimeter() {           // defined inside class: inline by default
        return 2 * (length + breadth);
    }
};

int main() {
    Rectangle r1(8.0, 5.0);
    cout << "Perimeter: " << r1.perimeter() << endl;
    return 0;
}

```

Output:

```
Perimeter: 26
```

Explanation:

Since `perimeter()` is defined inside the class body, the compiler treats it as inline by default and substitutes its code directly at the call site rather than performing a full function call, which is appropriate for such a short, frequently used calculation.

Problem 3: Constant member function to compute a discounted price

Write a program with a class `Product` that stores a price and a discount percentage, and provides a constant member function `discountedPrice()` that computes the final price without modifying the object.

Solution:

```

#include <iostream>
using namespace std;

class Product {
    float price;
    float discountPercent;
public:
    Product(float p, float d) { price = p; discountPercent = d; }

    float discountedPrice() const {           // constant member function
        return price - (price * discountPercent / 100);
    }
};

int main() {
    const Product p1(2000.0, 15.0);    // const object
    cout << "Final price: " << p1.discountedPrice() << endl;
    return 0;
}

```

Output:

```
Final price: 1700
```

Explanation:

`discountedPrice()` only reads `price` and `discountPercent` to compute a result and never changes either of them, so declaring it `const` is correct. This also makes it callable on the `const` object `p1`; a non-`const` member function could not be invoked on a `const` object at all.

Problem 4: Overload the binary + operator to add two 2x2 Matrix objects

Write a program to overload the + operator as a member function to add two 2x2 Matrix objects element by 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+(Matrix m2) { // binary + as member function
        return Matrix(m[0][0] + m2.m[0][0], m[0][1] + m2.m[0][1],
                      m[1][0] + m2.m[1][0], m[1][1] + m2.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), m2(5, 6, 7, 8);
    Matrix m3 = m1 + m2; // calls operator+
    m3.display();
    return 0;
}
```

Output:

```
6 8
10 12
```

Explanation:

The left operand of + is the calling object (m1) and the right operand is the explicit argument (m2). The function creates and returns a new Matrix whose every element is the sum of the corresponding elements of the two operand matrices.

Problem 5: Overload the binary * operator using a friend function to scale a Vector by an integer

Write a program to overload the * operator as a friend function so that a Vector object can be multiplied by an integer scalar to scale its components.

Solution:

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

class Vector {
    float x, y;
public:
    Vector(float a = 0, float b = 0) { x = a; y = b; }

    friend Vector operator*(Vector v, int scalar); // friend function
    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
```

```

Vector operator*(Vector v, int scalar) {
    return Vector(v.x * scalar, v.y * scalar);
}

int main() {
    Vector v1(2, 3);
    Vector v2 = v1 * 4;    // calls friend operator*
    v2.display();
    return 0;
}

```

Output:

```
(8, 12)
```

Explanation:

The operator function takes a Vector and an int as its two explicit arguments, since a friend function has no implicit calling object. This lets an expression like `v1 * 4` multiply a class-type object by a plain integer, something that could not be written using ordinary arithmetic between a class and a basic type.

Problem 6: Overload the postfix decrement (--) operator for a Countdown class

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

Solution:

```

#include <iostream>
using namespace std;

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

    Countdown operator--(int) {    // postfix -- (dummy int marks postfix)
        Countdown temp = *this;
        value--;
        return temp;
    }
    int getValue() const { return value; }
};

int main() {
    Countdown c(5);
    Countdown before = c--;    // postfix: returns original value, then decrements
    cout << "Before: " << before.getValue() << endl;
    cout << "After: " << c.getValue() << endl;
    return 0;
}

```

Output:

```

Before: 5
After: 4

```

Explanation:

The dummy int parameter in `operator--(int)` has no other purpose than telling the compiler that this overload is the postfix form. The function first saves the object's current state in `temp`, decrements the real object's value, and then returns the saved (pre-decrement) copy — matching the usual behaviour of postfix operators.

Problem 7: Overload the relational < operator to compare two Student objects by marks

Write a program to overload the < operator as a member function so that two Student objects can be compared directly based on their marks.

Solution:

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

class Student {
    string name;
    int marks;
public:
    Student(string n, int m) { name = n; marks = m; }

    bool operator<(Student s) {    // relational operator as member function
        return marks < s.marks;
    }
    string getName() const { return name; }
};

int main() {
    Student s1("Anita", 78), s2("Ravi", 85);
    if (s1 < s2)
        cout << s1.getName() << " scored less than " << s2.getName() << endl;
    else
        cout << s1.getName() << " did not score less than " << s2.getName() << endl;
    return 0;
}
```

Output:

```
Anita scored less than Ravi
```

Explanation:

operator< takes one explicit argument (the right-hand Student), while the left-hand Student is the calling object. It compares the marks members of the two objects and returns a bool, allowing Student objects to be compared with ordinary < syntax instead of a separately named comparison function.

Problem 8: Overload the function-call operator () to create a functor class

Write a program to overload the () operator for a class Adder so that objects of the class can be 'called' like a function to add a fixed value to their argument.

Solution:

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

class Adder {
    int base;
public:
    Adder(int b) { base = b; }

    int operator()(int x) {    // function-call operator
        return base + x;
    }
};

int main() {
    Adder add10(10);    // add10 behaves like a function that adds 10
    cout << add10(5) << endl;
}
```

```

    cout << add10(20) << endl;
    return 0;
}

```

Output:

```

15
30

```

Explanation:

Overloading () turns an object into a 'functor' — an object that can be used with call syntax, add10(5), just like a regular function. Here add10(5) is really shorthand for add10.operator()(5), which adds the object's stored base value (10) to the argument passed.

Problem 9: Class-to-basic conversion: convert a Marks object to its percentage as a float

Write a program with a class Marks that stores obtained and total marks, and provides a conversion function to convert a Marks object directly into a float representing the percentage.

Solution:

```

#include <iostream>
using namespace std;

class Marks {
    float obtained, total;
public:
    Marks(float o, float t) { obtained = o; total = t; }

    operator float() {          // conversion function: class -> float
        return (obtained / total) * 100;
    }
};

int main() {
    Marks m1(432, 500);
    float percentage = m1;      // implicit call to operator float()
    cout << "Percentage: " << percentage << endl;
    return 0;
}

```

Output:

```

Percentage: 86.4

```

Explanation:

operator float() is a member function that takes no arguments and specifies no explicit return type, since the type is given by the operator name itself. When a Marks object is assigned to a float variable, the compiler automatically invokes this function to perform the class-to-basic-type conversion.

Problem 10: Class-to-class conversion: convert Kilograms to Pounds using a constructor in the destination class

Write a program that converts an object of class Kilograms into an object of class Pounds using a constructor defined in the destination class (Pounds).

Solution:

```

#include <iostream>
using namespace std;

class Kilograms {

```

```

    public:
        float value;
        Kilograms(float v) { value = v; }
};

class Pounds {
    float value;
    public:
        Pounds(Kilograms k) {           // constructor takes a Kilograms object
            value = k.value * 2.205;
        }
        void display() {
            cout << value << " lb" << endl;
        }
};

int main() {
    Kilograms kg(10);
    Pounds lb = kg;           // Kilograms object converted to Pounds object
    lb.display();
    return 0;
}

```

Output:

```
22.05 lb
```

Explanation:

The conversion is achieved through a single-argument constructor in the destination class Pounds that accepts a Kilograms object. When a Kilograms object is assigned to a Pounds variable, the compiler automatically invokes this constructor to perform the conversion, here using the standard kilograms-to-pounds factor.