

Object Oriented Programming using C++

Chapter: Function and Operator Overloading

Coding Problems and Solutions — Set 2

This is a second, all-new set of practical programming problems on function overloading, inline and constant member functions, operator overloading (unary, binary, member and friend forms), and 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 find the volume of different solids

Write a C++ program that overloads a function volume() to compute the volume of a cube (one side), a cuboid (length, breadth, height), and a cylinder (radius, height).

Solution:

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

int volume(int side) {                // cube
    return side * side * side;
}
int volume(int l, int b, int h) {     // cuboid
    return l * b * h;
}
float volume(float r, float h) {      // cylinder
    return 3.14 * r * r * h;
}

int main() {
    cout << "Cube volume: " << volume(4) << endl;
    cout << "Cuboid volume: " << volume(2, 3, 5) << endl;
    cout << "Cylinder volume: " << volume(2.0f, 7.0f) << endl;
    return 0;
}
```

Output:

```
Cube volume: 64
Cuboid volume: 30
Cylinder volume: 87.92
```

Explanation:

The three volume() functions differ in the number and type of their parameters, which lets the compiler pick the correct one to call based purely on the arguments supplied at each call — one argument selects the cube version, three integer arguments select the cuboid version, and two float arguments select the cylinder version.

Problem 2: Inline member function to compute the area of a circle

Write a program with a class Circle that stores a radius and provides an inline member function getArea() to compute and return its area.

Solution:

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

```

class Circle {
    float radius;
public:
    Circle(float r) { radius = r; }

    float getArea() {                // defined inside class: inline by default
        return 3.14 * radius * radius;
    }
};

int main() {
    Circle c1(5.0);
    cout << "Area: " << c1.getArea() << endl;
    return 0;
}

```

Output:

```
Area: 78.5
```

Explanation:

Because getArea() is defined inside the class body, the compiler treats it as inline by default and attempts to substitute its code directly at the call site, which is efficient for such a small, frequently used calculation.

Problem 3: Constant member function to display employee details

Write a program with a class Employee that stores a name and salary, and provides a constant member function display() that can safely be called on a const Employee object.

Solution:

```

#include <iostream>
using namespace std;

class Employee {
    string name;
    float salary;
public:
    Employee(string n, float s) { name = n; salary = s; }

    void display() const {          // constant member function
        cout << name << " earns " << salary << endl;
    }
};

int main() {
    const Employee e1("Rahul", 32000.0); // const object
    e1.display();                        // allowed: display() is const
    return 0;
}

```

Output:

```
Rahul earns 32000
```

Explanation:

Since e1 is declared const, the compiler only allows const member functions to be called on it. display() only reads the object's data members and never modifies them, so declaring it const is both correct and necessary for it to be callable here; a non-const version of display() would not compile for a const object.

Problem 4: Overload the binary + operator to add two Time objects (with carry)

Write a program to overload the + operator as a member function to add two Time objects (storing hours and minutes), correctly carrying over minutes into hours.

Solution:

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

class Time {
    int hours, minutes;
public:
    Time(int h = 0, int m = 0) { hours = h; minutes = m; }

    Time operator+(Time t) {           // binary + as member function
        int h = hours + t.hours;
        int m = minutes + t.minutes;
        h += m / 60;
        m = m % 60;
        return Time(h, m);
    }
    void display() {
        cout << hours << " hr " << minutes << " min" << endl;
    }
};

int main() {
    Time t1(2, 45), t2(1, 40);
    Time t3 = t1 + t2;    // calls operator+
    t3.display();
    return 0;
}
```

Output:

```
4 hr 25 min
```

Explanation:

The left operand of + is the calling object (t1) and the right operand is passed as the explicit argument (t2). The function adds the hours and minutes separately, then converts any excess minutes (60 or more) into whole hours before returning the combined Time object.

Problem 5: Overload the binary - operator using a friend function for 2D vectors

Write a program to overload the - operator as a friend function to subtract two Vector2D objects.

Solution:

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

class Vector2D {
    float x, y;
public:
    Vector2D(float a = 0, float b = 0) { x = a; y = b; }

    friend Vector2D operator-(Vector2D v1, Vector2D v2); // friend function
    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

Vector2D operator-(Vector2D v1, Vector2D v2) {
    return Vector2D(v1.x - v2.x, v1.y - v2.y);
}
```

```

}

int main() {
    Vector2D v1(5, 8), v2(2, 3);
    Vector2D v3 = v1 - v2;    // calls friend operator-
    v3.display();
    return 0;
}

```

Output:

```
(3, 5)
```

Explanation:

Because operator- is declared friend, it is not a member of the class and therefore takes both operands explicitly as arguments. Being a friend, it still has direct access to the private members x and y of both Vector2D objects passed to it.

Problem 6: Overload the prefix decrement (--) operator for a Stock class

Write a program to overload the prefix -- operator for a class Stock that reduces the quantity in hand by one unit each time it is applied.

Solution:

```

#include <iostream>
using namespace std;

class Stock {
    int quantity;
public:
    Stock(int q) { quantity = q; }

    Stock operator--() {          // prefix -- : no explicit argument
        quantity--;
        return *this;
    }
    int getQuantity() const { return quantity; }
};

int main() {
    Stock s(10);
    --s;
    --s;
    cout << "Quantity left: " << s.getQuantity() << endl;
    return 0;
}

```

Output:

```
Quantity left: 8
```

Explanation:

Since -- is a unary operator here, the member function operator--() takes no explicit argument; the object on which it is applied is accessed through the implicit this pointer. Each application reduces quantity by one and returns the updated object.

Problem 7: Overload == and != to compare two Fraction objects

Write a program to overload both the == and != operators as member functions to compare two Fraction objects for equal value.

Solution:

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

class Fraction {
    int num, den;
public:
    Fraction(int n, int d) { num = n; den = d; }

    bool operator==(Fraction f) { // cross-multiplication for equal value
        return (num * f.den == f.num * den);
    }
    bool operator!=(Fraction f) {
        return !(*this == f); // reuses operator==
    }
};

int main() {
    Fraction f1(1, 2), f2(2, 4), f3(3, 5);
    cout << (f1 == f2 ? "Equal" : "Not equal") << endl;
    cout << (f1 != f3 ? "Not equal" : "Equal") << endl;
    return 0;
}
```

Output:

```
Equal
Not equal
```

Explanation:

Two fractions are equal in value when $\text{num1} * \text{den2}$ equals $\text{num2} * \text{den1}$, even if their numerators and denominators differ (as with $1/2$ and $2/4$). `operator!=` is written in terms of `operator==` using the `this` pointer, avoiding duplicate comparison logic.

Problem 8: Overload << and >> as friend functions for a Student class

Write a program that overloads both the << (insertion) and >> (extraction) operators as friend functions so that a Student object can be read from and displayed to the console directly.

Solution:

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

class Student {
    string name;
    int marks;
public:
    friend istream& operator>>(istream &in, Student &s);
    friend ostream& operator<<(ostream &out, Student &s);
};

istream& operator>>(istream &in, Student &s) {
    cout << "Enter name and marks: ";
    in >> s.name >> s.marks;
    return in;
}

ostream& operator<<(ostream &out, Student &s) {
    out << s.name << " scored " << s.marks << " marks";
    return out;
}
```

```
int main() {
    Student s1;
    // cin >> s1;        // reads name and marks from the user
    s1 = Student();      // (for this demo, values are set directly instead)
    cout << s1 << endl;
    return 0;
}
```

Output:

```
Enter name and marks: (reads input, then displays result)
```

Explanation:

Both operators must take the stream as the left operand (cin or cout), so neither can be a member function of Student; both are declared friend so they can still access the private members name and marks directly. operator>> takes the Student by reference so the values it reads are stored back into the caller's object, and both functions return the stream by reference to allow chaining.

Problem 9: Basic-to-class conversion: build Money from an int (paise) and overload +

Write a program with a class Money that can be constructed directly from an int number of paise (basic-to-class conversion) and overloads + to add two Money amounts.

Solution:

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

class Money {
    long paise;
public:
    Money(long p) { paise = p; }          // conversion constructor

    Money operator+(Money m) {
        return Money(paise + m.paise);
    }
    void display() {
        cout << "Rs " << paise / 100 << "." << paise % 100 << endl;
    }
};

int main() {
    Money m1(15075);          // Rs 150.75
    Money m2 = 5025;          // int converted to Money: Rs 50.25
    Money m3 = m1 + m2;       // calls operator+
    m3.display();
    return 0;
}
```

Output:

```
Rs 201.0
```

Explanation:

Because Money has a single-argument constructor, the compiler treats it as a conversion constructor: the plain int 5025 in Money m2 = 5025; is automatically converted into a Money object before being stored. The overloaded + then simply adds the paise values of the two Money objects.

Problem 10: Class-to-class conversion using a conversion function in the source class

Write a program that converts a Fahrenheit object into a Celsius object using a conversion function defined in the source class (Fahrenheit), rather than a constructor in the destination class.

Solution:

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

class Celsius {
public:
    float temp;
    Celsius(float t) { temp = t; }
    void display() { cout << temp << " C" << endl; }
};

class Fahrenheit {
    float temp;
public:
    Fahrenheit(float t) { temp = t; }

    operator Celsius() {                // conversion function to Celsius
        float c = (temp - 32) * 5 / 9;
        return Celsius(c);
    }
};

int main() {
    Fahrenheit f(212);
    Celsius c = f;          // calls Fahrenheit::operator Celsius()
    c.display();
    return 0;
}
```

Output:

```
100 C
```

Explanation:

Here the conversion route is placed in the source class (Fahrenheit) rather than the destination class (Celsius): the member function operator Celsius() takes no arguments and is named after the target class. When a Fahrenheit object is assigned to a Celsius variable, the compiler automatically calls this conversion function. This is the alternate method to a destination-class constructor, and both cannot be defined for the same conversion at once without causing ambiguity.