

Object Oriented Programming using C++

Chapter: Function and Operator Overloading

Important Exam-Oriented Questions and Answers

This set covers short answer questions, broad/long answer questions with complete solution steps, and programming (coding) type questions with answers, based on function overloading, inline and constant member functions, operator overloading, and type conversion — useful for semester exams as well as সরকারি চাকরির পরীক্ষায় technical/departmental papers.

Section A: Short Answer Type Questions

Q1. Define function overloading.

Function overloading is a feature of C++ where two or more functions share the same name in the same scope but differ in the number, type, or order of their parameters. The correct function is selected by the compiler at compile time based on the arguments passed.

Q2. Can functions be overloaded by changing only the return type?

No. The compiler resolves overloaded functions using the parameter list, not the return type. If two functions have identical parameter lists but different return types, the compiler cannot distinguish between them and reports an ambiguity error.

Q3. What is an inline function? Give the keyword used to declare one.

An inline function is a function whose code the compiler inserts directly at each point of call instead of performing a normal function call, in order to reduce call overhead. It is declared using the inline keyword before the function definition.

Q4. When does a member function become inline automatically?

A member function becomes inline automatically when it is defined inside the class definition itself, without needing the explicit inline keyword.

Q5. What is a constant member function? How is it declared?

A constant member function is a member function that cannot modify the data members of the calling object. It is declared by writing the const keyword immediately after the function's parameter list, e.g.,

```
int getValue() const;
```

Q6. Why is a constant member function needed for a const object?

A const object can only call member functions that are themselves declared const, because the compiler must guarantee that no function invoked on the object will attempt to modify its state.

Q7. Define operator overloading.

Operator overloading is a mechanism that allows the standard C++ operators (such as +, -, ==, <<) to be redefined for user-defined class types by writing a special operator function, so that objects can be manipulated using normal operator syntax.

Q8. How many explicit arguments does an overloaded unary operator member function take, and why?

It takes zero explicit arguments, because the single operand of a unary operator is the calling object itself, accessed implicitly through the this pointer.

Q9. How many explicit arguments does an overloaded binary operator member function take, and why?

It takes exactly one explicit argument, because the left operand is the calling object itself and the right operand is passed as the function's argument.

Q10. Give the general syntax for declaring an operator function.

```
return_type operator symbol (argument_list) {  
    // function body  
}
```

For example, `Complex operator+ (Complex c)` declares an overloaded `+` operator returning a `Complex` object.

Q11. Name any four operators that cannot be overloaded in C++.

Scope resolution (`::`), member selection (`.`), pointer-to-member (`.*`), and the conditional/ternary operator (`?:`). The `sizeof` and `typeid` operators also cannot be overloaded.

Q12. What is a conversion constructor? Give an example.

A conversion constructor is a single-argument constructor that the compiler automatically uses to convert a value of a basic data type into a class-type object.

```
class Distance {  
    int meters;  
public:  
    Distance(int m) { meters = m; }    // conversion constructor  
};  
Distance d = 10;    // int converted to Distance
```

Q13. What is a casting (conversion) operator function? State one restriction on it.

It is a member function used to convert a class-type object into a basic data type, written as `operator type()` with no arguments and no explicit return type. It must be a member function of the class — it cannot be defined as a friend function.

Q14. Can a class-to-class conversion be performed in more than one way? Name them.

Yes. It can be done either through a single-argument constructor in the destination class that takes an object of the source class, or through a conversion function defined in the source class named after the destination class.

Section B: Broad / Long Answer Type Questions

Q1. Explain function overloading in C++ with the rules governing it. Illustrate with a program that overloads a function to find the area of a square, rectangle, and circle.

Answer:

Function overloading allows multiple functions to share the same name within a scope, provided their parameter lists differ in number, type, or order. The compiler resolves the correct function to call at compile time by matching the arguments in the call with the closest matching function signature. Overloading cannot be done on the basis of return type alone.

Program:

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

void area(int side) {
    cout << "Area of square: " << side * side << endl;
}
void area(int l, int b) {
    cout << "Area of rectangle: " << l * b << endl;
}
void area(float r) {
    cout << "Area of circle: " << 3.14 * r * r << endl;
}

int main() {
    area(5);           // calls square version
    area(4, 6);        // calls rectangle version
    area(2.5f);        // calls circle version
    return 0;
}
```

Here, the same function name `area` is used for three different shapes; the compiler decides which definition to invoke purely from the arguments supplied in each call.

Q2. What are inline and constant member functions? Explain both with suitable examples and state the situations where each is used.

Answer:

An inline function instructs the compiler to substitute the function's code directly at the point of call rather than performing a normal call, reducing function-call overhead for small, frequently used functions. A member function becomes inline automatically when defined inside the class body, or can be made inline explicitly using the inline keyword outside it.

A constant member function is one that is guaranteed not to modify the object's data members. It is declared by placing `const` after the parameter list and is required whenever the function must be callable on a const object.

Program:

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

class Student {
    int marks;
```

```

public:
    Student(int m) { marks = m; }
    int getMarks() const {    // constant member function, inline by default
        return marks;
    }
};

int main() {
    const Student s(85);    // const object
    cout << "Marks: " << s.getMarks() << endl;
    return 0;
}

```

Inline functions are used for small accessor/mutator functions to improve speed, while constant member functions are used whenever a function must guarantee it will not alter the object, and are mandatory for calling on const objects.

Q3. Explain the overloading of unary and binary operators as member functions. Write a program to overload the unary minus (-) and the binary + operator for a class representing complex numbers.

Answer:

A unary operator, overloaded as a member function, acts on the single object that calls it and therefore takes no explicit argument. A binary operator, overloaded as a member function, acts on two operands — the calling object (left operand) and one explicit argument (right operand).

```

#include <iostream>
using namespace std;

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

    Complex operator-() {                // unary minus
        return Complex(-real, -imag);
    }
    Complex operator+(Complex c) {       // binary plus
        return Complex(real + c.real, imag + c.imag);
    }
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;    // calls operator+
    Complex c4 = -c1;        // calls operator-
    c3.display();
    c4.display();
    return 0;
}

```

The unary minus flips the sign of both the real and imaginary parts of a single Complex object, whereas the binary plus adds the corresponding members of two Complex objects to produce a new object.

Q4. State and explain the rules for overloading operators in C++.

Answer:

- Only existing operators can be overloaded; new symbols cannot be invented.
- The overloading does not change the precedence, associativity, or the number of operands (arity) of the operator.
- At least one operand involved in the operation must be an object of a user-defined class.
- Default arguments cannot be supplied in an operator function.
- Overloaded operators (except =) are not automatically inherited to a derived class in the usual sense and may need to be redefined.
- Operators such as =, [], (), and -> can be overloaded only as non-static member functions, never as friend functions.

These rules ensure that operator overloading extends the natural meaning of an operator to class types without violating the fundamental syntax and semantics of the language.

Q5. Explain the three types of type conversion involving class objects in C++, with one program illustrating each.

Answer:

(a) Basic type to class type — using a single-argument constructor:

```
class Distance {
    int meters;
public:
    Distance(int m) { meters = m; }
    void show() { cout << meters << " m" << endl; }
};
int main() {
    Distance d = 25;    // int converted to Distance object
    d.show();
}
```

(b) Class type to basic type — using a conversion (casting) function:

```
class Distance {
    int meters;
public:
    Distance(int m) { meters = m; }
    operator int() { return meters; }    // conversion function
};
int main() {
    Distance d(25);
    int m = d;    // Distance object converted to int
    cout << m;
}
```

(c) Class type to class type — using a constructor in the destination class:

```
class Feet {
public:
    int value;
    Feet(int v) { value = v; }
};
class Meters {
public:
    int value;
    Meters(Feet f) { value = f.value / 3; }    // converts Feet to Meters
};
int main() {
    Feet f(9);
}
```

```
Meters m = f;    // Feet object converted to Meters object
cout << m.value;
}
```

In each case, the compiler is given an explicit rule — a constructor or a conversion function — that tells it how to move a value from one type to another, since such conversions are never performed automatically for user-defined class types.

Section C: Programming (Coding) Type Questions

Q1. Write a C++ program to overload the binary + operator using a friend function to add two Distance objects (given in feet and inches).

Answer:

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

class Distance {
    int feet, inches;
public:
    Distance(int f = 0, int i = 0) { feet = f; inches = i; }

    friend Distance operator+(Distance d1, Distance d2); // friend function
    void display() {
        cout << feet << " feet " << inches << " inches" << endl;
    }
};

Distance operator+(Distance d1, Distance d2) {
    int f = d1.feet + d2.feet;
    int i = d1.inches + d2.inches;
    f += i / 12;
    i = i % 12;
    return Distance(f, i);
}

int main() {
    Distance d1(5, 8), d2(3, 9);
    Distance d3 = d1 + d2; // calls friend operator+
    d3.display();
    return 0;
}
```

Because it is a friend function, operator+ takes both operands explicitly as arguments and has direct access to the private data members feet and inches of the Distance class.

Q2. Write a program that overloads the == operator to compare two objects of a class Box for equal volume.

Answer:

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

class Box {
    int length, breadth, height;
public:
    Box(int l, int b, int h) { length = l; breadth = b; height = h; }
    int volume() const { return length * breadth * height; }

    bool operator==(Box b) { // overloaded equality operator
        return volume() == b.volume();
    }
};

int main() {
    Box b1(2, 3, 4), b2(4, 3, 2), b3(1, 1, 1);
```

```

    cout << (b1 == b2 ? "Equal volume" : "Different volume") << endl;
    cout << (b1 == b3 ? "Equal volume" : "Different volume") << endl;
    return 0;
}

```

The volume() function is declared const since it only reads data members, and operator== compares the volumes of the calling object and the argument object, returning a boolean result.

Q3. Write a program using a conversion function to convert a Fahrenheit temperature object into an equivalent Celsius double value.

Answer:

```

#include <iostream>
using namespace std;

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

    operator double() {          // conversion function: class -> double (Celsius)
        return (temp - 32) * 5 / 9;
    }
};

int main() {
    Fahrenheit f(98.6);
    double celsius = f;    // implicit call to operator double()
    cout << "Temperature in Celsius: " << celsius << endl;
    return 0;
}

```

When a Fahrenheit object is assigned to a double variable, the compiler automatically invokes the operator double() conversion function defined inside the class.

Q4. Write a program to overload the prefix increment (++) operator for a class Counter.

Answer:

```

#include <iostream>
using namespace std;

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

    Counter operator++() {    // prefix ++ : no explicit argument
        count++;
        return *this;
    }
    int getCount() const { return count; }
};

int main() {
    Counter c;
    ++c;
    ++c;
    ++c;
    cout << "Count: " << c.getCount() << endl;
    return 0;
}

```

Since ++ is a unary operator here, the overloaded member function operator++() takes no explicit argument; it increments the object's own count member and returns the updated object.