Object Oriented Programming using C++

Chapter: Function and Operator Overloading

These notes are prepared for engineering diploma / degree students appearing in semester examinations and are equally useful for candidates preparing for সরকারি চাকরির পরীক্ষায় technical and departmental exams that include Object Oriented Programming as a subject.

3.2 Function Overloading, Inline Member Functions, Constant Member Functions

Function Overloading

Function overloading is a feature of C++ that allows two or more functions to have the same name within the same scope, provided their parameter lists are different. The compiler decides which function to call by matching the number, type, and order of the arguments passed during the function call. This mechanism is an example of compile-time (static) polymorphism, because the decision of which function to invoke is made at compile time itself.

Functions can be overloaded by changing:

- The number of parameters (e.g., add(int, int) and add(int, int, int)).
- The data type of parameters (e.g., add(int, int) and add(float, float)).
- The order of parameters of different types (e.g., show(int, float) and show(float, int)).

Function overloading cannot be achieved merely by changing the return type of a function while keeping the parameter list identical, because the compiler resolves overloaded calls purely on the basis of the argument list, not the return type.

```
void area(int side);            // overload 1: square
void area(int l, int b);        // overload 2: rectangle
void area(float radius);        // overload 3: circle
```

Advantages of function overloading include improved code readability, since logically related operations can share a single, meaningful function name, and reduced program complexity, since the programmer does not need to remember different names for essentially the same operation performed on different data types.

Inline Member Functions

An inline function is a function for which the compiler is requested to insert the complete body of the function at each point the function is called, instead of generating a normal function call with the usual overhead of pushing arguments onto the stack, jumping to the function code, and returning. This substitution is done at compile time and is intended to improve execution speed at the cost of a slightly larger executable size.

A member function of a class becomes inline automatically when it is defined inside the class definition itself. A member function defined outside the class body can also be made inline explicitly by using the inline keyword before its definition.

```
class Box {
  public:
    int volume() {              // defined inside class: inline by default
```

```
    return length * breadth * height;
  }
private:
  int length, breadth, height;
};
```

Inline functions are best suited for small, frequently used functions such as get/set (accessor and mutator) functions. The inline keyword is only a request to the compiler; the compiler may ignore it for large functions, functions containing loops, switch statements, or recursive calls, and expand them as ordinary functions instead.

Constant Member Functions

A constant member function is a member function that guarantees it will not modify the data members of the object on which it is called. Such a function is declared by placing the const keyword after the closing parenthesis of the parameter list, both in the function declaration and its definition.

```
class Student {
  int marks;
  public:
    int getMarks() const {     // constant member function
      return marks;
    }
};
```

If a constant member function attempts to modify any non-static data member of the class, or attempts to call another member function that is not itself declared as const, the compiler generates an error. Constant member functions are essential when a member function needs to be called on a constant object, since a const object can only invoke const member functions. They also serve as a form of documentation, telling the programmer at a glance that the function only reads, and never writes to, the state of the object.

3.3 Operator Overloading and Type Conversion

Operator Overloading

Operator overloading allows the existing operators of C++ (such as +, -, *, ==, <<, etc.) to be redefined so that they work with user-defined data types (objects of a class) in addition to their usual meaning with built-in data types. It is achieved by writing a special function called an operator function, using the keyword operator followed by the symbol of the operator being overloaded. Operator overloading is another form of compile-time polymorphism and allows objects to be manipulated using natural, intuitive operator notation, for example adding two complex numbers using the + sign instead of calling a function such as addComplex().

Overloading Unary Operators

A unary operator acts upon a single operand, examples being the increment (++), decrement (--), and unary minus (-) operators. When a unary operator is overloaded as a member function, it takes no explicit arguments, because the single operand is the object itself (accessed through the implicit this pointer).

```
class Distance {
  int meters;
  public:
    Distance(int m) : meters(m) {}
    Distance operator-() {        // overloading unary minus
      return Distance(-meters);
```

```
    }
};
```

Overloading Binary Operators

A binary operator acts upon two operands, examples being +, -, *, and ==. When a binary operator is overloaded as a member function, it takes exactly one explicit argument, since the left-hand operand is the calling object itself and the right-hand operand is passed as the argument.

```
class Complex {
  float real, imag;
  public:
    Complex(float r=0, float i=0) : real(r), imag(i) {}
    Complex operator+(Complex c) {   // overloading binary +
      return Complex(real + c.real, imag + c.imag);
    }
};
```

Binary operators can also be overloaded using a friend function, in which case the function takes two explicit arguments, one for each operand, since a friend function has no implicit this pointer and no automatic access to the calling object.

Rules for Overloading Operators

- Only existing C++ operators can be overloaded; new operator symbols cannot be created.
- The basic meaning and precedence of an operator cannot be changed by overloading; overloading only extends the operator to work with new (class) types.
- At least one of the operands of an overloaded operator must be an object of a user-defined class.
- The number of operands an operator takes cannot be changed; a binary operator must remain binary and a unary operator must remain unary.
- Default arguments cannot be used with an overloaded operator function.
- Overloaded operators, except for the assignment operator (=), are not inherited automatically to any newly derived class in the same way as ordinary member functions; if needed, they must be explicitly redefined.
- An operator function must either be a member function of the class or a friend function of the class; it cannot be a plain global function unless it is declared friend or takes class-type arguments explicitly.
- Operators like =, [], (), and -> can only be overloaded as non-static member functions, not as friend functions.

Type Conversion

Since a class is a user-defined data type, the C++ compiler does not automatically know how to convert between class objects and the built-in (basic) data types, or between objects of two different classes. Programmers must define such conversions explicitly. There are three broad categories of type conversion involving class objects.

(a) Conversion from Basic Type to Class Type

This conversion is carried out using a single-argument constructor. A constructor that accepts exactly one argument (or accepts multiple arguments where all but one have default values) is automatically used by the

compiler as a conversion function whenever a value of the basic type is assigned to, or used to initialise, an object of the class.

```
class Distance {
  int meters;
  public:
    Distance(int m) { meters = m; }   // single-argument constructor
};

Distance d = 25;   // int 25 converted to a Distance object
```

(b) Conversion from Class Type to Basic Type

This conversion is carried out using a special member function of the class called a conversion function, or overloaded casting operator. It is written using the keyword operator followed by the name of the target basic type, takes no arguments, and specifies no return type (since the return type is implied by the operator name itself). It must be defined as a member function of the class being converted from; it cannot be a friend function.

```
class Distance {
  int meters;
  public:
    Distance(int m) { meters = m; }
    operator int() {           // conversion function: class -> int
      return meters;
    }
};

Distance d(25);
int m = d;   // Distance object converted to int
```

(c) Conversion from One Class Type to Another Class Type

When an object of one class needs to be converted into an object of another class, the conversion can be performed in either of two ways, and the compiler chooses whichever is applicable and accessible.

- Using a single-argument constructor in the destination class, which takes an object of the source class as its argument.
- Using a conversion function (operator function) in the source class, named after the destination class, for example operator DestinationClass().

If both methods are defined and applicable at the same time, the conversion becomes ambiguous and the compiler reports an error, so care must be taken to provide only one route for a given class-to-class conversion.

Operators That Cannot Be Overloaded

A small set of operators in C++ cannot be overloaded, mainly because changing their behaviour could break the fundamental rules of the language or the structure of a program. These are:

- Scope resolution operator (::)
- Member selection / dot operator (.)
- Pointer-to-member operator (.*)
- Conditional / ternary operator (?:)
- sizeof operator

- typeid operator

Multiple Choice Questions (MCQs)

(Each question carries one mark. Answer key is given at the end.)

1. Function overloading in C++ is an example of:

(a) Run-time polymorphism
(b) Compile-time polymorphism
(c) Inheritance
(d) Encapsulation

2. Which of the following can be used to overload functions?

(a) Changing only the return type
(b) Changing the number or type of parameters
(c) Changing the function's access specifier
(d) Changing the function's name

3. A member function defined inside the class body is treated by the compiler as:

(a) Static
(b) Virtual
(c) Inline by default
(d) Friend

4. The inline keyword is:

(a) A command that is always obeyed by the compiler
(b) A request that the compiler may or may not honour
(c) Applicable only to constructors
(d) Used only with overloaded operators

5. A constant member function is declared by placing const:

(a) Before the function name
(b) After the parameter list
(c) Before the return type
(d) Inside the function body only

6. A constant member function can be called on:

(a) Only non-const objects
(b) Only const objects
(c) Both const and non-const objects
(d) Neither const nor non-const objects

7. Operator overloading is implemented in C++ using:

(a) A macro
(b) A special function with the keyword operator
(c) The keyword overload
(d) Template specialisation only

8. When a unary operator is overloaded as a member function, the number of explicit arguments it takes is:

(a) Zero

(b) One
(c) Two
(d) Depends on the operator

9. When a binary operator is overloaded as a member function, the number of explicit arguments it takes is:

(a) Zero
(b) One
(c) Two
(d) Three

10. Which of the following statements about operator overloading is correct?

(a) New operator symbols can be created
(b) The precedence of an operator can be changed
(c) At least one operand must be a user-defined type
(d) Default arguments are allowed in operator functions

11. Conversion from a basic data type to a class type is normally achieved using:

(a) A destructor
(b) A single-argument constructor
(c) A friend function only
(d) The this pointer

12. A conversion function that converts a class object to a basic type:

(a) Must be a friend function
(b) Must specify an explicit return type
(c) Must be a member function and specifies no return type
(d) Cannot be defined by the programmer

13. Conversion from one class type to another can be done using:

(a) Only a constructor in the destination class
(b) Only a conversion function in the source class
(c) Either a constructor in the destination class or a conversion function in the source class
(d) It is not possible in C++

14. Which of the following operators CANNOT be overloaded in C++?

(a) + (plus)
(b) == (equal to)
(c) :: (scope resolution)
(d) [] (subscript)

15. Which operators can only be overloaded as member functions and not as friend functions?

(a) + and -
(b) = and []
(c) << and >>
(d) ++ and --

Answer Key

1-b 2-b 3-c 4-b 5-b 6-c 7-b 8-a 9-b 10-c 11-b 12-c 13-c 14-c 15-b

Broad / Long Answer Type Questions

1. What is function overloading? Explain the rules for overloading functions in C++ with suitable examples.
2. Distinguish between function overloading and function overriding. Why can functions not be overloaded on the basis of return type alone?
3. What is an inline function? Explain how a member function becomes inline, and discuss the situations in which the compiler may refuse to expand a function inline even after it is declared so.
4. What is a constant member function? Explain its syntax and significance with an example, and state why it is necessary when dealing with constant objects.
5. What is operator overloading? Explain, with an example, how a binary operator is overloaded as a member function of a class.
6. Explain, with a suitable example, how a unary operator such as the increment operator (++) can be overloaded for a user-defined class, both as a member function and as a friend function.
7. State and explain, with examples, the general rules that must be followed while overloading operators in C++.
8. Explain the process of converting a basic data type into a class type, with a complete program illustrating the use of a single-argument constructor for this purpose.
9. Explain the process of converting a class type into a basic data type, with a complete program illustrating the use of a conversion (casting) function.
10. Discuss the different ways in which an object of one class can be converted into an object of another class. Explain, with reasoning, what happens if both methods are defined simultaneously for the same conversion.