Unit - III

Operator Overloading

Customised behaviour of operators

Unit Introduction

This unit covers operator overloading

Unit Objectives

After covering this unit you will understand…

- Operator overloading
- Different types of operator and their overloading
- Operators that cannot be overloaded
- Inheritance and overloading
- Automatic type conversion

Introduction

- Operator overloading
  - Enabling C++'s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care; when overloading is misused, program difficult to understand
  - Examples of already overloaded operators
    - Operator **<<** is both the stream-insertion operator and the bitwise left-shift operator
    - **+** and **-**, perform arithmetic on multiple types
  - Compiler generates the appropriate code based on the manner in which the operator is used

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`operator+`** used to overload the addition operator (**`+`**)
- Using operators
  - To use an operator on a class object it must be overloaded unless the assignment operator **`(=)`** or the address operator **`(&)`**
    - Assignment operator by default performs memberwise assignment
    - Address operator (&) by default returns the address of an object

Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

- C++ Operators that cannot be overloaded

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators `&`, `*`, `+` and `-` each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

Operator Functions as Class Members vs. as friend Functions

- Member vs non-member
  - In general, operator functions can be member or non-member functions
    - When overloading `( )`, `[ ]`, `->` or any of the assignment operators, must use a member function
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
    - If left operand of a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - Must be **`friend`**s if needs to access private or protected members
  - Enable the operator to be commutative

Overloading Stream-Insertion and Stream-Extraction Operators

- Overloaded **<<** and **>>** operators
  - Overloaded to perform input/output for user-defined types
  - Left operand of types **ostream &** and **istream &**
  - Must be a non-member function because left operand is not an object of the class
  - Must be a **friend** function to access private data members

Overloading Unary Operators

- Overloading unary operators
  - Can be overloaded with no arguments or one argument
  - Should usually be implemented as member functions
    - Avoid **`friend`** functions and classes because they violate the encapsulation of a class
  - Example declaration as a member function:

```
class String {
public:
   bool operator!() const;
   ...
};
```

Overloading Unary Operators

- Example declaration as a non-member function

```
class String {
   friend bool operator!( const
String & )
   ...
}
```

Overloading Binary Operators

- Overloaded Binary operators
  - Non-static member function, one argument
  - Example:

```
class Complex {
public:
    const Complex operator+ (
                    const Complex &);
    ...
};
```

Overloading Binary Operators

- Non-member function, two arguments
- Example:

```
class Complex {
    friend Complex operator +(
      const Complex &, const Complex &
  );
    ...
};
```

Example: Operator Overloading

```
class OverloadingExample
{
    private:
        int m_LocalInt;
    public:
        OverloadingExample(int j) // default constructor
        {
            m_LocalInt = j;
        }
        int operator+ (int j) // overloaded + operator
        {
            return (m_LocalInt + j);
        }
};
```

Example: Operator Overloading (contd.)

```
void main()
{
    OverloadingExample object1(10);
    cout << object1 + 10; // overloaded operator called
}
```

Types of Operator

- Unary operator
- Binary operator

Unary Operators

- Operators attached to a single operand (-a, +a, --a, a--, ++a, a++)

Example: Unary Operators

```
class UnaryExample
{
    private:
        int m_LocalInt;
    public:
        UnaryExample(int j)
        {
            m_LocalInt = j;
        }
        int operator++ ()
        {
            return (m_LocalInt++);
        }
};
```

Example: Unary Operators (contd.)

```
void main()
{
    UnaryExample object1(10);
    cout << object1++; // overloaded operator results in value
                       // 11
}
```

Unary Overloaded Operators -- Member Functions

- Invocation in Two Ways -- *Object@* (Direct) or *Object.operator@()* (As a Function)

```
class number{
    int n;
  public:
    number(int x = 0):n(x){};
    number operator-(){return number (-n);}
};
main()
{
  number a(1), b(2), c, d;
  //Invocation of "-" Operator -- direct
  d = -b; //d.n = -2
  //Invocation of "-" Operator -- Function
  c = a.operator-(); //c.n = -1
}
```

Binary Overloaded Operators -- Member Functions

- Invocation in Two Ways -- *ObjectA @ ObjectB* (direct) or *ObjectA.operator@(ObjectB)* (As a Function)

```
class number{
    int n;
  public:
    number(int x = 0):n(x){};
    number operator+(number ip)
        {return number (ip.n + n);}
};
main()
{
  number a(1), b(2), c, d;
  //Invocation of "+" Operator -- direct
  d = a + b; //d.n = 3
  //Invocation of "+" Operator -- Function
  c = d.operator+(b); //c.n = d.n + b.n = 5
}
```

Binary Operators

- Operators attached to two operands (a-b, a+b, a*b, a/b, a%b, a>b, a>=b, a<b, a<=b, a==b)

Example: Binary Operators

```
class BinaryExample
{
    private:
        int m_LocalInt;
    public:
        BinaryExample(int j)
        {
            m_LocalInt = j;
        }
        int operator+ (BinaryExample& rhsObj)
        {
            return (m_LocalInt + rhsObj.m_LocalInt);
        }
};
```

Example: Binary Operators (contd.)

```
void main()
{
    BinaryExample object1(10), object2(20);
    cout << object1 + object2; // overloaded operator called
}
```

Non-Overloadable Operators

- Operators that can not be overloaded due to safety reasons:
  - Member Selection '.' operator
  - Member dereference '.*' operator
  - Exponential '**' operator
  - User-defined operators
  - Operator precedence rules

Operator Overloading and Inheritance

- An operator is overloaded in super class but not overloaded in derived class is called non-member operator in derived class
- In above, if operator is also overloaded in derived class it is called member-operator
- = ( ) [ ] –> –>* operators must be member operators
- Other operators can be non-member operators

Automatic Type Conversion

- Automatic type conversion by the C++ compiler from the type that doesn't fit, to the type it wants
- Two types of conversion:
  - Constructor conversion
  - Operator conversion

Constructor Conversion

- Constructor having a single argument of another type, results in automatic type conversion by the compiler
- Prevention of constructor type conversion by use of **explicit** keyword

Example: Constructor Conversion

```
class One
{
    public:
        One() {}
};
class Two
{
    public:
        Two(const One&) {}
};
void f(Two) {}

void main()
{
    One one;
    f(one); // Wants a Two, has a One
}
```

Operator Conversion

- Create a member function that takes the current type
- Converts it to the desired type using the **operator** keyword followed by the type you want to convert to
- Return type is the **name** of the operator overloaded
- Reflexivity - global overloading instead of member overloading; for code saving

Example: Operator Conversion

```
class Three
{
    int m_Data;
    public:
        Three(int ii = 0, int = 0) : m_Data(ii) {}
};
class Four
{
    int m_Data;
    public:
        Four(int x) : m_Data(x) {}
        operator Three() const
        {
            return Three(m_Data);
        }
};
void g(Three) {}
```

Example: Operator Conversion (contd.)

```
void main()
{
    Four four(1);
    g(four);
    g(1); // Calls Three(1,0)
}
```

Type Conversion Pitfalls

- Compiler performs automatic type conversion independently, therefore it may have the following pitfalls:
  - Ambiguity with two classes of same type
  - Automatic conversion to more than one type - **fan-out**
  - Adds hidden activities (copy-constructor etc)

Unit Summary

In this unit you have covered …

- Operator overloading
- Different types of operator
- Operators that cannot be overloaded
- Inheritance and overloading
- Automatic type conversion