Object-Oriented Programming (OOP)
Lecture No. 16

Operator Overloading

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

Operator overloading

- Consider the following class:

```
class Complex{
private:
   double real, img;
public:
   Complex Add(const Complex &);
   Complex Subtract(const Complex &);
   Complex Multiply(const Complex &);
   …
}
```

Operator overloading

- **Function implementation:**

```
Complex Complex::Add(
      const Complex & c1){
   Complex t;
   t.real = real + c1.real;
   t.img  = img  + c1.img;
   return t;
}
```

Operator overloading

- The following statement:

```
Complex c3 = c1.Add(c2);
```

  Adds the contents of `c2` to `c1` and assigns it to `c3` (copy constructor)

Operator overloading

- To perform operations in a single mathematical statement e.g:

```
c1+c2+c3+c4
```

- We have to explicitly write:

```
c1.Add(c2.Add(c3.Add(c4)))
```

Operator overloading

- Alternative way is:

```
t1 = c3.Add(c4);
t2 = c2.Add(t1);
t3 = c1.Add(t2);
```

Operator overloading

- If the mathematical expression is big:
  - Converting it to C++ code will involve complicated mixture of function calls
  - Less readable
  - Chances of human mistakes are very high
  - Code produced is very hard to maintain

Operator overloading

- C++ provides a very elegant solution:

  "*Operator overloading*"

- C++ allows you to overload common operators like +, – or * etc…
- Mathematical statements don't have to be explicitly converted into function calls

Operator overloading

- Assume that operator **`+ has`** been overloaded
- Actual C++ code becomes:

```
c1+c2+c3+c4
```

- The resultant code is very easy to read, write and maintain

Operator overloading

- C++ automatically overloads operators for pre-defined types
- Example of predefined types:

```
int
float
double
char
long
```

Operator overloading

- Example:

```
float x;
int y;
x = 102.02 + 0.09;
Y = 50 + 47;
```

Operator overloading

The compiler probably calls the correct overloaded low level function for addition i.e:

```
// for integer addition:
Add(int a, int b)


// for float addition:
Add(float a, float b)
```

Operator overloading

- Operator functions are not usually called directly
- They are automatically invoked to evaluate the operations they implement

Operator overloading

- List of operators that can be overloaded in C++:

```
new     delete      new[]       delete[]
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      >>=     <<=     ==      !=
<=      >=      &&      ||      ++      --      ,       ->*     ->
()      []
```

Operator overloading

- List of operators that can't be overloaded:

```
.     .*     ::     ?:     #     ##
```

- Reason: They take name, rather than value in their argument except for **?:**
- **?:** is the only ternary operator in C++ and can't be overloaded

Operator overloading

- The precedence of an operator is **NOT** affected due to overloading
- Example:

  `c1*c2+c3`

  `c3+c2*c1`

  both yield the same answer

Operator overloading

- Associativity is **NOT** changed due to overloading
- Following arithmetic expression always is evaluated from left to right:

```
c1 + c2 + c3 + c4
```

Operator overloading

- Unary operators and assignment operator are right associative, e.g:

  `a=b=c` is same as `a=(b=c)`

- All other operators are left associative:

  `c1+c2+c3` is same as

  `(c1+c2)+c3`

Operator overloading

- Always write code representing the operator
- Example:

  Adding subtraction code inside the + operator will create chaos

Operator overloading

- Creating a new operator is a syntax error (whether unary, binary or ternary)
- You cannot create $

Operator overloading

- Arity of an operator is NOT affected by overloading
- Example:

  Division operator will take exactly two operands in any case:

```
b = c / d
```

Binary operators

- Binary operators act on two quantities
- Binary operators:

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	,	->*	->		

Binary operators

- General syntax:

  Member function:

```
TYPE1 CLASS::operator B_OP(
                TYPE2 rhs){
   ...
}
```

Binary operators

- General syntax:

  Non-member function:

```
TYPE1 operator B_OP(TYPE2 lhs,
                    TYPE3 rhs){
   ...
}
```

Binary operators

- The "`operator OP`" must have at least one formal parameter of type class (user defined type)
- Following is an error:

```
int operator + (int, int);
```

Binary operators

- Overloading + operator:

```
class Complex{
private:
   double real, img;
public:
   …
   Complex operator +(const
        Complex & rhs);
};
```

Binary operators

```
Complex Complex::operator +(
   const Complex & rhs){
   Complex t;
   t.real = real + rhs.real;
   t.img = img + rhs.img;
   return t;
}
```

Binary operators

- The return type is Complex so as to facilitate complex statements like:

```
Complex t = c1 + c2 + c3;
```

- The above statement is automatically converted by the compiler into appropriate function calls:

```
(c1.operator +(c2)).operator
+(c3);
```

Binary operators

- If the return type was **void**,

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

Binary operators

```
void Complex::operator+(const
     Complex & rhs){
   real = real + rhs.real;
   img = img + rhs.img;
};
```

Binary operators

- we have to do the same operation `c1+c2+c3` as:

```
c1+c2

c1+c3

// final result is stored in c1
```

Binary operators

- Drawback of void return type:
  - Assignments and cascaded expressions are not possible
  - Code is less readable
  - Debugging is tough
  - Code is very hard to maintain