Operator Overloading

Sumaiyah Zahid

Operator Overloading

- C++ allows you to specify more than one definition for an operator in the same scope, which is called operator overloading.
- You can redefine or overload most of the built-in operators available in C++
- It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.

Operator Overloading

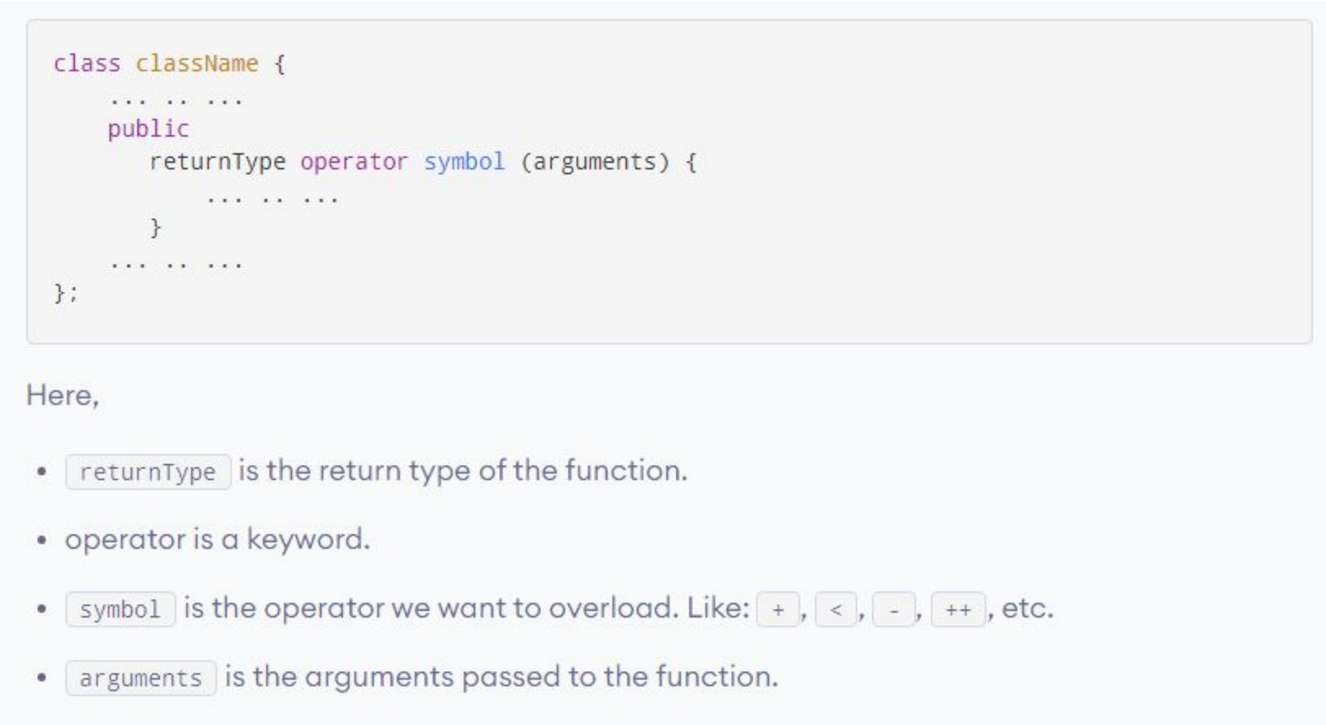
- Defining a new behavior for common operators of a language
- C++ enables you to overload most operators to be sensitive to the context in which they’re used
- Using operator overloading makes a program clearer than accomplishing the same operations with function calls

Operator Overloading

- An operator is overloaded by writing a non-static member function definition or global function definition
- When operators are overloaded as member functions, they must be non-static
- To use an operator on class objects (as operands), that operator **“must” be overloaded**

Operator Overloading

- Operator overloading cannot change the arity of an operator
- Operator overloading works when *at least* one argument of that operator is an object
- We cannot create new operators using operator overloading

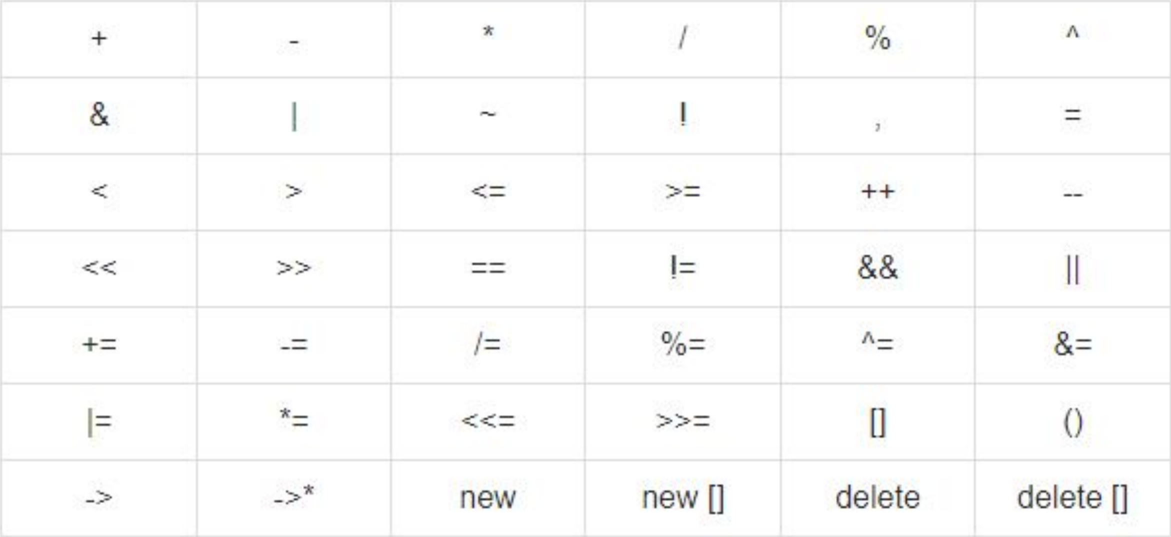


```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Here,

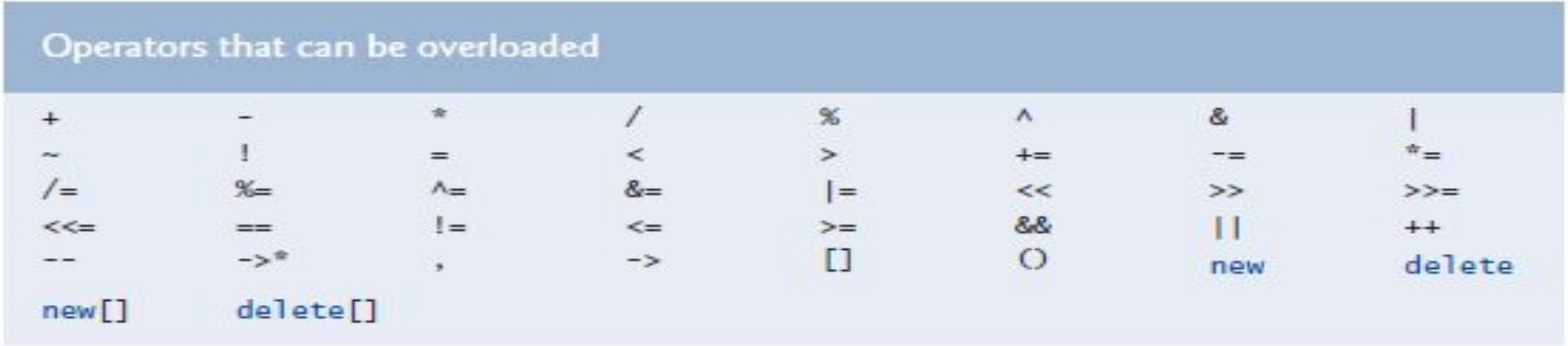
- `returnType` is the return type of the function.
- operator is a keyword.
- `symbol` is the operator we want to overload. Like: `+`, `<`, `-`, `++`, etc.
- `arguments` is the arguments passed to the function.

Overloadable Operators



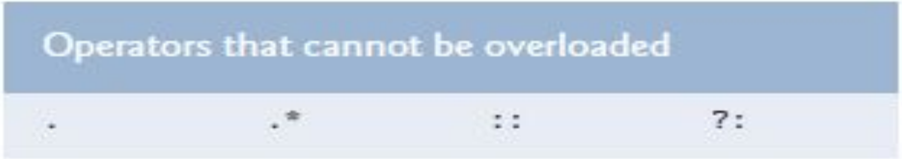
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Restrictions on Operator Overloading



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

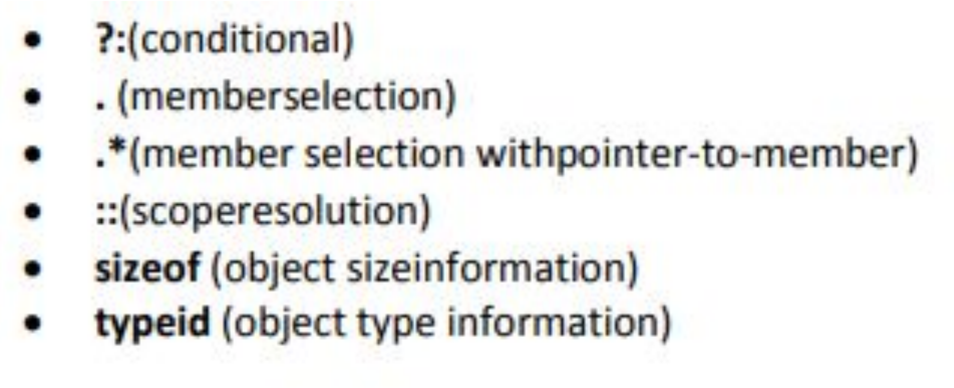
Operators that can be overloaded.



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

Operators that cannot be overloaded.

Non-Overloadable Operators



- **?:**(conditional)
- **.** (memberselection)
- **.***(member selection withpointer-to-member)
- ::(scoperesolution)
- **sizeof** (object sizeinformation)
- **typeid** (object type information)

BUILT IN OVERLOADS

Most operators are already overloaded for fundamental types.

Example:

1) In the case of the expression: a / b the operand type determines the machine code created by the compiler for the division operator. If both operands are integral types, an integral division is performed; in all other cases floating-point division occurs. Thus, different actions are performed depending on the operand types involved.

2) <<, which is used both as the stream insertion operator and as the bitwise left-shift operator.

Works fine

```
#include <iostream>
using namespace std;
int main() {
    int a=5;
    int b=3;
    int z=a+b;
    cout << z;
     return 0;
}
```

Output?

```
#include <iostream>
using namespace std;
 class Complex {
   private:
      int real;
      int image;
   public:
Complex(){
         real = 0;
         image = 0; }
Complex(int r, int i){
         real = r;
         image = i;}
```

```
void displayComplex() {
 cout << "real: "<< real << "
Imaginary:" <<image <<endl;
}
};
int main() {
   Complex c1(2,1);
   Complex c2(3,1);
   Complex c3;
   c3=c1+c2;
    return 0;}
```

Output

```
[Error] no match for 'operator+' (operand types are
'Complex' and 'Complex')
```

CRITERIA/RULES TO DEFINE THE OPERATOR FUNCTION:

In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.

Example of Operator Overloading

```
class Complex {
   private:
      int real;
      int image;
   public:
Complex(){
         real = 0;
         image = 0; }
Complex(int r, int i){
         real = r;
         image = i;}
```

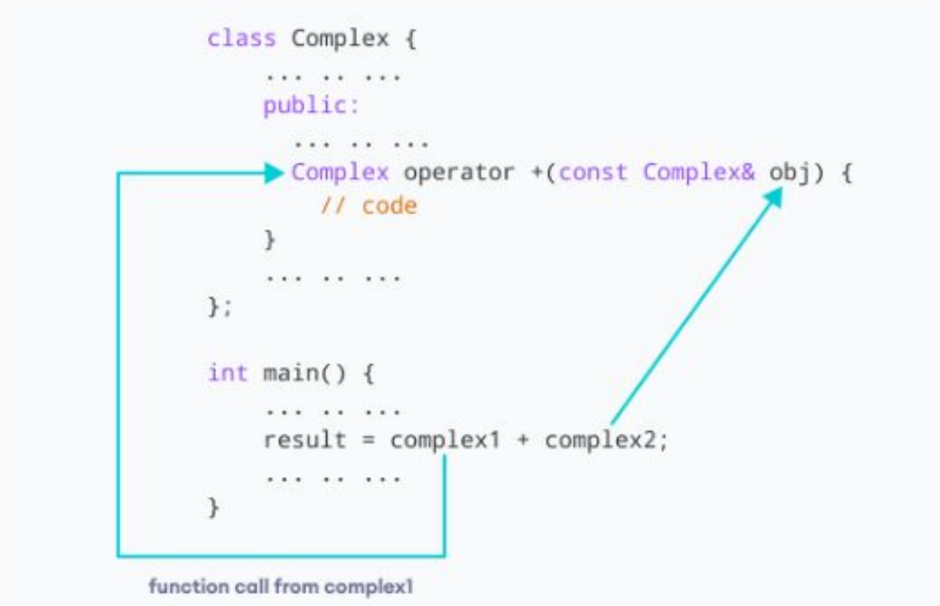
Example of Operator Overloading

```
void displayComplex() {
 cout << "real: "<< real << " Imaginary:" <<image <<endl;     }

Complex operator+ (Complex c) {
        Complex temp;
         temp.real=real+c.real;
         temp.image=image+c.image;
         return temp;
      }


};
```

Example of Operator Overloading

```
int main() {
    Complex c1(2,1);
    Complex c2(3,1);
    Complex c3;
          c3 = c1.operator+ ( c2 );
           //c3=c1+c2;
          c3.displayComplex();
    return 0;}
```

Binary Operator



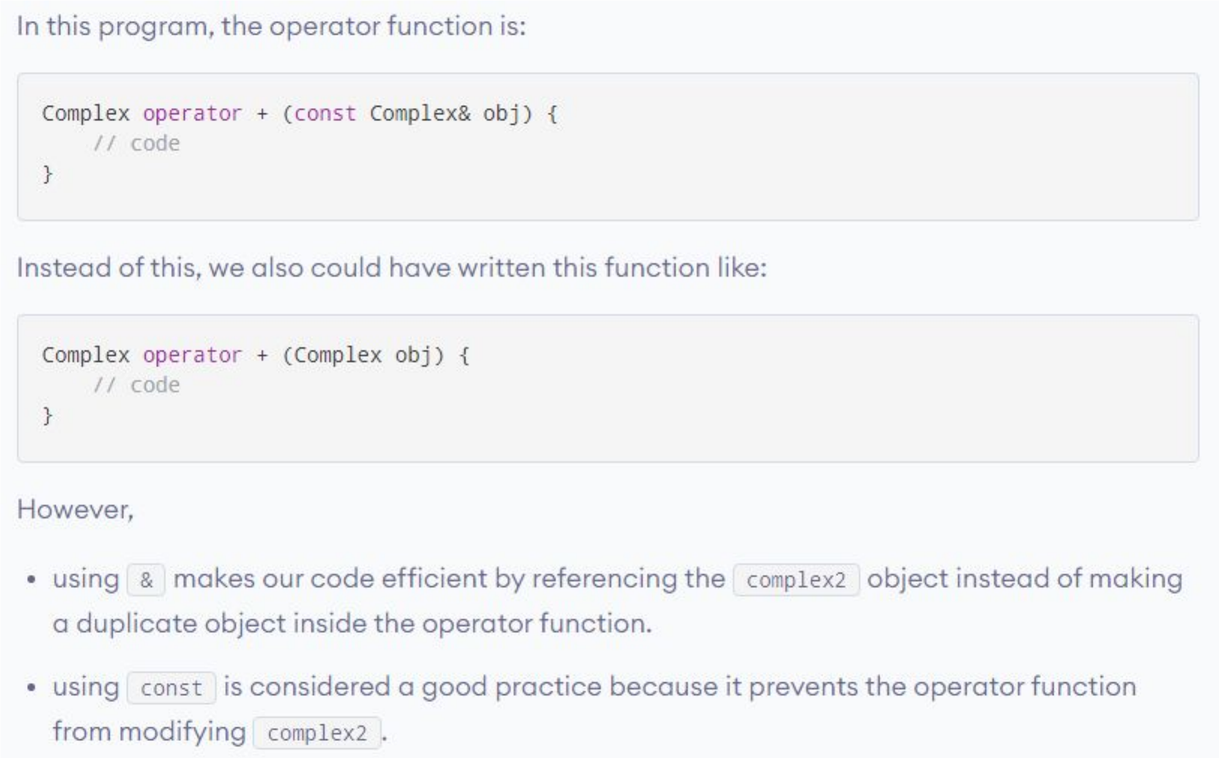
```
class Complex {
    ... .. ...
    public:
     ... .. ...
     Complex operator +(const Complex& obj) {
       // code
    }
    ... .. ...
};

int main() {
    ... .. ...
    result = complex1 + complex2;
    ... .. ...
}
```

Binary Operator Example

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

class Complex {
   private:
    float real;
    float imag;

   public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    void input() {
        cout << "Enter real and imaginary parts”
        cin >> real;
        cin >> imag;
    }

    // Overload the + operator
    Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
```

```
 void output() {
    if (imag < 0)
      cout << "Output Complex number: " << real <<"-"<< imag << "i";
    else
      cout << "Output Complex number: " << real << "+" << imag << "i";
  }
};

int main() {
   Complex complex1, complex2, result;

   cout << "Enter first complex number:\n";
   complex1.input();

   cout << "Enter second complex number:\n";
   complex2.input();

  // complex1 calls the operator function
  // complex2 is passed as an argument to the function
   result = complex1 + complex2;
   result.output();

   return 0;
}
```

Binary Operator



In this program, the operator function is:

```
Complex operator + (const Complex& obj) {
    // code
}
```

Instead of this, we also could have written this function like:

```
Complex operator + (Complex obj) {
    // code
}
```

However,

- using `&` makes our code efficient by referencing the `complex2` object instead of making a duplicate object inside the operator function.
- using `const` is considered a good practice because it prevents the operator function from modifying `complex2`.

For Prefix ++ operator

```
void operator ++ ( )
{
  ++x;
  ++y;
}
```

(Works the same way for prefix decrement operator)

For Prefix ++ operator

```
class Prefix{
       int i;
    public:
       Prefix(): i(0) {  }
       void operator ++()
          { ++i; }
       void Display()
          { cout << "i=" << i << endl; }};
int main(){
    Prefix obj;
    obj.Display();
    ++obj;
    //you can also write  obj.operator ++();
    obj.Display();
     return 0;}
```

For Postfix ++ Operator

```
Vector operator ++ ( int )
{
  Vector temp;
  temp.x = x++;
  temp.y = y++;
  return temp;
}
```

For Postfix ++ Operator

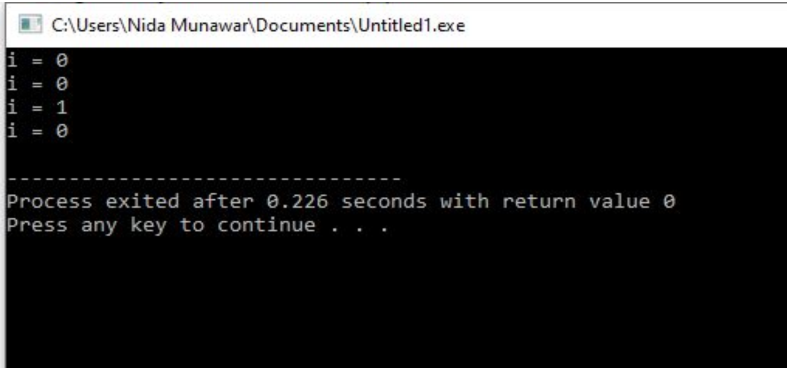
```
class Postfix{
       int i;
    public:
       Postfix(): i(0) {  }
       void operator ++(int)
          { i++; }
       void Display()
          { cout << "i=" << i << endl; }};
int main(){
    Postfix obj;
    obj.Display();
    obj++;
    //you can also write obj.operator ++(4);
    obj.Display();
    return 0;}
```

Output?

```
class Check{
    int i;
  public:
    Check(): i(0) {  }
    Check operator ++ (int){
        Check temp;
        temp.i = i++;
        return temp;}
void Display()
    { cout << "i = "<< i <<endl; }};
```

OUTPUT?

```
int main(){
    Check obj, obj1;
    obj.Display();
    obj1.Display();
    obj1 = obj++;
    obj.Display();
    obj1.Display();
    return 0;}
```



```
i = 0
i = 0
i = 1
i = 0

--------------------------------
Process exited after 0.226 seconds with return value 0
Press any key to continue . . .
```

Unary Operator Example

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

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

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches
<<endl;
      }
```

```
// overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return *this;
    }
     Distance operator++ () {
      ++feet;
      ++inches;
      return *this;
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
  D1.displayDistance();    // display D1

  ++D2;                  // apply D2.displayDistance();    // display D2

  return 0;
}
```

For += operator

```
Distance operator += (const Distance &ob)
{
    feet += ob.feet;
    inches += ob.inches;
    return *this;
}
```

Class Activity

1. Write a class Time which represents time. The class should have three fields for hours, minutes and seconds. It should have constructor to initialize the hours, minutes and seconds. A function print Time() to print the current time. Overload the following operators:

   - Plus operator (+) to add two time objects based on 24-hour clock.
   - Operator< to compare two time objects.

Home Task

1. Complete the following tasks:
   a. Design a Meal class with two fields—one that holds the name of the entrée, the other that holds a calorie count integer. Include a constructor that sets a Meal’s fields with parameters, or uses default values when no parameters are provided.
   b. Include an overloaded operator+()function that allows you to add two or more Meal objects. Adding two Meal objects means adding their calorie values and creating a summary Meal object in which you store “Daily Total” in the entrée field.
   c. Write a main()function that declares three Meal objects named breakfast, lunch, dinner, and total. Provide values for the breakfast, lunch, and dinner objects. Include the statement total = breakfast + lunch + dinner; in your program, then display values for the three Meal objects.
   d. Write a main()function that declares an array of 21 Meal objects. Allow a user to enter values for 21 Meals for the week. Total these meals and display the calorie total for the end of the week. (Hint: You might find it useful to create a constructor for the Meal class.)

References

https://www.programiz.com/cpp-programming/operator-overloading

Unary Operators Overloading

Binary Operators Overloading

Relational Operators Overloading

++ and -- Operators Overloading

Assignment Operators Overloading