Basic Concepts

C++ lets us define what the operators mean when applied to objects of class type. It also lets us define conversions for class types.

- overloaded operators are funtions with special names: the keyword `operator` followed by the symbol for the operator being defined. Like any other function, an overload operator has a return type, a parameter list, and a body.

```
Foo & operator=(const Foo &);
Foo & operator-(const Foo &);
//...
```

An overloaded operator function has the same number of parameters as the operator has operands. That is, a unary operator has one parameter; a binary operator has two. In a binary operator, the left-hand operand is passed to the first parameter and the right-hand operand to the second. *Except for the overloaded function-call operator, `operator()`, an overloaded operator may not have default arguments.*

> If an operator function is a number function, the first (left-hand) operand is bound to the implicit `this` pointer. Because the first operand is implicitly bound to `this`, a member operator function has one less (explicit) parameter than the operator has operands.

An operator function must either be a member of a class or have at least one parameter of class type.

```
// error, cannot redefine the built-in operator for ints
int operator+(int,int);
```

This restriction means that we cannot change the meaning of an operator when applied to operands of built-in type.

We can overload most, but not all, of the operators. The follow table shows whether or not an operator may be overloaded.

Operators that cannot be overloaded:

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

We can overload only existing operators and cannot invent new operator symbols. For example, we cannot define `operator**` to provide exponentiation.

It is worth noting that (`+`,`-`,`*`,and `&`) serve as both unary and binary operators. Either or both of these operators can be overloaded. The number of parameters determines which operator is being defined.

```
+ represents positive sign
- negative sign
* dereference
& address
```

An overloaded operator has the same precedence and associativity as the corresponding built-in operator. E.g., regradless of the operand types

```
x==x+y;// always equivalent to x==(x+y).
```

Calling an Overloaded Operator Function Directly

Ordinarily, we "call" an overloaded operator function indirectly by using the operator on arguments of the appropriate type. However, we can also call an overloaded function directly in the same way that we call an ordinarily function. We name the function and pass an appropriate number of arguments of the appropriate type:

```
// for nonmember operator function
class Foo{
    // members of Foo
    int val;
    Foo operator+(const Foo &f,const Foo &f2);// return type is an rvalue
};
Foo Foo::operator+(const Foo &f,const Foo &f2){
    Foo temp;
    temp.val=f.val+f2.val;
    return temp;
}
Foo f1,f2,f3;

// equivalent calls
f3=f1+f2;
f3=operator+(f1,f2);
```

We call a member operator function explicitly in the same way that we call any other member function. We name an object (or pointer) on which to run the function and use the dot (or arrow) operator to fetch the function we wish to call:

```
data1+=data2;// expression-based "call", operator+=(const data &).
data1.operator+=(data2);// equivalent call to a member operator function
```

Each of these statements calls the member function `operator+=`, bind `this` to the address of `data1` and passing `data2` as an argument.

Use Definitions That Are Consistent with the built-in Meaning

Those operations with a logical mapping to an operator are good candidates for defining as overloaded operators. It is worth noting that,

- if the class does IO, define the shift operators to be consistent with how IO is done for the built-in types.
- if the class has an operation to test for equality, define `operator==`. If the class has `operator==`, it should usually have `operator!=` as well.
- if the class has a single, natural ordering operation, define `operator<`. If the class has `operator<`, it should probably have all of the relational operators.
- the return type of an overloaded operator usually should be copatible with the return from the built-in version of the operator: the logical and relational operators should return `bool`, the arithmetic operators should return a value of the class type, and the assignment and compund assignment should return a reference to the left-hand operand.

Assignment and Compound Assignment Operators

Assignment operators should behave analogously to the systhesized operators: After an assignment, the values in the left-hand and right-hand operators should have the same value, and the operators should return a reference to its left-hand operand. Overloaded assignment should generalize the built-in meaning of assignment, not circumvent it.

If a class has an arithmetic or bitwise operator, the it is usually a good idea to provide the corresponding compound-assignment operator as well. That is,

```
+    +=
-    -=
>>   >>=
<<   <<=
```

Choosing Member or Nonmember Implementation

When we define an overloaded operator, we must decide whether to make the operator a class member or an ordinary nonmember function. In some cases, there is no choice-some operators are required to be members; in other cases, we may not be able to define the operator appropriately if it is a member.

The following guidellines can be of help in deciding whether to make an operator a member or an ordinary nonmember function:

- the assginment `=`, subscript `[]`, call `()`, and member access arrow `->` operators must be defined as members.
- the compound-assignment operators ordinarily ought to be members. However, unlike assignment, they are not required to be members.
- operators that change the state of their object or that are closely tied to their given type-such as increment, decrement, and dereference-usually should be members.
- symmetric operators-those that might convert either operand, such as the arithmetic, equility, relational, and bitwise operators-usually should be defined as ordinary nonmember functions.
    - Why? For example, we can add an `int` and a `double`. The addition is symmetric because we can use either type.

```
int a=3;
double b=3.14;
// equivalent
a+b;
b+a;
```

It is easily see that if we want to provide similar mixed-type expressions involving class objects, then the operator must be defined as a nonmember function.

```
class Foo;
class Bar;
T operator+(const Foo &f,const Bar &);// nonmember function
```

When we define an operator as a member function, then the left-hand operand must be an object of the class of which that operator is a member. For example:

```
string s="world";
string t=s+"~"; // ok, we can add a const char * to a string

string u="hi"+s;// would be an error if + were a member of string
// string u=s.operator+("hi"); // equivalent to the previous one

string u="hi".operator(s);// error, "hi" is not a string type, and cannot call the
member operator function
```

But we can define a *nonmember* operator function, like as follows:

```
string operator+(const char *cstr,const string &s){
    string t=s.operator+(cstr);
    return t;
}
```

calling this function will be

```
string u=operator("hi",s);
// or
string u="hi"+s;
```