2.8 Member Operator Overloading

Sue is working on her Date class but can't seem to get the assignment operator working. Why is that? Of all the operators, the assignment seems to be the most useful (with the exception of the insertion and the extraction operator). While investigating the cause of this obvious oversite, she stumbles upon member operator overloading.

Objectives

By the end of this chapter, you will be able to:

- Implement operator loading as non-member, non-member with friends, or as member functions
- List which operators should and must be overloaded as member functions
- Be able to use either the prefix or the infix notation for an overloaded operator

Prerequisites

Before reading this chapter, please make sure you are able to:

- Explain the differences between infix and prefix function notation (Chapter 2.6)
- Create a function to implement non-member operator overloading (Chapter 2.6)

What is member overloading and why you should care

Member operator overloading is the process of using infix notation of methods. This is similar to non-member operator overloading except the left-hand-side part of the equation is always `this`. It also allows operator functions to access member variables without using the `friend` modifier.

Most operators can be overloaded using non-members or as members. There are a few operators that can only be overloaded as non-members and a few that can only be overloaded as members. The non-member variety are those whose left-hand-side is not `this`. The classic examples of this are the insertion and extraction operators. For the insertion operator the left-hand-side is an `ostream` object; for the extraction operator the left-hand-side is an istream object.

The assignment operator can only be overloaded as a member function. The same is true with the array index operator also known as the square bracket operator (`[]`). That being said, the following rules-of-thumb are applied when considering whether to make an operator member or non-member:

- Member if it is a unary operator (like the negative (`-`), not (`!`), dereference (`*`), address-of- (`&`), increment (`++`) or decrement (`--`)).
- Non-member if both operands are treated equally and are left unchanged (like addition (`+`), multiplication (`*`), equivalence (`== !=`), comparison (`> >= < <=`), and (`&&`) and or (`||`)).
- Member if the binary operator does not treat both operands equally (like assignment (`=`), add-onto (`+= -= *= /= %= &= !=`), and array index (`[]`)).

Syntax overview

The syntax of member operator overloading is identical to non-member with two important distinctions. First, the function is part of a class. Therefore it is defined as a member function, either prototyped in the class definition or completely defined as inline in the class definition. Second, the left-hand-side parameter is hidden. Recall from Chapter 2.1 that a function modifying an object needs to pass the object as a parameter. In the case of member functions, this parameter is hidden and given the name `this`. The same is true with member operator overloading. .

The following operators **must** be overloaded as **member** functions:

Operator	Example	Use
assignment	`a = b`	Assign the value of b onto a
square bracket	`a[b]`	Retrieve an item from a collection
function call	`a(b)`	Can take any number of parameters, used similarly to non-default constructors

The following operators **must** be overloaded as **non-member** functions:

Operator	Example	Use
insertion	`a << b`	When the left-hand-side is an `ostream` object
extraction	`a >> b`	When the left-hand-side is an `istream` object
addition	`a + b`	When the returned data type is not the same as the left-hand-side data type
multiplication	`a * b`	When the returned data type is not the same as the left-hand-side data type

The following **should** be overloaded as **member** functions:

Operator	Example	Use
add onto	`a += b`	Right-hand-side can be any data type
subtract from	`a -= b`	Right-hand-side can be any data type
multiply onto	`a *= b`	Right-hand-side can be any data type
divide by	`a /= b`	Right-hand-side can be any data type
modulus from	`a %= b`	Right-hand-side can be any data type
increment	`a++  ++a`	Add one, postfix or prefix
decrement	`a--  --a`	Subtract one, postfix or prefix
negative	`-a`	Negative, opposite, disjoint set
not	`!a`	Logical not, opposite

The following **should** be overloaded as **non-member** functions

Operator	Example	Use
addition	`a + b`	Sum, add, adding onto, etc.
subtraction	`a - b`	Difference, subtraction, distance between, removing from, etc.
multiplication	`a * b`	Multiplying, extending, etc.
division	`a / b`	Division, dividing, reducing, etc.
modulus	`a % b`	Remainder
equivalence	`a == b`	Are they the same?
different	`a != b`	Are they different?
greater than	`a > b`	Is one larger than another?
greater than or equal to	`a >= b`	Is one larger or equal to another?
less than	`a < b`	Is one smaller than another?
less than or equal to	`a <= b`	Is one smaller or equal to another?
and	`a && b`	Logical "and", subset
or	`a || b`	Logic "or", superset

Assignment

Conceptually the assignment (=) operator works the same as the copy constructor with the exception that a new object is not created. Instead, an exact copy of the object on the left-hand-side is made to the object on the right-hand side. The one exception to this general rule is those times when the left-hand-side represents a different data type than the object on the right-hand-side. Be careful with those situations; make sure the overloading the assignment operator makes sense in that context.

Using the operator

With the assignment operator (as with the += operator), the left-hand-side gets changed:

```
{
   Complex c1(10.0, 9.6);                    // initialize to 10.0 + 9.6i
   Complex c2;                               // initially 0 + 0i
   Complex c3;                               // same here
   c3 = c2 = c1;                             // c2 assigned to 4.5 + 9.6i. Next
}                                            //    c3 is assigned to 4.5 + 9.6i
```

The right-hand-side of the assignment operator is the thing being copied. This means it should not change, thus being a constant. Since we should not make a copy of the right-hand-side, it additionally should be by-reference.

The left-hand-side of the assignment operator is being changed. Since the assignment operator must be a member function, it comes through as `this`. Since `this` changes, the method cannot be constant.

Finally, we need to be able to chain or stack assignment operators. The return value should be the newly changed left-hand-side. Thus the assignment operator in the above code can be re-written as:

```
c3 = (c2 = c1);
```

Prototype

The prototype for the assignment operator defined with the `Complex` class on the right-hand-side is:

The name of the function is "`Complex :: operator = ()`". We can call it with "`c1.operator=(c2)`" or "`c1 = c2`"

```
Complex & Complex :: operator = (const Complex & rhs);
```

Return the left-hand-side by-reference so we don't make a copy of `this`

The left-hand-side is `this` because we are overloading a member operator

The right-hand-side is an object constant by-reference so we don't change or make a copy of `rhs`

Implementation

The following is the implementation of the assignment operator for the `Complex` class:

```
Complex & Complex :: operator = (const Complex & rhs)
{
   this->r = rhs.r;                          // the left-hand-side of the operator
   this->i = rhs.i;                          //   comes in as "this"
   return *this;                             // return *this because "this" is
}                                            //    a pointer to a Complex
```

The most common mistake is to forget to return `*this`, thereby making it impossible to chain the assignment operator.

Square bracket

The square bracket operator (`[]`), also known as the array index operator, is commonly used to retrieve an item from a collection. The interesting thing about this operator, however, is that the parameter does not need to be an integer.

Using the operator

The square bracket operator is commonly used with containers (classes storing data). To demonstrate, we will start with the simple container class `Array`:

```
class Array
{
... code removed for brevity ...
   private:
      double * data;                    // where the data for the array is stored
      int      sizeArray;               // the number of items currently in the array
};
```

We can use the square-bracket operator both as an accessor and as a mutator:

```
{
   Array array(10);                     // initial capacity of 10
   array[0] = 9.9;                      // change the 1st item by setting it to 9.9
   cout << array[0] << endl;            // access the 1st item, the value 9.9
}
```

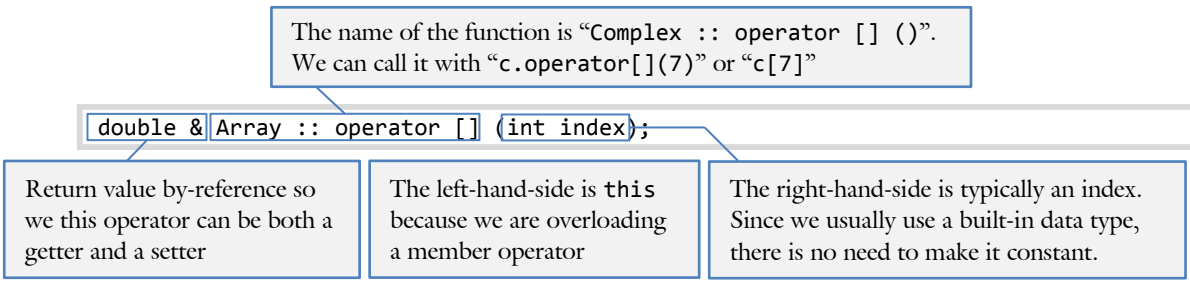
The right-hand-side of the square bracket operator traditionally is an integer, corresponding to the index of the item in the container. Note, however, that this can be any data type.

The left-hand-side is the class itself, represented as this. If you wish the square-bracket operator to function only as a getter, make `this` constant by making the method itself constant.

The return value is the data being accessed from the container. If the square bracket operator is to function as just a getter, either return by-value or a constant by-reference. If the square bracket operator is to function as both a getter and a setter, then the function must return by-reference.

Prototype

The prototype for the square-bracket operator defined with the `Array` class on the right-hand-side is:



Implementation

The following is the implementation of the square-bracket operator for the `Array` class:

```
double & Array :: operator [] (int index) throw(bool)
{
   if (index < 0 || index >= sizeArray)
      throw false;
   return data[index];
}
```

Function call

The function call operator `()` can also be overloaded as a member function. While this is not commonly done, it does have a few rather unusual benfits. First, it can be overloaded with any number of operands. Most operators are unary or binary, but the function call operator can be anything. The second use for the function call operator is how it works similarly to a non-default constructor. Recall that a constructor can take any number of parameters, serving to initialize the class with the passed values. The function call operator can look and work the same, except a new object is not created. In other words, it can be a `set()` function.

Using the operator

It is rare to overload the function operator. However, in those situations when initializing or setting a value to an object needs to occur frequently and with many parameters, it might be the most convenient tool for the job at hand.

```
{
   Complex c1(10.0, 9.6);                    // initialize to 10.0 + 9.6i
   c1(3.4, 2.1);                             // set the value to 3.4 + 2.1i
}
```

The left-hand-side is the object being operated on. Since we usually change the left-hand-side, the method is frequently not a constant.

There can be any number of parameters to the function call operator. Usually they come in as constant by-reference, but there is no hard-and-fast rule here.

The return value of the function call operator is usually `void`. That being said, it can be anything.

Prototype

The prototype for the square-bracket operator defined with the `Array` class on the right-hand-side is:

The name of the function is "`Complex :: operator () ()`". We can call it with "`c.operator()(3.4, 2.1)`" or "`c(3.4, 2.1)`"

```
void Complex :: operator () (float real, float imaginary);
```

Return value can be anything. In this case, the operator works like a setter

The left-hand-side is `this` because we are overloading a member operator

There can be any number of parameters here. In this case, we are sending to built-in parameters by-value

Implementation

The following is the implementation of the function-call operator for the `Complex` class:

```
void Complex :: operator () (float real, float imaginary = 0.0)
{
   this->r = real;
   this->i = imaginary;
}
```

Logical operators

The logical "and" (`&&`) and logical "or" (`||`) operators can be overloaded as with any other operator. The problem with overloading these operators originates in how they are usually used. Traditionally the "and" and "or" operators take a Boolean value on both the left-hand-side and the right-hand-side.

```
{
   bool value1 = false;
   bool value2 = true;

   bool logicalAnd = value1 && value2;    // set intersection ∩
   bool logicalOr  = value1 || value2;    // set union U
}
```

Note that we can also use "and" to mean set instersection (the list of elements two sets have in common) and "or" to mean union (the list of elements in either of the two sets).

Traditional functions

First, we will write a function to compute all the elements in common between two `Arrays`:

```
Array intersection(const Array & lhs, const Array & rhs) // return a new set, by value
{
   Array array;                           // the new Array to be returned

   for (int i = 0; i < lhs.size(); i++)   // go though all the items
      if (rhs.isMember(lhs.get(i)))       // if it is not present
         array += lhs.get(i);             // add it.

   return array;                          // return the new array by-value
}
```

Operator overloading

The "and" operator can be implemented as:

```
Array Array :: operator && (const Array & rhs) const
{
   Array array;                           // the new Array to be returned

   for (int i = 0; i < size(); i++)       // access size() or this->size()
      if (rhs.isMember(data[i]))          // access the private member variable data[]
         array += data[i];

   return array;                          // return the same way: by-value
}
```

The "or" operator (`||`) works in much the same way. While it is uncommon to want to overload the logical "and" or "or" operator, the union and intersection metaphor is commonly needed.

Member vs. non-member

As mentioned previously, most operators can be overloaded as a member or as a non-member. A few examples are:

Addition

Non-member

```
class Complex
{
... code removed for brevity ...
      inline friend Complex operator + (const Complex & lhs, const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         return Complex(lhs.r + rhs.r, lhs.i + rhs.i);
      }
};
```

Member

```
class Complex
{
... code removed for brevity ...
      Complex operator + (const Complex & rhs) const
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         return Complex(r + rhs.r, i + rhs.i);
      }
};
```

Observe how we have one less parameter because `lhs` becomes `*this`. Also, if the `lhs` parameter with the non-member function is a `const`, the corresponding method is a `const` with member operator overloading.

Add onto

Non-member

```
class Complex
{
... code removed for brevity ...
      inline friend Complex & operator += (Complex & lhs, const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         lhs.r += rhs.r;
         lhs.i += rhs.i;
         return lhs;
      }
};
```

Member

```
class Complex
{
... code removed for brevity ...
      Complex & operator += (const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         r += rhs.r;
         i += rhs.i;
         return *this;
      }
};
```

Because member functions have access to member variables without using the dot operator, the member variables `r` and `i` are directly accessible in the member version of the `+=` function. Also, since `this` changes in the member function, the method is not a `const`.

Increment prefix

Non-member

```
class Complex
{
... code removed for brevity ...
      inline friend Complex & operator ++ (Complex & c)
      {
         return c += 1.0;        // prefix ++ is exactly the same as += 1
      }
};
```

Member

```
class Complex
{
... code removed for brevity ...
      Complex & operator ++ ()
      {
         return *this += 1.0;        // prefix ++ is exactly the same as += 1
      }
};
```

In the non-member function, we returned "`c += 1.0;`". Since we have no '`c`' parameter, we have to use `*this` instead. Note that we could also have said "`return operator+=(1.0);`".

Comparison

Non-member

```
class Complex
{
... code removed for brevity ...
      inline friend bool operator == (const Complex & lhs, const Complex & rhs)
      {
         return (lhs.r == rhs.r && lhs.i == rhs.i);
      }
};
```

Member

```
class Complex
{
... code removed for brevity ...
      bool operator == (const Complex & rhs) const
      {
         return (r == rhs.r && i == rhs.i);
      }
};
```

In each of the above examples, less code needs to be written to implement the operator as a member.

Calling operator functions

Operator overloading is designed to make it easier and more convenient for the client to call a common function. That being said, C++ allows the programmer to call the function either with the infix notation or the prefix notation. Therefore, the following are equivalent:

```
int main()
{
   cout << "Hello world";
   return 0;
}
```

```
int main()
{
   operator << (cout, "Hello world");
   return 0;
}
```

Of course, one would never want to do this. However, it does more clearly illustrate how functions work. Consider the following code for the `Complex` class:

```
class Complex
{
... code removed for brevity ...
      inline friend Complex & operator += (Complex & lhs, const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         lhs.r += rhs.r;
         lhs.i += rhs.i;
         return lhs;
      }
};
```

In this example, it is possible to call the `+=` operator several ways:

```
{
   Complex c1(4, 5);
   Complex c2(6, 1);
   Complex c3(6, 0);

   c1 += c2 += c3;
}
```

```
{
   Complex c1(4, 5);
   Complex c2(6, 1);
   Complex c3(6, 0);

   operator += (c1, operator += (c2,
c3));
}
```

From this example, it becomes clear why the `+=` operator needs to return a `Complex` object by-reference. If it was lacking, it would be impossible to chain more than one `+=` operator on the same line.

Sam's Corner

It is very important to implement operator overloading so the functions work the way the client expects. If an unfamiliar metaphor is used or if the parameters are not implemented the way they do for the built-in data types, the whole purpose of operator overloading will be defeated: the class will be harder not easier to work with!

Example 2.8 – Complex

Demo

This example will demonstrate how to implement the common operators as member functions. In this example, every function that can be overloaded as a method will be, regardless of whether it is easier for the client to work. All the operators will be overloaded as `inline` functions.

Problem

Take the code from "Example 2.7 – Complex" and convert every possible non-member operator to a member operator.

Solution

In this example, all the methods are implemented as `inline`.

```
class Complex
{
... code removed for brevity ...
   public:
      Complex operator + (const Complex & rhs) const
      {
         return Complex(r + rhs.r, i + rhs.i);
      }
      Complex operator + (double rhs) const
      {
         return Complex(r + rhs, i);
      }
      // cannot be a method because the left-hand-side is not a Complex
      inline friend Complex operator + (int lhs, const Complex & rhs)
      {
         return rhs + lhs;
      }

      Complex & operator += (const Complex & rhs)
      {  // (a + bi) + (c + di) = (a + c) + (b + d)i
         r += rhs.r;
         i += rhs.i;
         return *this;
      }
      Complex & operator += (double rhs)
      {  // (a + bi) + c = (a + c) + bi
         r += rhs;
         return *this;
      }

      Complex & operator ++ ()
      {
         return *this += 1.0;
      }
      Complex operator ++ (int postfix)
      {
         Complex old(*this);
         *this += 1.0;
         return old;
      }
};
```

See Also

The complete solution is available at 2-8-complex.html or:

```
/home/cs165/examples/2-8-complex/
```

Example 2.8 – Card

Demo

This example will demonstrate how to implement the common operators as member functions. In this example, every function that can be overloaded as a method will be, regardless of whether it is easier for the client to work. All the operators will be overloaded as non-`inline` functions.

Problem

Take the code from "Example 2.7 – Card" and convert every possible non-member operator to a member operator.

Solution

In this example, all the methods are not implemented as `inline`.

```
class Card
{
... code removed for brevity ...
   // insertion and extraction operators
   friend std::ostream & operator << (std::ostream & out, const Card & card);
   friend std::istream & operator >> (std::istream & in,        Card & card);

   // increment and decrement ... only changing rank
   Card & operator ++ ();              // prefix
   Card & operator -- ();              // prefix
   Card operator ++ (int postfix);     // postfix
   Card operator -- (int postfix);     // postfix

   // change a card by adding or subtracting one
   Card operator + (const int input) const;
   friend Card operator + (const int input, const Card & lhs);
   Card operator - (const int input) const;
   Card & operator += (const int input);
   Card & operator -= (const int input);

   // assignment
   Card & operator = (const Card & rhs);
   Card & operator () (int iSuit, int iRank);

   // Relative comparison... only comparing rank
   bool operator >= (const Card & rhs) const;
   bool operator >  (const Card & rhs) const;
   bool operator <= (const Card & rhs) const;
   bool operator <  (const Card & rhs) const;

   // Absolute comparision... comparing both rank and suit
   bool operator == (const Card & rhs) const;
   bool operator != (const Card & rhs) const;
};
```

See Also

The complete solution is available at 2-8-card.html or:

```
/home/cs165/examples/2-8-card/
```

Example 2.8 – Array

Demo

This example will demonstrate how to use the array-index operator as well as use the `+=` operator to append, the `&&` operator to find set intersection, and the `||` operator to find set union.

Problem

Write a function to implement an Array. This will behave much like the STL vector class except it will append with the `+=` operator rather than `push_back()` and it will display the contents with the insertion operator.

Solution

In this example, all the methods are implemented as `inline`.

```
class Array
{
... code removed for brevity ...
   public:
      // access a given item for getting and setting
      double & operator [](int index) throw(bool)
      {
         if (index < 0 || index >= sizeArray)
            throw false;
         return data[index];
      }
      // push an item onto the back of the list
      Array & operator += (double value) throw(bool)
      {
         grow(sizeArray + 1);
         data[sizeArray++] = value;
         return *this;
      }
      // copy the contents of one Array onto another
      Array & operator = (const Array & rhs) throw(bool)
      {
         grow(rhs.capacity());
         sizeArray = 0;
         for (int i = 0; i < rhs.size(); i++)
            (*this) += rhs.get(i);
         return *this;
      }
      // grow the array to a given size
      void grow(int capacity) throw(bool);
      // set intersection
      Array operator && (const Array & rhs) const;
      // set union
      Array operator || (const Array & rhs) const;
      // fetch the size or capacity
      int size()     const { return sizeArray;     }
      int capacity() const { return capacityArray; }

   private:
      double * data;           // where the data for the array is stored
      int      sizeArray;      // the number of items currently in the array
      int      capacityArray;  // the capacity of the array
};
```

See Also

The complete solution is available at 2-8-array.html or:

```
/home/cs165/examples/2-8-array/
```

Review 1-8

What are the data types for the return value, left-hand-side, and right-hand-side for the following operators for the `Complex` class?

	Expression	Return value	Left-Hand-Side	Right-Hand-Side
1.	`lhs + rhs`			
2.	`lhs += rhs`			
3.	`lhs++`			
4.	`++lhs`			
5.	`lhs == rhs`			
6.	`lhs << rhs`			
7.	`lhs >> rhs`			
8.	`-lhs`			

Please see page 197 *for a hint.*

Problem 9

Given the following friend non-member operator overloading:

```
Time operator + (const Time & lhs, int secondsToAdd)
{
   return Time(lhs.secondsSinceMidnight + secondsToAdd);
}
```

Define the member version of the addition operator.

Please see page 194 *for a hint.*

Problem 10

Given the following friend non-member operator overloading:

```
bool operator == (const Time & lhs, const Time & rhs)
{
   return (lhs.secondsSinceMidnight == rhs.secondsSinceMidnight);
}
```

Define the member version of the equivalence operator.

Please see page 194 *for a hint.*

Problem 11

Write an assignment operator for the following class:

```
class Time
{
   public:
      Time()        : secondsSinceMidnight(0)   {}
      Time(int rhs) : secondsSinceMidnight(rhs) {}
      Time(const Time & rhs)
      {
         secondsSinceMidnight = rhs.secondsSinceMidnight;
      }
   private:
      int secondsSinceMidnight;
};
```

Please see page 190 *for a hint.*

Problem 12

Given the following `Position` structure:

```
struct Position
{
   int row;
   int col;
};
```

Given the following `TicTacToe` class:

```
class TicTacToe
{
```

... code removed for brevity ...

```
   private:
      char board[3][3];
};
```

And given the following method:

```
char & TicTacToe :: getPiece(const Position & pos)
{
   return board[pos.row][pos.col];
}
```

Write the array-index operator for the `TicTacToe` class.

Please see page 191 *for a hint.*