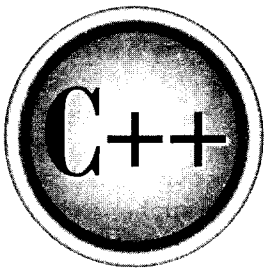
The Complete Reference



Chapter 15

Operator Overloading

Closely related to function overloading is operator overloading. In C++, you can overload most operators so that they perform special operations relative to classes that you create. For example, a class that maintains a stack might overload + to perform a push operation and -- to perform a pop. When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded.

The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.

You overload operators by creating operator functions. An *operator function* defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions. Therefore, each will be examined separately, beginning with member operator functions.

Creating a Member Operator Function

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
  // operations
}
```

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #. For example, if you are overloading the / operator, use **operator/.** When you are overloading a unary operator, *arg-list* will be empty. When you are overloading binary operators, *arg-list* will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.)

Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the + operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+( )**:

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
};

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1.show(); // displays 10 20
  ob2.show(); // displays 5 30

  ob1 = ob1 + ob2;
  ob1.show(); // displays 15 50

  return 0;
}
```

As you can see, **operator+( )** has only one parameter even though it overloads the binary + operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that **operator+( )** takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the **this** pointer. The operand on the right is passed in the parameter **op2**. The fact that the left operand is passed using **this** also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function.

As mentioned, it is common for an overloaded operator function to return an object of the class it operates upon. By doing so, it allows the operator to be used in larger expressions. For example, if the **operator+( )** function returned some other type, this expression would not have been valid:

```
ob1 = ob1 + ob2;
```

In order for the sum of **ob1** and **ob2** to be assigned to **ob1**, the outcome of that operation must be an object of type **loc**.

Further, having **operator+( )** return an object of type **loc** makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In this situation, **ob1+ob2** generates a temporary object that ceases to exist after the call to **show( )** terminates.

It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates.

One last point about the **operator+( )** function: It does not modify either operand. Because the traditional use of the + operator does not modify either operand, it makes sense for the overloaded version not to do so either. (For example, 5+7 yields 12, but neither 5 nor 7 is changed.) Although you are free to perform any operation you want inside an operator function, it is usually best to stay within the context of the normal use of the operator.

The next program adds three additional overloaded operators to the **loc** class: the –, the =, and the unary ++. Pay special attention to how these functions are defined.

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

class loc {
  int longitude, latitude;
```

```
public:
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2)
{
  loc temp;

  // notice order of operands
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Overload asignment for loc.
```

```
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // i.e., return object that generated call
}

// Overload prefix ++ for loc.
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);

  ob1.show();
  ob2.show();

  ++ob1;
  ob1.show(); // displays 11 21

  ob2 = ++ob1;
  ob1.show(); // displays 12 22
  ob2.show(); // displays 12 22

  ob1 = ob2 = ob3; // multiple assignment
  ob1.show(); // displays 90 90
  ob2.show(); // displays 90 90

  return 0;
}
```

First, examine the **operator–( )** function. Notice the order of the operands in the subtraction. In keeping with the meaning of subtraction, the operand on the right side of the minus sign is subtracted from the operand on the left. Because it is the object on the left that generates the call to the **operator–( )** function, **op2**'s data must be subtracted

from the data pointed to by **this**. It is important to remember which operand generates the call to the function.

In C++, if the = is not overloaded, a default assignment operation is created automatically for any class you define. The default assignment is simply a member- by-member, bitwise copy. By overloading the =, you can define explicitly what the assignment does relative to a class. In this example, the overloaded = does exactly the same thing as the default, but in other situations, it could perform other operations. Notice that the **operator=( )** function returns ***this**, which is the object that generated the call. This arrangement is necessary if you want to be able to use multiple assignment operations such as this:

```
ob1 = ob2 = ob3; // multiple assignment
```

Now, look at the definition of **operator++( ).** As you can see, it takes no parameters. Since ++ is a unary operator, its only operand is implicitly passed by using the **this** pointer.

Notice that both **operator=( )** and **operator++( )** alter the value of an operand. In the case of assignment, the operand on the left (the one generating the call to the **operator=( )** function) is assigned a new value. In the case of the ++, the operand is incremented. As stated previously, although you are free to make these operators do anything you please, it is almost always wisest to stay consistent with their original meanings.

Creating Prefix and Postfix Forms of the Increment and Decrement Operators

In the preceding program, only the prefix form of the increment operator was overloaded. However, Standard C++ allows you to explicitly create separate prefix and postfix versions of the increment or decrement operators. To accomplish this, you must define two versions of the **operator++( )** function. One is defined as shown in the foregoing program. The other is declared like this:

```
loc operator++(int x);
```

If the ++ precedes its operand, the **operator++( )** function is called. If the ++ follows its operand, the **operator++(int x)** is called and **x** has the value zero.

The preceding example can be generalized. Here are the general forms for the prefix and postfix ++ and – – operator functions.

```
// Prefix increment
type operator++( ) {
  // body of prefix operator
}
```

```
// Postfix increment
type operator++(int x) {
 // body of postfix operator
}

// Prefix decrement
type operator--() {
  // body of prefix operator
}

// Postfix decrement
type operator--(int x) {
 // body of postfix operator
}
```

Note *You should be careful when working with older C++ programs where the increment and decrement operators are concerned. In older versions of C++, it was not possible to specify separate prefix and postfix versions of an overloaded ++ or --. The prefix form was used for both.*

Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as +=, -=, and the like. For example, this function overloads += relative to **loc**:

```
loc loc::operator+=(loc op2)
{
  longitude = op2.longitude + longitude;
  latitude = op2.latitude + latitude;

  return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

Operator Overloading Restrictions

There are some restrictions that apply to operator overloading. You cannot alter the precedence of an operator. You cannot change the number of operands that an operator takes. (You can choose to ignore an operand, however.) Except for the function call

operator (described later), operator functions cannot have default arguments. Finally, these operators cannot be overloaded:

. :: .* ?

As stated, technically you are free to perform any activity inside an operator function. For example, if you want to overload the + operator in such a way that it writes **I like C++** 10 times to a disk file, you can do so. However, when you stray significantly from the normal meaning of an operator, you run the risk of dangerously destructuring your program. When someone reading your program sees a statement like **Ob1+Ob2**, he or she expects something resembling addition to be taking place—not a disk access, for example. Therefore, before decoupling an overloaded operator from its normal meaning, be sure that you have sufficient reason to do so. One good example where decoupling is successful is found in the way C++ overloads the << and >> operators for I/O. Although the I/O operations have no relationship to bit shifting, these operators provide a visual "clue" as to their meaning relative to both I/O and bit shifting, and this decoupling works. In general, however, it is best to stay within the context of the expected meaning of an operator when overloading it.

Except for the = operator, operator functions are inherited by a derived class. However, a derived class is free to overload any operator (including those overloaded by the base class) it chooses relative to itself.

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

In this program, the **operator+( )** function is made into a friend:

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

class loc {
  int longitude, latitude;
public:
```

```
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, loc op2); // now a friend
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2.longitude;
  temp.latitude = op1.latitude + op2.latitude;

  return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2)
{
  loc temp;

  // notice order of operands
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Overload assignment for loc.
```

```
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // i.e., return object that generated call
}

// Overload ++ for loc.
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1 = ob1 + ob2;
  ob1.show();

  return 0;
}
```

There are some restrictions that apply to friend operator functions. First, you may not overload the =, ( ), [ ], or -> operators by using a friend function. Second, as explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Using a Friend to Overload ++ or --

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. This is because friend functions do not have **this** pointers. Assuming that you stay true to the original meaning of the ++ and -- operators, these operations imply the modification of the operand they operate upon. However, if you overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed

a **this** pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call. However, you can remedy this situation by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call. For example, this program uses friend functions to overload the prefix versions of ++ and -- operators relative to the **loc** class:

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator=(loc op2);
  friend loc operator++(loc &op);
  friend loc operator--(loc &op);
};

// Overload assignment for loc.
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // i.e., return object that generated call
}

// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
```

```
  op.longitude++;
  op.latitude++;

  return op;
}

// Make op-- a friend; use reference.
loc operator--(loc &op)
{
  op.longitude--;
  op.latitude--;

  return op;
}

int main()
{
  loc ob1(10, 20), ob2;

  ob1.show();
  ++ob1;
  ob1.show(); // displays 11 21

  ob2 =  ++ob1;
  ob2.show(); // displays 12 22

  --ob2;
  ob2.show(); // displays 11 21

  return 0;
}
```

If you want to overload the postfix versions of the increment and decrement operators using a friend, simply specify a second, dummy integer parameter. For example, this shows the prototype for the **friend,** postfix version of the increment operator relative to **loc.**

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```

Friend Operator Functions Add Flexibility

In many cases, whether you overload an operator by using a friend or a member function makes no functional difference. In those cases, it is usually best to overload by using member functions. However, there is one situation in which overloading by using a friend increases the flexibility of an overloaded operator. Let's examine this case now.

As you know, when you overload a binary operator by using a member function, the object on the left side of the operator generates the call to the operator function. Further, a pointer to that object is passed in the **this** pointer. Now, assume some class that defines a member **operator+( )** function that adds an object of the class to an integer. Given an object of that class called **Ob**, the following expression is valid:

```
Ob + 100 // valid
```

In this case, **Ob** generates the call to the overloaded **+** function, and the addition is performed. But what happens if the expression is written like this?

```
100 + Ob // invalid
```

In this case, it is the integer that appears on the left. Since an integer is a built-in type, no operation between an integer and an object of **Ob**'s type is defined. Therefore, the compiler will not compile this expression. As you can imagine, in some applications, having to always position the object on the left could be a significant burden and cause of frustration.

The solution to the preceding problem is to overload addition using a friend, not a member, function. When this is done, both arguments are explicitly passed to the operator function. Therefore, to allow both *object+integer* and *integer+object*, simply overload the function twice—one version for each situation. Thus, when you overload an operator by using two **friend** functions, the object may appear on either the left or right side of the operator.

This program illustrates how **friend** functions are used to define an operation that involves an object and built-in type:

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

class loc {
  int longitude, latitude;
public:
```

```
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, int op2);
  friend loc operator+(int op1, loc op2);
};

// + is overloaded for loc + int.
loc operator+(loc op1, int op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2;
  temp.latitude = op1.latitude + op2;

  return temp;
}
// + is overloaded for int + loc.
loc operator+(int op1, loc op2)
{
  loc temp;

  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

  ob1.show();
```

```
  ob2.show();
  ob3.show();

  ob1 = ob2 + 10; // both of these
  ob3 = 10 + ob2; // are valid

  ob1.show();
  ob3.show();

  return 0;
}
```

Overloading new and delete

It is possible to overload **new** and **delete**. You might choose to do this if you want to use some special allocation method. For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted. Whatever the reason, it is a very simple matter to overload these operators.

The skeletons for the functions that overload **new** and **delete** are shown here:

```
// Allocate an object.
void *operator new(size_t size)
{
  /* Perform allocation.  Throw bad_alloc on failure.
     Constructor called automatically. */
  return pointer_to_memory;
}

// Delete an object.
void operator delete(void *p)
{
  /* Free memory pointed to by p.
     Destructor called automatically. */
}
```

The type **size_t** is a defined type capable of containing the largest single piece of memory that can be allocated. (**size_t** is essentially an unsigned integer.) The parameter **size** will contain the number of bytes needed to hold the object being allocated. This is the amount of memory that your version of **new** must allocate. The overloaded **new** function must return a pointer to the memory that it allocates, or throw a **bad_alloc** exception if an allocation error occurs. Beyond these constraints, the overloaded **new** function can do anything else you require. When you allocate an

object using **new** (whether your own version or not), the object's constructor is automatically called.

The **delete** function receives a pointer to the region of memory to be freed. It then releases the previously allocated memory back to the system. When an object is deleted, its destructor is automatically called.

The **new** and **delete** operators may be overloaded globally so that all uses of these operators call your custom versions. They may also be overloaded relative to one or more classes. Lets begin with an example of overloading **new** and **delete** relative to a class. For the sake of simplicity, no new allocation scheme will be used. Instead, the overloaded operators will simply invoke the standard library functions **malloc( )** and **free( ).** (In your own application, you may, of course, implement any alternative allocation scheme you like.)

To overload the **new** and **delete** operators for a class, simply make the overloaded operator functions class members. For example, here the **new** and **delete** operators are overloaded for the **loc** class:

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);
};

// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
  void *p;
```

```
    cout << "In overloaded new.\n";
    p =  malloc(size);
    if(!p) {
      bad_alloc ba;
      throw ba;
    }
    return p;
}

// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
  cout << "In overloaded delete.\n";
  free(p);
}

int main()
{
  loc *p1, *p2;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
  }

  p1->show();
  p2->show();

  delete p1;
  delete p2;

  return 0;
}
```

Output from this program is shown here.

```
In overloaded new.
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

When **new** and **delete** are for a specific class, the use of these operators on any other type of data causes the original **new** or **delete** to be employed. The overloaded operators are only applied to the types for which they are defined. This means that if you add this line to the **main( )**, the default **new** will be executed:

```
int *f = new float; // uses default new
```

You can overload **new** and **delete** globally by overloading these operators outside of any class declaration. When **new** and **delete** are overloaded globally, C++'s default **new** and **delete** are ignored and the new operators are used for all allocation requests. Of course, if you have defined any versions of **new** and **delete** relative to one or more classes, then the class-specific versions are used when allocating objects of the class for which they are defined. In other words, when **new** or **delete** are encountered, the compiler first checks to see whether they are defined relative to the class they are operating on. If so, those specific versions are used. If not, C++ uses the globally defined **new** and **delete**. If these have been overloaded, the overloaded versions are used.

To see an example of overloading **new** and **delete** globally, examine this program:

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

class loc {
int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }
    .
```

```
  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
};

// Global new
void *operator new(size_t size)
{
  void *p;

  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Global delete
void operator delete(void *p)
{
  free(p);
}

int main()
{
  loc *p1, *p2;
  float *f;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
```

```
  }

  try {
    f = new float; // uses overloaded new, too
  } catch (bad_alloc xa) {
    cout << "Allocation error for f.\n";
    return 1;;
  }

  *f = 10.10F;
  cout << *f << "\n";

  p1->show();
  p2->show();

  delete p1;
  delete p2;
  delete f;

  return 0;
}
```

Run this program to prove to yourself that the built-in **new** and **delete** operators have indeed been overloaded.

Overloading new and delete for Arrays

If you want to be able to allocate arrays of objects using your own allocation system, you will need to overload **new** and **delete** a second time. To allocate and free arrays, you must use these forms of **new** and **delete**.

```
// Allocate an array of objects.
void *operator new[](size_t size)
{
  /* Perform allocation.  Throw bad_alloc on failure.
     Constructor for each element called automatically. */
  return pointer_to_memory;
}

// Delete an array of objects.
void operator delete[](void *p)
```

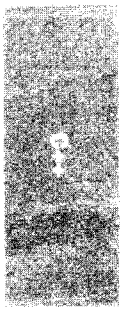
```
{
  /* Free memory pointed to by p.
     Destructor for each element called automatically.
  */
}
```

When allocating an array, the constructor for each object in the array is automatically called. When freeing an array, each object's destructor is automatically called. You do not have to provide explicit code to accomplish these actions.

The following program allocates and frees an object and an array of objects of type **loc.**

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

class loc {
  int longitude, latitude;
public:
  loc() {longitude = latitude = 0;}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);

  void *operator new[](size_t size);
  void operator delete[](void *p);
};

// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
```

```
void *p;

  cout << "In overloaded new.\n";
  p = malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
  cout << "In overloaded delete.\n";
  free(p);
}

// new overloaded for loc arrays.
void *loc::operator new[](size_t size)
{
  void *p;

  cout << "Using overload new[].\n";
  p = malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete overloaded for loc arrays.
void loc::operator delete[](void *p)
{
  cout << "Freeing array using overloaded delete[]\n";
  free(p);
}

int main()
{
  loc *p1, *p2;
```



```
  int i;

  try {
    p1 = new loc (10, 20); // allocate an object
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;;
  }

  try {
    p2 = new loc [10]; // allocate an array
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
  }

  p1->show();

  for(i=0; i<10; i++)
    p2[i].show();

  delete p1; // free an object
  delete [] p2; // free an array

  return 0;
}
```

Overloading the nothrow Version of new and delete

You can also create overloaded **nothrow** versions of **new** and **delete**. To do so, use these skeletons.

```
// Nothrow version of new.
void *operator new(size_t size, const nothrow_t &n)
{
  // Perform allocation.
  if(success) return pointer_to_memory;
  else return 0;
}

// Nothrow version of new for arrays.
```

```
void *operator new[](size_t size, const nothrow_t &n)
{
  // Perform allocation.
  if(success) return pointer_to_memory;
  else return 0;
}

void operator delete(void *p, const nothrow_t &n)
{
  // free memory
}

void operator delete[](void *p, const nothrow_t &n)
{
  // free memory
}
```

The type **nothrow_t** is defined in **<new>**. This is the type of the **nothrow** object. The **nothrow_t** parameter is unused.

Overloading Some Special Operators

C++ defines array subscripting, function calling, and class member access as operations. The operators that perform these functions are the **[ ]**, **( )**, and **->**, respectively. These rather exotic operators may be overloaded in C++, opening up some very interesting uses.

One important restriction applies to overloading these three operators: They must be nonstatic member functions. They cannot be **friend**s.

Overloading [ ]

In C++, the [ ] is considered a binary operator when you are overloading it. Therefore, the general form of a member **operator[ ]( )** function is as shown here:

```
type class-name::operator[](int i)
{
  //...
}
```

Technically, the parameter does not have to be of type **int**, but an **operator[ ]( )** function is typically used to provide array subscripting, and as such, an integer value is generally used.

Given an object called **O**, the expression

```
O[3]
```

translates into this call to the **operator[ ]( )** function:

```
O.operator[](3)
```

That is, the value of the expression within the subscripting operators is passed to the **operator[ ]( )** function in its explicit parameter. The **this** pointer will point to **O**, the object that generated the call.

In the following program, **atype** declares an array of three integers. Its constructor initializes each member of the array to the specified values. The overloaded **operator[ ]( )** function returns the value of the array as indexed by the value of its parameter.

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

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int operator[](int i) { return a[i]; }
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // displays 2

  return 0;
}
```

You can design the **operator[ ]( )** function in such a way that the [ ] can be used on both the left and right sides of an assignment statement. To do this, simply specify the

return value of **operator[ ]( )** as a reference. The following program makes this change and shows its use:

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

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i) { return a[i]; }
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // displays 2
  cout << " ";

  ob[1] = 25; // [] on left of =

  cout << ob[1]; // now displays 25

  return 0;
}
```

Because **operator[ ]( )** now returns a reference to the array element indexed by **i,** it can be used on the left side of an assignment to modify an element of the array. (Of course, it may still be used on the right side as well.)

One advantage of being able to overload the [ ] operator is that it allows a means of implementing safe array indexing in C++. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded [ ] subscripting operator, then you can intercept an out-of-range index. For example, this program adds a range check to the preceding program and proves that it works:

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i);
};

// Provide range checking for atype.
int &atype::operator[](int i)
{
  if(i<0 || i> 2) {
    cout << "Boundary Error\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // displays 2
  cout << " ";

  ob[1] = 25; // [] appears on left
  cout << ob[1]; // displays 25

  ob[3] = 44; // generates runtime error, 3 out-of-range

  return 0;
}
```

In this program, when the statement

```
ob[3] = 44;
```

executes, the boundary error is intercepted by **operator[]( )**, and the program is terminated before any damage can be done. (In actual practice, some sort of error-handling function would be called to deal with the out-of-range condition; the program would not have to terminate.)

Overloading ( )

When you overload the **( )** function call operator, you are not, per se, creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters. Let's begin with an example. Given the overloaded operator function declaration

```
double operator()(int a, float f, char *s);
```

and an object **O** of its class, then the statement

```
O(10, 23.34, "hi");
```

translates into this call to the **operator( )** function.

```
O.operator()(10, 23.34, "hi");
```

In general, when you overload the **( )** operator, you define the parameters that you want to pass to that function. When you use the **( )** operator in your program, the arguments you specify are copied to those parameters. As always, the object that generates the call (**O** in this example) is pointed to by the **this** pointer.

Here is an example of overloading **( )** for the **loc** class. It assigns the value of its two arguments to the longitude and latitude of the object to which it is applied.

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

class loc {
  int longitude, latitude;
public:
```

```
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator()(int i, int j);
};

// Overload ( ) for loc.
loc loc::operator()(int i, int j)
{
  longitude = i;
  latitude = j;

  return *this;
}

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;
  return temp;
}

int main()
{
  loc ob1(10, 20), ob2(1, 1);

  ob1.show();
  ob1(7, 8); // can be executed by itself
  ob1.show();
```

```
  ob1 = ob2 + ob1(10, 10); // can be used in expressions
  ob1.show();

  return 0;
}
```

The output produced by the program is shown here.

```
10 20
7 8
11 11
```

Remember, when overloading **( )**, you can use any type of parameters and return any type of value. These types will be dictated by the demands of your programs. You can also specify default arguments.

Overloading ->

The -> pointer operator, also called the *class member access* operator, is considered a unary operator when overloading. Its general usage is shown here:

object->element;

Here, *object* is the object that activates the call. The **operator->( )** function must return a pointer to an object of the class that **operator->( )** operates upon. The *element* must be some member accessible within the object.

The following program illustrates overloading the -> by showing the equivalence between **ob.i** and **ob->i** when **operator->( )** returns the **this** pointer:

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

class myclass {
public:
  int i;
  myclass *operator->() {return this;}
};

int main()
{
  myclass ob;
```

```
  ob->i = 10; // same as ob.i

  cout << ob.i << " " << ob->i;

  return 0;
}
```

An **operator–>( )** function must be a member of the class upon which it works.

Overloading the Comma Operator

You can overload C++'s comma operator. The comma is a binary operator, and like all overloaded operators, you can make an overloaded comma perform any operation you want. However, if you want the overloaded comma to perform in a fashion similar to its normal operation, then your version must discard the values of all operands except the rightmost. The rightmost value becomes the result of the comma operation. This is the way the comma works by default in C++.

Here is a program that illustrates the effect of overloading the comma operator.

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator,(loc op2);
```

```
};

// overload comma for loc
loc loc::operator,(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude;
  temp.latitude = op2.latitude;
  cout << op2.longitude << " " << op2.latitude << "\n";

  return temp;
}

// Overload + for loc
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);

  ob1.show();
  ob2.show();
  ob3.show();
  cout << "\n";

  ob1 = (ob1, ob2+ob2, ob3);

  ob1.show(); // displays 1 1, the value of ob3

  return 0;
}
```

This program displays the following output:

```
10 20
5 30
1 1

10 60
1 1
1 1
```

Notice that although the values of the left-hand operands are discarded, each expression is still evaluated by the compiler so that any desired side effects will be performed.

Remember, the left-hand operand is passed via **this**, and its value is discarded by the **operator,( )** function. The value of the right-hand operation is returned by the function. This causes the overloaded comma to behave similarly to its default operation. If you want the overloaded comma to do something else, you will have to change these two features.