第15章　运算符重载

与函数重载密切相关的是运算符重载。在C++中，可以重载绝大多数运算符，使它们完成和创建的类有关的特殊操作。例如，一个维护堆栈的类，重载+以执行入栈操作，重载-以执行出栈操作。运算符重载时，不会失去它原来的意义。相反，它适用的对象类型得到了拓宽。

重载运算符的能力是C++最强大的特征之一。它允许把新类完全集成到编程环境中。在重载了合适的运算符后，可以在表达式中使用对象，就像使用C++的内嵌数据类型一样。运算符重载也形成了C++ I/O方法的基础。

通过创建运算符函数，可以重载运算符。运算符函数定义了重载运算符将要执行的操作，该操作与它作用的类有关。运算符函数使用关键字operator创建。运算符函数既可以是一个类的成员，也可以不是一个类的成员。但非成员运算符函数几乎总是该类的友元函数。在成员和非成员函数之间，运算符函数的书写方法是不同的。下面我们从成员运算符函数开始一一介绍。

15.1　创建成员运算符函数

成员运算符函数的一般格式是：

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

通常，运算符函数返回它们操作的类的对象，但ret-type可以是任何有效的类型。#是一个占位符。创建运算符函数时，用运算符代替#。例如，如果正在重载运算符/，应使用operator/。如果重载一元运算符，则arg-list为空。如果重载二元运算符，arg-list将含有一个参数（这种情况看上去不正常，等一会儿就会清楚了）。

下面是第一个简单的运算符重载的例子。该程序创建了一个类loc，用来存储经度和纬度值。它重载和loc类相关的“+”运算符。仔细阅读这个程序，特别要注意operator+()的定义：

```
#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;
}
```

可以看出，虽然operator+( )重载的是二元运算符+，但它只有一个参数（你可能期望有两个参数与二元运算符的两个操作数相对应）。operator+( )只带一个参数的原因是：+左边的操作数是用this指针隐式地传给函数的。右边的操作数通过参数op2传递。左边的操作数用this指针传递的事实指出了一个重点：当重载二元运算符时，是左边的对象产生了对运算符函数的调用。

如前所述，重载运算符函数返回它所操作的类的对象，这是很常见的。这样，它允许运算符用在较大的表达式中。例如，如果operator+( )函数返回某个其他类型，下面的表达式是无效的：

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

为了把ob1，ob2之和赋给ob1，该操作的结果必须是loc类型的对象。

此外，只有operator+( )返回loc类型的对象，下面的语句才是可能的：

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

在这种情况下，ob1+ob2产生了一个临时对象。这个临时对象在show( )调用结束时便不复存在了。

重要的是要弄清楚运算符函数可以返回任何类型，并且返回的类型只依赖于特定的应用。通常的情况是，运算符函数将返回它操作的类的对象。

关于operator+( )的最后一点是：它不能修改两个操作数中的任何一个。由于运算符+的传统用法不能修改两个操作数中的任何一个，因此重载版本也应该如此（例如，5+7等于12，但

5和7都不会改变)。虽然可以让运算符函数执行我们所希望的任何操作，但最好是让运算符函数执行其通常的操作。

下面的程序给类loc增加了三个附加的重载运算符：-，=和一元运算符++。要特别注意这些函数是如何定义的。

```
#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;
}
```

首先看operator-( )函数。注意减法中操作数的顺序。为保持减法的含义，我们把减号右边的操作数从左边的操作数中减去。由于是左边的对象产生对operator-( )函数的调用，所以op2的数据必须从this所指的数据中减去，重要的是要记住是哪个操作数产生了对函数的调用。

在C++中，如果不重载=，对用户定义的任何类都将自动创建默认的赋值操作。默认的赋值操作就是简单的一个成员接一个成员的按位复制。通过重载=，可以显式地定义关于类的赋值做些什么。在本例中，重载的=和默认时做相同的事情，但在其他情况下，它可以执行其他操作。注意，operator=( )函数返回*this，*this是生成这个调用的对象。如果希望使用如下所示的多个赋值语句，这样安排是必要的：

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

现在，让我们看一看operator++( )的定义。可以看出，它没带任何参数。因为++是一元运算符，所以它的惟一的操作数是通过this指针隐式地传递的。

注意，operator=( )和operator++( )都改变操作数的值。对于赋值运算，左边的操作数（产生对operator=( )函数的调用的操作数）被赋予一个新的值；对于++运算，操作数增1。以前已经讲过，虽然可以随意让这些函数做你要求的事情，但保持它们原来的含义总是明智的。

15.1.1　创建增量和减量运算符的前缀和后缀形式

在前面的程序中，仅仅增量运算符的前缀形式得到了重载。然而，标准C++允许显式地创建增量和减量运算符的前缀和后缀形式。要做到这一点，必须定义两种形式的operator++( )函数。一种如前面的程序所示，另一种声明如下：

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

如果 ++ 位于操作数之前，就调用operator++( )函数，如果 ++ 位于操作数之后，就调用operator++(int x)，且 x 为 0。

可以对上面的例子进行归纳。下面是 ++、-- 运算符函数前缀和后缀的一般形式：

```
// 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
}
```

注意： 当使用老版本的C++程序时，若涉及到增量和减量运算符，应该小心些。在老版本的C++中，不能分别指定一个重载的 ++ 或 - - 的前缀和后缀形式。二者都使用前缀形式。

15.1.2　重载缩写运算符

用户可以重载C++的任何“缩写”运算符，比如 +=，-=，等等。例如，下面的函数重载关于loc的 +=：

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

  return *this;
}
```

当重载这些运算符之一时，记住，你正在把一个赋值和另一种类型的操作简单地组合在一起。

15.1.3　运算符重载限制

运算符重载也有一些限制：用户不能改变运算符的优先级，也不能改变运算符所带的操作数的个数（但可以有选择地省略一个操作数）。除了函数调用运算符外（后面讨论），运算符函

数不能有默认变元。最后，下面的运算符不能重载：

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

这说明，从技术上讲，用户可以自由地在运算符函数中执行任何活动。例如，如果想重载+运算符，从而向一个磁盘文件写10次“I like C++”，你可以这样做。然而，如果脱离运算符原义太远，就会冒使程序造成混乱的危险。例如，当别人读我们的程序时，发现像ob1+ob2这样的语句，他会期望类似于加法的某种操作，而不是磁盘存取等。所以，在使重载运算符脱离原义之前，必须保证有充分的理由这样做。C++重载<<和>>运算符用于输入输出就是一个成功的例子。尽管I/O操作和移位操作没有多大关系，但这些运算符提供了关于I/O和移位的意义和这种分离工作的直观线索。然而，通常在重载运算符时最好保持其原义。

除了=运算符之外，运算符函数可以被派生类继承。但派生类可以自由重载它选择的与自身相关的任何运算符（包括基类重载的运算符）。

15.2 使用友元函数的运算符重载

使用一个非成员函数（它通常是一个类的友元），可以重载一个类的运算符。因为友元函数不是类的成员，所以它没有this指针。因此，重载的友元运算符函数显式地传递操作数，这意味着重载二元运算符的友元函数有两个参数，而重载一元运算符的友元函数有一个参数。当使用友元函数重载二元运算符时，左操作数通过第一个参数传递，右操作数通过第二个参数传递。

下面的程序把operator+( )函数变为一个友元：

```
#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;
}
```

对友元运算符函数也有一些限制。第一，不能用友元函数重载=,(),[ ]或->运算符。第二，在重载增量或减量运算符时，若使用友元函数，则需要使用引用参数。

15.2.1　使用友元来重载 ++ 或 --

如果要用友元函数重载增量和减量运算符，必须把操作数作为引用参数传递。这是因为友元函数没有 this 指针。如果保持 ++ 和 -- 运算符的原义，那么这些操作就意味着要修改它们操作的操作数。然而，如果用一个友元重载这些运算符，那么操作数就当作参数通过值传递，这

意味着友元运算符函数无法修改操作数。因为友元运算符函数不给操作数传递this指针，而是传递操作数的副本，对参数的改变不会影响生成该调用的操作数。然而，通过指定传给友元运算符函数的参数为引用参数，可以改变这种状况，这就导致对函数内参数的任何改变都会影响生成该调用的操作数。例如，下面的程序使用友元函数重载与loc类有关的++和--运算符的前缀形式：

```
#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;
}
```

如果想要使用友元重载增量和减量运算符的后缀形式，只需简单地说明第二个空的整型参数。例如，下面的例子展示了使用友元的、与 loc 有关的增量运算符后缀形式的原型：

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

15.2.2　友元运算符函数增加了灵活性

在许多情况下，用友元还是成员函数重载运算符在功能上没有什么区别。在那些情况下，通常最好用成员函数重载。然而，有一种情况是，用友元重载可增加重载运算符的灵活性。让我们看看这种情况。

当用成员函数重载二元运算符时，运算符左边的对象产生对运算符函数的调用。进一步讲，指向该对象的指针通过 this 指针传递。现在，假设某类定义了成员 operator+( )函数，该函数把一个类的对象增加到一个整数中。如果那个类的一个对象称为 Ob，下面的表达式是有效的：

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

在这种情况下，Ob 产生了对重载 + 函数的调用，并执行加法操作。如果改写成如下表达式，会怎么样呢？

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

在这种情况下，整数出现在左边。因为整数是内嵌的类型，所以整数和 Ob 的对象之间没有定义操作。因此，编译器不会编译该表达式。可以想像，在某些应用中，总得把对象放在左边是一个沉重的负担，甚至会带来灾难。

解决这个问题的办法就是用友元而不是成员函数重载加法。这样，两个变元都被显式地传给运算符函数。所以，为了既允许 object+integer 又允许 integer+object 形式，只要简单地重载该函数两次——每种情况一种形式。因此，当用两个友元函数重载一个运算符时，对象既可以出现在运算符的左边，又可以出现在运算符的右边。

下面的程序说明如何使用友元函数来定义涉及对象和内嵌类型的操作。

```
#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;
}
```

15.3 重载new和delete

重载new和delete是可能的。如果要使用某种特殊的分配方法，就可以这样做。例如，我

们可能想要一些分配例程，这些分配例程在堆用完时自动开始使用磁盘空间作为虚存。无论是什么样的理由，重载这些运算符是一件简单的事情。

下面展示的是重载 new 和 delete 函数的架构。

```
// 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. */
}
```

类型 size_t 被定义为包含可分配的最大单片内存的类型（size_t 本质上是无符号整型）。参数 size 将包含容纳正被分配的对象所需的字节数。这是这种形式的 new 必须分配的内存数。重载的 new 函数必须返回指向它分配的内存的指针，并在分配发生错误时抛出一个 bad_alloc 异常。除此之外，重载的 new 函数可以做你要求的任何事情。当使用 new 分配对象时（不管是什么版本的 new），会自动调用对象的构造函数。

delete 函数接受一个指向要释放的内存区的指针，然后把先前分配的内存释放回系统。当删除对象时，自动调用它的析构函数。

new 和 delete 运算符可以全局重载，以便这些运算符的所有用法都调用你的自定义版本。它们也可以相对于一个或更多的类重载。下面我们从一个与类有关的重载 new 和 delete 的例子开始。为简单起见，不使用新的分配方案，相反，重载函数只简单地调用标准库函数 malloc( )和 free( )（在应用程序里，用户可以实现他们喜欢的任何分配方案）。

要为一个类重载 new 和 delete 运算符，只要简单地把重载运算符函数变成类成员。例如，下面为类 loc 重载 new 和 delete 运算符：

```
#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;
}
```

程序的输出如下所示：

```
In overloaded new.
```

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

在 new 和 delete 用于特定的类时，这两个运算符作用于任何类型的数据时都会用到原始的 new 和 delete。这些重载的运算符只适用于它们为之定义的类型。这意味着，如果把下面一行程序加入 main( )中，则会执行默认的 new：

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

可以通过在任何类声明之外重载 new 和 delete 运算符，从而全局重载 new 和 delete。当 new 和 delete 被全局重载时，可忽略 C++ 的默认 new 和 delete 并把新的运算符用于所有的分配请求。当然，如果定义了与一个或多个类相关的 new 和 delete 形式，那么当为所定义的类分配类对象时，将使用该类的特定形式。换句话说，当遇到 new 或 delete 时，编译器将首先检测相对于它们操作的类，它们是否已定义。如果没有定义，C++ 使用全局定义的 new 和 delete，否则，就使用重载形式。

下面的程序是全局重载 new 和 delete 的例子。

```
#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;
}
```

运行这个程序，以证明内嵌的new和delete运算符确实重载了。

15.3.1 重载数组的new和delete

如果希望使用自己的分配系统分配对象数组，则必须再次重载new和delete。要分配和释放数组，必须使用如下形式的new和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.
  */
}
```

分配数组时，数组中每个对象的构造函数将被自动调用。释放数组时，每个对象的析构函数将被自动调用，不必提供显式代码以完成这些功能。

下面的程序分配和释放了一个对象和一个对象数组，二者都是loc类型的。

```
#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;
}
```

15.3.2　重载 new 和 delete 的 nothrow 形式

也可以创建重载的 new 和 delete 的 nothrow 形式。要这样做，使用下面的架构：

```
// 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
}
```

类型 nothrow_t 在<new>中定义。这是 nothrow 类型的对象，没有使用 nothrow_t 参数。

15.4　重载某些特殊运算符

C++ 把数组下标、函数调用和类成员访问定义为运算。执行这些功能的运算符分别为[ ]，( )和 ->。这些特殊运算符可以在 C++ 中重载，以开发一些令人感兴趣的用途。

重载这三个运算符有一个重要的限制，即它们必须是非静态成员函数，不能是友元函数。

15.4.1　重载[ ]

在 C++ 中，重载[ ]时，把[ ]作为二元运算符使用。成员 operator[ ]( )函数的一般形式是：

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

从技术上讲，参数不一定是 int 型，但 operator[ ]( )函数典型地用于提供数组下标，为此，一般要使用整型值。

如果给定对象 O，表达式

```
O[ 3]
```

翻译成对operator[ ]( )函数的调用：

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

也就是说，下标运算符范围内的表达式的值通过显式参数传给operator[ ]( )函数，this指针指向产生这个调用的对象O。

在下面的程序里，atype声明一个包含三个整数的数组。它的构造函数初始化数组的每个成员为指定的值。重载的operator[ ]( )函数返回通过它的参数值索引的数组值。

```
#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;
}
```

可以设计operator[ ]( )函数，使[ ]既可用在赋值语句的左边又可用在右边。为此，只需简单地指定operator[ ]( )的返回值为一个引用即可。下面的程序做了这个改动并展示了它的用途：

```
#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;
}
```

由于operator[ ]( )现在返回一个其下标为i的数组元素的引用，所以它可以用在赋值语句的左边以修改数组元素（当然，也可以用在右边）。

重载[ ]运算符有一个好处，即提供了在C++中实现安全的数组下标的一种方法。我们知道，在C++中，程序运行时可能超过数组范围，但不产生运行时错误信息。然而，如果创建一个包含数组的类，并只允许通过重载的[ ]下标运算符访问那个数组，那么就能拦截越界下标。例如，下面的程序在前一个程序的基础上增加了范围检测功能并验证了它的作用。

```
// 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;
}
```

当该程序执行下面的语句时：

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

operator[]( )将拦截越界错误，且在造成任何损害之前程序终止（在实际应用中，调用某种错误处理函数来处理越界错误，程序不必终止）。

15.4.2 重载( )

当重载( )函数调用运算符时，实际上用户并不创建一种调用函数的新方法，而是创建可以传递任意多个参数的运算符函数。下面先看一个例子，如果给出重载运算符函数声明：

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

和它的类的对象O，那么语句

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

翻译成对 operator( )函数的调用。

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

通常，重载( )运算符时，用户定义了要传给该函数的参数。当在程序中使用( )运算符时，用户指定的变元被复制到这些参数中，而且 this 指针总是指向产生调用的对象（在本例中为O）。

下面是一个关于 loc 类重载( )的例子，它把两个变元的值赋给所应用对象的经度和纬度。

```
#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;
}
```

程序生成的输出如下所示。

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

记住：在重载( )时，可以使用任何类型的参数并返回任何类型的值。这些类型通过用户程序命令指定。也可以指定默认变元。

15.4.3　重载 ->

指针运算符 ->，也称为类成员访问运算符，在重载时被认为是一元运算符，它的一般用法如下：

object->element;

其中，object 是激活调用的对象。operator->( )函数必须返回一个 operator->( )操作的类对象的指针。element 必须是对象内可访问的某个成员。

下面的程序通过示范当 operator->( )返回 this 指针时 ob.i 和 ob->i 的等价性来介绍重载 -> 的方法。

```
#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;
}
```

所有operator->( )函数必须是它操作的类的成员。

15.5 重载逗号运算符

可以重载C++的逗号运算符。逗号是二元运算符，像所有重载的运算符一样，可以让重载的逗号执行任何期望的运算。但是，如果希望重载的逗号以和它通常的运算相似的方式工作，那么必须丢弃除了最右边操作数以外的所有操作数的值，并使最右边的值成为逗号运算的结果。这是C++中逗号的默认工作方法。

下面的程序演示了重载逗号运算符的效果。

```
#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;
}
```

该程序显示如下结果：

```
10 20
5 30
1 1
10 60
1 1
1 1
```

注意，虽然左边操作数的值被丢弃了，但编译器还是执行每个表达式以产生任何期望的效果。

记住，左边的操作数通过this传递，且operator( )函数丢弃了它的值。右边运算的值由函数返回。这使重载逗号的行为类似于它的默认操作。如果希望重载的逗号做些别的事情，就必须改变这两个特性。