第11章　运算符重载

运算符重载只是一种“语法修饰”，这意味着它是另一种调用函数的方法。

不同之处是对于函数的参数不是出现在圆括号内，而是在我们总认为是运算符的字符的附近。

但在C++中，可以定义一个和类一起工作的新运算符。除了这个名字函数以关键字operator开始，以运算符本身结束以外，这个定义和一个普通函数是一样的。这是仅有的差别。它像其他函数一样也是一个函数，当编译器看到它以适当的模式出现时，就调用它。

11.1　警告和确信

对于运算符重载，人们容易变得过于热心。首先，它是一个娱乐玩具。注意，它仅仅是一个语法修饰，是另外一种调用函数的方法而已。用这种眼光看，没有理由重载一个运算符，除非它会使包含我们的类的代码变得更易写，尤其是更易读。（记住，读代码的情况更多）如果不是这种情况，就不必麻烦去重载运算符。

对于运算符重载，另外一个通常的反映是恐慌：突然，C运算符不再有熟悉的意思。“所有的东西都改变了，我的所有C代码将做不同的事情！”但这不是事实。所有用于仅包含内部数据类型的表达式的运算符是不可能被改变的。我们永远不能重载下面的运算符使执行的行为不同。

```
1 << 4;
```

或者重载运算符使得下面的表达式有意义。

```
1.414 << 2;
```

仅仅是包含用户自定义类型的表达式可以有重载的运算符。

11.2　语法

定义一个重载运算符就像定义一个函数，只是该函数的名字是operator@，这里@代表运算符。函数参数表中参数的个数取决于两个因素：

1) 运算符是一元的（一个参数）还是二元的（两个参数）。

2) 运算符被定义为全局函数（对于一元是一个参数，对于二元是两个参数）还是成员函数（对于一元没有参数，对于二元是一个参数 — 对象变为左侧参数）。

这里有一个很小的类来显示运算符重载语法。

```
//: OPOVER.CPP -- Operator overloading syntax
#include <iostream.h>

class integer {
  int i;
public:
  integer(int I) { i = I; }
  const integer
```

```
  operator+(const integer& rv) const {
    cout << "operator+" << endl;
    return integer(i + rv.i);
  }
  integer&
  operator+=(const integer& rv){
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  integer I(1), J(2), K(3);
  K += I + J;
}
```

这两个重载的运算符被定义为内联成员函数。对于二元运算符，单个参数是出现在运算符右侧的那个。当一元运算符被定义为成员函数时，没有参数。成员函数被运算符左侧的对象调用。

对于非条件运算符（条件运算符通常返回一个布尔值），如果两个参数是相同的类型，希望返回和运算相同类型的对象或引用。如果它们不是相同类型，它作什么样的解释就取决于程序设计者。用这种方法可以组合复杂的表达式：

```
K += I + J;
```

运算符+号产生一个新的整数（临时的），这个整数被用作运算符‘ += ’的rv参数。一旦这个临时整数不再需要时就被消除。

11.3 可重载的运算符

虽然可以重载几乎所有C中可用的运算符，但使用它们是相当受限制的。特别地，不能结合C中当前没有意义的运算符（例如**求幂），不能改变运算符的优先级，不能改变运算符的参数个数。这样限制有意义——所有这些行为产生的运算符只会造成意思混淆而不是使之清楚。

下面两个小部分给出所有“经常用的”运算符的例子，这些被重载的运算符的形式会经常用到。

11.3.1 一元运算符

下面的例子显示了所有一元运算符重载的语法，它们既以全局函数形式又以成员函数形式表示。它们将扩充先前显示的类integer和加入新类byte。具体运算符的意思取决于如何使用它们。

```
//: UNARY.CPP -- Overloading unary operators
#include <iostream.h>
class integer {
  long i;
  integer* This() { return this; }
public:
  integer(long I = 0) : i(I) {}
  // No side effects takes const& argument:
  friend const integer&
    operator+(const integer& a);
  friend const integer
    operator-(const integer& a);
  friend const integer
    operator~(const integer& a);
  friend integer*
    operator&(integer& a);
  friend int
    operator!(const integer& a);
  // Side effects don't take const& argument:
  // Prefix:
  friend const integer&
    operator++(integer& a);
  // Postfix:
  friend const integer
    operator++(integer& a, int);
  // Prefix:
  friend const integer&
    operator--(integer& a);
  // Postfix:
  friend const integer
    operator--(integer& a, int);
};

// Global operators:
const integer& operator+(const integer& a) {
  cout << "+integer\n";
  return a; // Unary + has no effect
}
const integer operator-(const integer& a) {
  cout << "-integer\n";
  return integer(-a.i);
}
const integer operator~(const integer& a) {
  cout << "~integer\n";
  return integer(~a.i);
}
```

```
integer* operator&(integer& a) {
  cout << "&integer\n";
  return a.This(); // &a is recursive!
}
int operator!(const integer& a) {
  cout << "!integer\n";
  return !a.i;
}
// Prefix; return incremented value
const integer& operator++(integer& a) {
  cout << "++integer\n";
  a.i++;
  return a;
}
// Postfix; return the value before increment:
const integer operator++(integer& a, int) {
  cout << "integer++\n";
  integer r(a.i);
  a.i++;
  return r;
}
// Prefix; return decremented value
const integer& operator--(integer& a) {
  cout << "--integer\n";
  a.i--;
  return a;
}
// Postfix; return the value before decrement:
const integer operator--(integer& a, int) {
  cout << "integer--\n";
  integer r(a.i);
  a.i--;
  return r;
}

void f(integer a) {
  +a;
  -a;
  ~a;
  integer* ip = &a;
  !a;
  ++a;
  a++;
  --a;
  a--;
}
```

```
// Member operators (implicit "this"):
class byte {
  unsigned char b;
public:
  byte(unsigned char B = 0) : b(B) {}
  // No side effects: const member function:
  const byte& operator+() const {
    cout << "+byte\n";
    return *this;
  }
  const byte operator-() const {
    cout << "-byte\n";
  return byte(-b);
}
const byte operator~() const {
  cout << "~byte\n";
  return byte(~b);
}
byte operator!() const {
  cout << "!byte\n";
  return byte(!b);
}
byte* operator&() {
  cout << "&byte\n";
  return this;
}
// Side effects: non-const member function:
const byte& operator++() { // Prefix
  cout << "++byte\n";
  b++;
  return *this;
}
const byte operator++(int) { // Postfix
  cout << "byte++\n";
  byte before(b);
  b++;
  return before;
}
const byte& operator--() { // Prefix
  cout << "--byte\n";
  --b;
  return *this;
}
const byte operator--(int) { // Postfix
  cout << "byte--\n";
```

```
  byte before(b);
  --b;
  return before;
}
};

void g(byte b) {
  +b;
  -b;
  ~b;
  byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

main() {
  integer a;
  f(a);
  byte b;
  g(b);
}
```

根据参数传递的方法，将函数分组。如何传递和返回参数的方针在后面给出。上面的形式（和下一小节的形式）是典型的使用形式，所以当重载自己的运算符时可以以它们作为范式开始。

• 自增和自减

重载的++和--号运算符出现了两难选择的局面，这是因为希望根据它们出现在它们作用的对象前面（前缀）还是后面（后缀）来调用不同的函数。解决是很简单的，但一些人在开始时却发现它们容易令人混淆。例如当编译器看到 ++a（先自增）时，它就调用 operator++(a);但当编译器看到a++时，它就调用operator++(a,int)。即编译器通过调用不同的函数区别这两种形式。在UNARY.CPP成员函数版中，如果编译器看到 ++b，它就产生一个对B::perator++( )的调用；如果编译器看到b++，它就产生一个对B::operator++(int)的调用。

除非对于前缀和后缀版本用不同的函数调用，否则用户永远看不到它动作的结果。然而，实质上两个函数调用有不同的署名，所以它们和两个不同函数体相连。编译器为 int参数（因为这个值永远不被使用，所以它永远不会被赋给一个标识符）传递一个哑元常量值用来为后缀版产生不同的署名。

11.3.2 二元运算符

下面的清单是用二元运算符重复 UNARY.CPP的例子。全局版本和成员函数版本都在里面。

```
//: BINARY.CPP -- Overloading binary operators
#include <fstream.h>
#include "..\allege.h"
```

```
ofstream out("binary.out");

class integer { // Combine this with UNARY.CPP
  long i;
public:
  integer(long I = 0) : i(I) {}
  // Operators that create new, modified value:
  friend const integer
    operator+(const integer& left,
              const integer& right);
  friend const integer
    operator-(const integer& left,
              const integer& right);
  friend const integer
    operator*(const integer& left,
              const integer& right);
  friend const integer
    operator/(const integer& left,
              const integer& right);
friend const integer
  operator%(const integer& left,
            const integer& right);
friend const integer
  operator^(const integer& left,
            const integer& right);
friend const integer
  operator&(const integer& left,
            const integer& right);
friend const integer
  operator|(const integer& left,
            const integer& right);
friend const integer
  operator<<(const integer& left,
             const integer& right);
friend const integer
  operator>>(const integer& left,
             const integer& right);
// Assignments modify & return lvalue:
friend integer&
  operator+=(integer& left,
             const integer& right);
friend integer&
  operator-=(integer& left,
             const integer& right);
friend integer&
  operator*=(integer& left,
```

```
                const integer& right);
  friend integer&
    operator/=(integer& left,
                const integer& right);
  friend integer&
    operator%=(integer& left,
                const integer& right);
  friend integer&
    operator^=(integer& left,
               const integer& right);
  friend integer&
    operator&=(integer& left,
               const integer& right);
  friend integer&
    operator|=(integer& left,
               const integer& right);
  friend integer&
    operator>>=(integer& left,
                const integer& right);
  friend integer&
    operator<<=(integer& left,
                const integer& right);
  // Conditional operators return true/false:
  friend int
    operator==(const integer& left,
               const integer& right);
  friend int
    operator!=(const integer& left,
               const integer& right);
  friend int
    operator<(const integer& left,
              const integer& right);
  friend int
    operator>(const integer& left,
              const integer& right);
  friend int
    operator<=(const integer& left,
               const integer& right);
  friend int
    operator>=(const integer& left,
               const integer& right);
  friend int
    operator&&(const integer& left,
               const integer& right);
  friend int
    operator||(const integer& left,
```

```
                const integer& right);
  // Write the contents to an ostream:
  void print(ostream& os) const { os << i; }
};

const integer
  operator+(const integer& left,
            const integer& right) {
  return integer(left.i + right.i);
}
const integer
  operator-(const integer& left,
            const integer& right) {
  return integer(left.i - right.i);
}
const integer
  operator*(const integer& left,
            const integer& right) {
  return integer(left.i * right.i);
}
const integer
  operator/(const integer& left,
            const integer& right) {
  allege(right.i != 0, "divide by zero");
  return integer(left.i / right.i);
}
const integer
  operator%(const integer& left,
            const integer& right) {
  allege(right.i != 0, "modulo by zero");
  return integer(left.i % right.i);
}
const integer
  operator^(const integer& left,
            const integer& right) {
  return integer(left.i ^ right.i);
}
 const integer
   operator&(const integer& left,
             const integer& right) {
   return integer(left.i & right.i);
 }
 const integer
   operator|(const integer& left,
             const integer& right) {
```

```
  return integer(left.i | right.i);
}
const integer
  operator<<(const integer& left,
             const integer& right) {
  return integer(left.i << right.i);
}
const integer
  operator>>(const integer& left,
             const integer& right) {
  return integer(left.i >> right.i);
}
// Assignments modify & return lvalue:
integer& operator+=(integer& left,
                    const integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i += right.i;
   return left;
}
integer& operator-=(integer& left,
                    const integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i -= right.i;
   return left;
}
integer& operator*=(integer& left,
                    const integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i *= right.i;
   return left;
}
integer& operator/=(integer& left,
                    const integer& right) {
   allege(right.i != 0, "divide by zero");
   if(&left == &right) {/* self-assignment */}
   left.i /= right.i;
   return left;
}
integer& operator%=(integer& left,
                    const integer& right) {
   allege(right.i != 0, "modulo by zero");
   if(&left == &right) {/* self-assignment */}
   left.i %= right.i;
   return left;
}
integer& operator^=(integer& left,
```

```
                    const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i ^= right.i;
  return left;
}
integer& operator&=(integer& left,
                    const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i &= right.i;
  return left;
}
integer& operator|=(integer& left,
                    const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i |= right.i;
  return left;
}
integer& operator>>=(integer& left,
                     const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i >>= right.i;
  return left;
}
integer& operator<<=(integer& left,
                     const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i <<= right.i;
  return left;
}
// Conditional operators return true/false:
int operator==(const integer& left,
               const integer& right) {
    return left.i == right.i;
}
int operator!=(const integer& left,
               const integer& right) {
    return left.i != right.i;
}
int operator<(const integer& left,
              const integer& right) {
    return left.i < right.i;
}
int operator>(const integer& left,
              const integer& right) {
    return left.i > right.i;
}
```

```
int operator<=(const integer& left,
               const integer& right) {
    return left.i <= right.i;
}
int operator>=(const integer& left,
               const integer& right) {
    return left.i >= right.i;
}
int operator&&(const integer& left,
               const integer& right) {
    return left.i && right.i;
}
int operator||(const integer& left,
               const integer& right) {
    return left.i || right.i;
}

void h(integer& c1, integer& c2) {
  // A complex expression:
  c1 += c1 * c2 + c2 % c1;
  #define TRY(op) \
  out << "c1 = "; c1.print(out); \
  out << ", c2 = "; c2.print(out); \
  out << ";  c1 " #op " c2 produces ";\
  (c1 op c2).print(out); \
  out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Conditionals:
  #define TRYC(op) \
  out << "c1 = "; c1.print(out); \
  out << ", c2 = "; c2.print(out); \
  out << ";  c1 " #op " c2 produces ";\
  out << (c1 op c2); \
  out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

// Member operators (implicit "this"):
class byte { // Combine this with UNARY.CPP
  unsigned char b;
```

```
public:
  byte(unsigned char B = 0) : b(B) {}
  // No side effects: const member function:
 const byte
   operator+(const byte& right) const {
   return byte(b + right.b);
 }
 const byte
   operator-(const byte& right) const {
   return byte(b - right.b);
 }
 const byte
   operator*(const byte& right) const {
   return byte(b * right.b);
 }
 const byte
   operator/(const byte& right) const {
   allege(right.b != 0, "divide by zero");
   return byte(b / right.b);
 }
 const byte
   operator%(const byte& right) const {
   allege(right.b != 0, "modulo by zero");
   return byte(b % right.b);
 }
 const byte
   operator^(const byte& right) const {
   return byte(b ^ right.b);
 }
 const byte
   operator&(const byte& right) const {
   return byte(b & right.b);
 }
 const byte
   operator|(const byte& right) const {
   return byte(b | right.b);
 }
 const byte
   operator<<(const byte& right) const {
   return byte(b << right.b);
 }
const byte
  operator>>(const byte& right) const {
  return byte(b >> right.b);
}
```

```
// Assignments modify & return lvalue.
// operator= can only be a member function:
byte& operator=(const byte& right) {
  // Handle self-assignment:
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
byte& operator+=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b += right.b;
  return *this;
}
byte& operator-=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b -= right.b;
  return *this;
}
byte& operator*=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b *= right.b;
  return *this;
}
byte& operator/=(const byte& right) {
  allege(right.b != 0, "divide by zero");
  if(this == &right) {/* self-assignment */}
  b /= right.b;
  return *this;
}
byte& operator%=(const byte& right) {
  allege(right.b != 0, "modulo by zero");
  if(this == &right) {/* self-assignment */}
  b %= right.b;
  return *this;
}
byte& operator^=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b ^= right.b;
  return *this;
}
byte& operator&=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b &= right.b;
  return *this;
}
byte& operator|=(const byte& right) {
```

```
    if(this == &right) {/* self-assignment */}
    b |= right.b;
    return *this;
  }
  byte& operator>>=(const byte& right) {
    if(this == &right) {/* self-assignment */}
    b >>= right.b;
    return *this;
  }
  byte& operator<<=(const byte& right) {
    if(this == &right) {/* self-assignment */}
    b <<= right.b;
    return *this;
  }
  // Conditional operators return true/false:
  int operator==(const byte& right) const {
      return b == right.b;
  }
  int operator!=(const byte& right) const {
      return b != right.b;
  }
  int operator<(const byte& right) const {
      return b < right.b;
  }
  int operator>(const byte& right) const {
      return b > right.b;
  }
  int operator<=(const byte& right) const {
      return b <= right.b;
  }
  int operator>=(const byte& right) const {
      return b >= right.b;
  }
  int operator&&(const byte& right) const {
      return b && right.b;
  }
  int operator||(const byte& right) const {
      return b || right.b;
  }
  // Write the contents to an ostream:
  void print(ostream& os) const {
    os << "0x" << hex << int(b) << dec;
  }
};

void k(byte& b1, byte& b2) {
```

```
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(op) \
  out << "b1 = "; b1.print(out); \
  out << ", b2 = "; b2.print(out); \
  out << ";  b1 " #op " b2 produces ";\
  (b1 op b2).print(out); \
  out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(op) \
  out << "b1 = "; b1.print(out); \
  out << ", b2 = "; b2.print(out); \
  out << ";  b1 " #op " b2 produces ";\
  out << (b1 op b2); \
  out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Chained assignment:
  byte b3 = 92;
  b1 = b2 = b3;
}

main() {
  integer c1(47), c2(9);
  h(c1, c2);
  out << "\n member functions:" << endl;
  byte b1(47), b2(9);
  k(b1, b2);
}
```

可以看到运算符‘ = ’仅允许作为成员函数。这将在后面解释。

作为总的方针，我们注意到在运算符重载中所有赋值运算符都有代码用于核对自赋值 (self-assignment)。在一些情况下，这是不需要的。例如，可以用运算符‘ += ’写A+=A，使得A自身相加。最重要的核对自赋值的地方是运算符‘ = ’，因为复杂的对象可能因为它而发生灾难性

的结果（在一些情况下这不会有问题，不管怎么说，在写运算符‘ = ’时，应该小心一些）。

先前的两个例子中的运算符重载处理单一类型。也可能重载运算符处理混合类型，所以可以“ 把苹果加到橙子里 ”。然而，在开始进行运算符重载之前，应该看一下本章后面有关自动类型转换部分。经常在正确的地方使用类型转换可以减少许多运算符重载。

11.3.3 参数和返回值

开始看UNARY.CPP和BINARY.CPP例子时会发现参数传递和返回方法完全不同，这似乎有点令人混淆。虽然可以用任何想用的方法传递和返回参数，但这些例子方法却不是随便选择的。它们遵守一种非常合乎逻辑的模式，我们在大部分情况下都应选择这种模式：

1) 对于任何函数参数，如果仅需要从参数中读而不改变它，缺省地应当按 const引用来传递它。普通算术运算符（像+和-号等）和布尔运算符不会改变参数，所以以const引用传递是使用的主要方式。当函数是一个类成员的时候，就转换为 const成员函数。只是对于会改变左侧参数的赋值运算符(operator-assignment，像+=）和运算符‘ = ’，左侧参数才不是常量(constant)，但因为参数将被改变，所以参数仍然按地址传递。

2) 应该选择的返回值取决于运算符所期望的类型。（可以对参数和返回值做任何想做的事）如果运算符的效果是产生一个新值，将需要产生一个作为返回值的新对象。例如，integer::operator+必须生成一个操作数之和的 integer对象。这个对象作为一个 const通过传值方式返回，所以作为一个左值结果不会被改变。

3) 所有赋值运算符改变左值。为了使得赋值结果用于链式表达式（像 A=B=C），应该能够返回一个刚刚改变了的左值的引用。但这个引用应该是 const还是nonconst呢？虽然我们是从左向右读表达式 A=B=C，但编译器是从右向左分析这个表达式，所以并非一定要返回一个 nonconst值来支持链式赋值。然而人们有时希望能够对刚刚赋值的对象进行运算，例如（A=B）.foo（ ），这是B赋值给A后调用foo( )。因此所有赋值运算符的返回值对于左值应该是nonconst引用。

4) 对于逻辑运算符，人们希望至少得到一个 int返回值，最好是bool返回值。（在大多数编译器支持C++内置bool类型之前开发的库函数使用int或typedef等价物）。

5) 因为有前缀和后缀版本，所以自增和自减运算符出现了两难局面。两个版本都改变对象，所以不能把这个对象看作一个 const。因此，前缀版本返回这个对象被改变后的值。这样，用前缀版本我们只需返回*this作为一个引用。因为后缀版本返回改变之前的值，所以被迫创建一个代表这个值的单个对象并返回它。因此，如果想保持我们的本意，对于后缀必须通过传值方式返回。（注意，我们经常会发现自增和自减运算返回一个 int 值或bool值，例如用来指示是否有一个循环子(iterator)在表的结尾）。现在问题是：这些应该按const被返回还是按nonconst被返回？如果允许对象被改变，一些人写了表达式(++A).foo( )，则foo( )作用在A上。但对于表达式(A++).foo( )，foo( )作用在通过后缀运算符 ++号返回的临时对象上。临时对象自动定为 const，所以被编译器标记。但为了一致性，使两者都是 const更有意义，就像这儿所做的。因为想给自增和自减运算符赋予各种意思，所以它们需要就事论事考虑。

1. 按const通过传值方式返回

按const通过传值方式返回，开始看起来有些微妙，所以值得多加解释。我们来考虑二元运算符+号。假设在一个表达式像 f(A+B)中使用它，A+B的结果变为一个临时对象，这个对象用于f( ) 调用。因为它是临时的，自动被定为const，所以无论使返回值为const还是不这样做都没有影响。

然而，也可能发送一个消息给A+B的返回值而不是仅传递给一个函数。例如，可以写表达式(A+B).g( )，这里g( )是integer的成员函数。通过设返回值为 const，规定了对于返回值只有const成员函数才可以被调用。用 const是恰当的，这是因为这样可以防止在很可能丢失的对象中存贮有价值的信息。

2. 返回效率

当为通过传值方式返回而创建一个新对象时，要注意使用的形式。例如用运算符 +号：

```
return integer (left.i + right.i) ;
```

一开始看起来像是一个“ 对一个构造函数的调用 ”，但其实并非如此。这是临时对象语法，它是这样陈述的：“ 创建一个临时对象并返回它 ”。因为这个原因，我们可能认为如果创建一个命名的本地对象并返回它结果将会是一样的。其实不然。如果像下面这样表示，将发生三件事。首先，tmp对象被创建，与此同时它的构造函数被调用。然后，拷贝构造函数把 tmp拷贝到返回值外部存储单元里。最后，当tmp在作用域的结尾时调用析构函数。

```
integer tmp(left.i + right.i) ;
return tmp ;
```

相反，“ 返回临时对象 ”的方法是完全不同的。看这样情况时，编译器明白对创建的对象没有其他需求，只是返回它，所以编译器直接地把这个对象创建在返回值外面的内存单元。因为不是真正创建一个局部对象，所以仅需要单个的普通构造函数调用（不需要拷贝构造函数），并且不会调用析构函数。因此，这种方法不需要什么花费，效率是非常高的。

11.3.4 与众不同的运算符

有些其他的运算符对于重载语法有明显的不同。

下标运算符‘ [ ] ’必须是成员函数并且它需要单个参数。因为它暗示对象像数组一样动作，可以经常从这个运算符返回一个引用，所以它可以被很方便地用于等号左侧。这个运算符经常被重载；可以在本书其他部分看到有关的例子。

当逗号出现在逗号运算对象左右时，逗号运算符被调用。然而，逗号运算符在函数参数表中出现时不被调用，此时，逗号仅在对象中起分割作用。除了使语言保持一致性外，这个运算符似乎没有许多实际用途。这儿有一个例子用于显示当逗号出现在对象前面以及后面时，逗号函数是如何被调用的：

```
//: COMMA.CPP -- Overloading operator,
#include <iostream.h>

class after {
public:
  const after& operator,(const after&) const {
    cout << "after::operator,()" << endl;
    return *this;
  }
};

class before {};

before& operator,(int, before& b) {
```

```
  cout << "before::operator,()" << endl;
  return b;
}

main() {
  after a, b;
  a, b;  // Operator comma called

  before c;
  1, c;  // Operator comma called
}
```

全局函数允许逗号放在被讨论的对象的前面。这里的用法显得相当晦涩和令人怀疑。虽然，可以用逗号分割作为表达式的一部分，但它太敏感以至于在大多数情况下不能使用。

运算符( )的函数调用必须是成员函数，它是唯一的允许在它里面有任意个参数的函数。这使得对象看起来像一个真正的函数名，因此，它最好为仅有单一运算的类型使用，或至少是特别优先的一种类型。

运算符new和delete控制动态内存分配，也可被重载。这是下一章非常重要的主题。

运算符->*是其行为像所有其他二元运算符的二元运算符。它是为模仿前一章介绍的内部数据类型的成员指针行为的情形而提供的。

为了使一个对象的表现是一个指针，就要设计使用灵巧 (smart)指针：->。如果想为类包装一个指针以使得这个指针安全，或是在一个普通的循环子（iterator）的用法中，则这样做特别有用。循环子是一个对象，这个对象可以作用于其他对象的包容器或集合上，每次选择它们中的一个，而不用提供对包容器实现的直接访问。（在类函数里经常发现包容器和循环子。）

灵巧指针必须是成员函数。它有一个附加的非典型的内容：它必须返回一个对象（或对象的引用），这个对象也有一个灵巧指针或指针，可用于选择这个灵巧指针所指向的内容。这儿提供了一个例子：

```
//: SMARTP.CPP -- Smart pointer example
#include <iostream.h>
#include <string.h>

class obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
};

// Static member definitions:
int obj::i = 47;
int obj::j = 11;

// Container:
class obj_container {
  enum { sz = 100 };
```

```
  obj* a[sz];
  int index;
public:
  obj_container() {
    index = 0;
    memset(a, 0, sz * sizeof(obj*));
  }
  void add(obj* OBJ) {
    if(index >= sz) return;
    a[index++] = OBJ;
  }
  friend class sp;
};

// Iterator:
class sp {
  obj_container* oc;
  int index;
public:
  sp(obj_container* OC) {
    index = 0;
    oc = OC;
  }
  // Return value indicates end of list:
  int operator++() { // Prefix
    if(index >= oc->sz) return 0;
    if(oc->a[++index] == 0) return 0;
    return 1;
  }
  int operator++(int) { // Postfix
    return operator++(); // Use prefix version
  }
  obj* operator->() const {
    if(oc->a[index]) return oc->a[index];
    static obj dummy;
    return &dummy;
  }
};

main() {
  const sz = 10;
  obj o[sz];
  obj_container OC;
  for(int i = 0; i < sz; i++)
    OC.add(&o[i]); // Fill it up
```

```
  sp SP(&OC); // Create an iterator
  do {
    SP->f(); // Smart pointer calls
    SP->g();
  } while(SP++);
}
```

类obj定义了程序中使用的对象。函数f( )和g( )用静态数据成员打印令人感兴趣的值。使用obj_container的函数add( )将指向这些对象的指针储存在类型 obj_container的包容器中。obj_container看起来像一个指针数组，但却发现没有办法得到这些指针。然而，类 sp声明为friend类，所以它允许进入这个包容器内。类sp看起来像一个聪明的指针 —— 可以使用运算符++向前移动它（也可以定义一个运算符 --），它不会超出包容器的范围，它可以返回它指向的内容（通过这个灵巧指针）。注意，循环子(iterator)是与包容器配套使用的 —— 不像指针，没有“ 通用目的 ”的循环子。包容器和循环子将在第 15章深入讨论。

在main( )中，一旦包容器OC装入obj对象，一个循环子SP就创建了。灵巧指针按下面的表达式调用：

```
SP->f( ) ; //Smart pointer calls
SP->g( ) ;
```

这里，尽管SP实际上并不含成员函数f( ) 和g( )，但结构指针机制通过obj*调用这些函数，obj* 是通过sp::operator->返回的。编译器进行所有检查以确信函数调用正确。

虽然，灵巧指针的执行机制比其他运算符复杂一些，但目的是一样的 ---为类的用户提供更为方便的语法。

11.3.5 不能重载的运算符

在可用的运算符集合里存在一些不能重载的运算符。这样限制的通常原因是出于对安全的考虑：如果这些运算符也可以被重载的话，将会造成危害或破坏安全机制，使得事情变得困难或混淆现有的习惯。

现在，成员选择运算符‘ . ’在类中对任何成员都有一定的意义。但如果允许它重载，就不能用普通的方法访问成员，只能用指针和指针运算符 ->访问。

成员指针逆向引用的运算符‘ .* ’因为与运算符‘ . ’同样的原因而不能重载。

没有求幂运算符。大多数通常的选择是从 Fotran语言引用运算符**，但这出现了难以分析的问题。C也没有求幂运算符， C++似乎也不需要，因为这可以通过函数调用来实现。求幂运算符增加了使用的方便，但没有增加新的语言功能，反而给编译器增加了复杂性。

不存在用户定义的运算符，即不能编写目前运算符集合中没有的运算符。不能这样做的部分原因是难以决定其优先级，另一部分原因是没有必要增加麻烦。

不能改变优先级规则。否则人们很难记住它们。

11.4 非成员运算符

在前面的一些例子里，运算符可能是成员运算符或非成员运算符，这似乎没有多大差异。这就会出现一个问题：“ 我应该选择哪一种？ ”总的来说，如果没有什么差异，它们应该是成员运算符。这样做强调了运算符和类的联合。当左侧操作数是当前类的对象时，运算符会工作得很好。

但也不完全是这种情况——有时我们左侧运算符是别的类对象。这种情况通常出现在为iostreams重载运算符<< 和 >>时候。

```
//: IOSOP.CPP -- Iostream operator overloading
// Example of non-member overloaded operators
#include <iostream.h>
#include <strstrea.h>
#include <string.h>
#include "..\allege.h"

class intarray {
  enum { sz = 5 };
  int i[sz];
public:
  intarray() {
    memset(i, 0, sz* sizeof(*i));
  }
  int& operator[](int x) {
    allege(x >= 0 && x < sz,
           "operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os,
               const intarray& ia);
  friend istream&
    operator>>(istream& is, intarray& ia);
};

ostream& operator<<(ostream& os,
                    const intarray& ia){
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, intarray& ia){
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}

main() {
```

```
  istrstream input("47 34 56 92 103");
  intarray I;
  input >> I;
  I[4] = -1; // Use overloaded operator[]
  cout << I;
}
```

这个类也包含重载运算符[]，这个运算符在数组里返回了一个合法值的引用。一个引用被返回，所以下面的表达式：

```
I[4] = -1 ;
```

看起来不仅比使用指针更规范些，而且它也达到了预期的效果。

被重载的移位运算符通过引用方式传递和返回，所以运算将影响外部对象。在函数定义中，表达式像

```
os << ia.i[j] ;
```

会使现有的重载运算符函数被调用（即那些定义在 IOSTREAM.H中的）。在这个情况下，被调用的函数是ostream& operator<<(ostream&,int)，这是因为ia.i[j]是一个int值。

一旦所有的动作在istream或ostream上完成，它将被返回，因此它可被用于更复杂的表达式。

这个例子使用的是插入符和提取符的标准形式。如果我们想为自己的类创建一个集合，可以拷贝这个函数署名和返回类型，并遵从它的体的形式。

基本方针

Murry [1] 为在成员和非成员之间的选择提出了如下的方针：

运算符	建议使用
所有的一元运算符	成员
= ( ) [] ->	必须是成员
+= -= /= *= ^= &= |= %= >>= <<=	成员
所有其他二元运算符	非成员

11.5 重载赋值符

赋值符在C++中常常产生混淆。这是毫无疑问的，因为‘ = ’在编程中是最基本的运算符，是在机器层上拷贝寄存器。另外，当使用‘ = ’时也能引起拷贝构造函数（上一章内容）调用：

```
foo B ;
foo A = B ;
A = B ;
```

第2行定义了对象A。一个新对象先前不存在，现在正被创建。因为我们现在知道了 C++编译器关于对象初始化是如何保护的，所以知道在对象被定义的地方构造函数总是必须被调用。但是哪个构造函数呢？A是从现有的foo对象创建的，所以只有一个选择：拷贝构造函数。所以虽然这里只包括一个‘ = ’，但拷贝构造函数仍被调用。

[1] Rob Murray, C++ Strategies & Tactics, Addision-Wesley, 1993, page 47.

第3行情况不同了。在‘ = ’左侧有一个以前初始化了的对象。很清楚，不用为一个已经存在的对象调用构造函数。在这种情况下，为 A调用foo::operator=，把foo::operator=右侧的任何东西作为参数。（我们可以有多种取不同右侧参数的operator=函数）

对于拷贝构造函数没有这个限制。我们在任何时候使用一个‘ = ’代替普通形式的构造函数调用来初始化一个对象时，无论=右侧是什么，编译器都会为我们寻找一个构造函数：

```
//: FEEFI.CPP -- Copying vs. initialization

class fi {
public:
  fi() {}
};

class fee {
public:
  fee(int) {}
  fee(const fi&) {}
};

main() {
  fee f = 1; // fee(int)
  fi FI;
  fee fum = FI; // fee(fi)
}
```

当处理‘ = ’时，记住这个差别是非常重要的：如果对象还没有被创建，初始化是需要的，否则使用赋值运算符‘ = ’。

对于初始化，使用‘ = ’可以避免写代码。但这要用显式的构造函数形式。于是最后一行应写成：

```
fee fum(FI) ;
```

这个方法可以避免使读者混淆。

运算符‘ = ’的行为

在BINARY.CPP中，我们看到运算符‘ = ’仅是成员函数，它密切地与‘ = ’左侧的对象相联系。如果允许我们全局性地定义运算符‘ = ’，那么我们就会试图重新定义内置的‘ = ’：

```
int operator=(int,foo) ; // global = not allowed!
```

这是绝对不允许的，编译器通过强制运算符‘ = ’为成员函数而避开这个问题。

当创建一个运算符‘ = ’时，必须从右侧对象中拷贝所有需要的信息完成为类的“ 赋值 ”，对于单个对象，这是显然的：

```
//: SIMPCOPY.CPP -- Simple operator=()
#include <iostream.h>

class value {
  int a, b;
```

```
  float c;
public:
  value(int A = 0, int B = 0, float C = 0.0) {
    a = A;
    b = B;
    c = C;
  }
  value& operator=(const value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
  friend ostream&
    operator<<(ostream& os, const value& rv) {
      return os << "a = " << rv.a << ", b = "
        << rv.b << ", c = " << rv.c;
    }
};

main() {
  value A, B(1, 2, 3.3);
  cout << "A: " << A << endl;
  cout << "B: " << B << endl;
  A = B;
  cout << "A after assignment: " << A << endl;
}
```

这里，‘ = ’左侧的对象拷贝了右侧对象中的所有内容，然后返回它的引用，所以我们还可以创建更加复杂的表达式。

这个例子犯了一个普通的错误。当准备给两个相同类型的对象赋值时，应该首先检查一下自赋值(self-assignment)：这个对象是否对自身赋值了？在一些情况下，例如本例，无论如何执行这些赋值运算都是无害的，但如果对类的实现作了修改，那么将会出现差异。如果我们习惯于不做检查，就可能忘记并产生难以发现的错误。

1. 类中指针

如果对象不是如此简单时将会发生什么事情？例如，如果对象里包含指向别的对象的指针将如何？简单地拷贝一个指针意味着以指向相同的存储单元的对象而结束。这种情况，就需要自己做注释记住这点。

这里有两个解决问题的方法。当我们做一个赋值运算或一个拷贝构造函数时，最简单的技术是拷贝这个指针所涉及的一切，这是非常简单的。

```
//: COPYMEM.CPP -- Duplicate during assignment
#include <stdlib.h>
#include <string.h>
#include "..\allege.h"
```

```
class withPointer {
  char* p;
  enum { blocksz = 100 };
public:
  withPointer() {
    p = (char*)malloc(blocksz);
    allegemem(p);
    memset(p, 1, blocksz);
  }
  withPointer(const withPointer& wp) {
    p = (char*)malloc(blocksz);
    allegemem(p);
    memcpy(p, wp.p, blocksz);
  }
  withPointer&
  operator=(const withPointer& wp) {
    // Check for self-assignment:
    if(&wp != this)
      memcpy(p, wp.p, blocksz);
    return *this;
  }
  ~withPointer() {
    free(p);
  }
};

main() {}
```

这里展示了当我们的类包含了指针时，总是需要定义的 4个函数：所有必需的普通构造函数、拷贝构造函数、运算符‘ =’(无论定义它还是不允许它）和析构函数。对运算符‘ =’当然要检查自赋值，虽然这儿不需要，但我们应养成这种习惯。这实际上减少了改变代码而忘记检查自赋值的可能性。

这里，构造函数分配存储单元并对它初始化，运算符‘ =’拷贝它，析构函数释放存储单元。然而，如果要处理许多存储单元并对其初始化时，我们也许想避免这种拷贝。解决这个问题的通常方法被称为引用记数 (reference counting)。可以使一块存储单元具有智能，它知道有多少对象指向它。拷贝构造函数或赋值运算意味着把另外的指针指向现在的存储单元并增加引用记数。消除意味着减小引用记数，如果引用记数为 0意味着销毁这个对象。

但如果向这块存储单元写入将会如何呢？因为不止一个对象使用这块存储单元，所以当我们修改自己的存储单元时，也等于也修改了他人的存储单元。为了解决这个问题，经常使用另外一个称为写拷贝(copy-on-write)的技术。在向这块存储单元写之前，应该确信没有其他人使用它。如果引用记数大于 1，在写之前必须拷贝这块存储单元，这样就不会影响他人了。这儿提供了一个简单的引用记数和关于写拷贝的例子：

```
//: REFCOUNT.CPP -- Reference count, copy-on-write
#include <string.h>
```

```
#include <assert.h>

class counted {
  class memblock {
    enum { size = 100 };
    char c[size];
    int refcount;
  public:
    memblock() {
      memset(c, 1, size);
      refcount = 1;
    }
    memblock(const memblock& rv) {
      memcpy(c, rv.c, size);
      refcount = 1;
    }
    void attach() { ++refcount; }
    void detach() {
      assert(refcount != 0);
      // Destroy object if no one is using it:
      if(--refcount == 0) delete this;
    }
    int count() const { return refcount; }
    void set(char x) { memset(c, x, size); }
    // Conditionally copy this memblock.
    // Call before modifying the block; assign
    // resulting pointer to your block;
    memblock* unalias() {
      // Don't duplicate if not aliased:
      if(refcount == 1) return this;
      --refcount;
      // Use copy-constructor to duplicate:
      return new memblock(*this);
    }
  } * block;
public:
  counted() {
    block = new memblock; // Sneak preview
  }
  counted(const counted& rv) {
    block = rv.block; // Pointer assignment
    block->attach();
  }
  void unalias() { block = block->unalias(); }
  counted& operator=(const counted& rv) {
```

```
    // Check for self-assignment:
    if(&rv == this) return *this;
    // Clean up what you're using first:
    block->detach();
    block = rv.block; // Like copy-constructor
    block->attach();
    return *this;
  }
  // Decrement refcount, conditionally destroy
  ~counted() { block->detach(); }
  // Copy-on-write:
  void write(char value) {
    // Do this before any write operation:
    unalias();
    // It's safe to write now.
    block->set(value);
  }
};

main() {
  counted A, B;
  counted C(A);
  B = A;
  C = C;
  C.write('x');
}
```

嵌套类memblock是被指向的一块存储单元。（注意指针block定义在嵌套类的最后）它包含了一个引用记数及控制和读引用记数的函数。同时这里存在一个拷贝构造函数，所以我们可以从现有的类创建一个新的memblock。

函数attach( )增加一个memblock引用记数用以指示有另一个对象使用它。函数 detach( )减少引用记数。如果引用记数为0，则说明没有对象使用它，所以通过表达式delete this 成员函数销毁它自己的对象。

可以用函数 set( ) 修改存储单元。但在做修改之前，应该确信不是在别的对象使用的memblock上进行。可以通过调用connted::unalias( )，connted::unalias( )调用memblock::unalias( )来做到这点。如果引用记数为 1（意味着没有别的对象指向这块存储单元），后面这个函数将返回block指针，但如果引用记数大于1就要复制这个存储单元。

这个例子已涉及到下一章的内容。C++运算符new和delete代替C语言的malloc( )和free( )来创建和销毁对象。对于这个例子，除了new在分配了存储单元后调用构造函数，delete在释放存储单元之前调用析构函数之外，可以认为new 和delete与malloc( )和free( )一样。

拷贝构造函数给源对象block赋值block，而不是创建它自己的存储单元。然后因为现在增加了使用这个存储单元的对象，所以通过调用 memblock::attach( )增加引用记数。

运算符‘ = ’处理‘ = ’左侧已创建的对象，所以它必须通过为 memblock调用detach( )而首先整理这个存储单元。如果没有其他对象使用它，这个老的 memblock将被销毁。然后运算符

‘=’重复拷贝构造函数的行为。注意它首先检查是否给它本身赋予相同的对象。

析构函数调用detach( )有条件地销毁memblock。

为了实现写拷贝，必须控制所有写存储单元的动作。这意味着不能向外部传递原有指针。我们会说：“告诉我您想做什么，我将为您做！”例如成员函数 write( )允许对这个存储单元修改数值。但它首先必须使用unalias( )防止修改一个已别名化了的存储单元（超过一个对象使用的存储单元）。

在main( )中测试了几个必须正确实现引用记数的函数：构造函数、拷贝构造函数、运算符‘=’和析构函数。在main( )中也通过为对象C调用write( )测试了写拷贝，对象C是已别名化了的A存储单元。

2. 跟踪输出

为了验证这个方案是正确的，最好的方法是对类增加信息和功能以便产生可被分析的跟踪输出。这儿的REFCOUNT.CPP增加了跟踪信息。

```
//: RCTRACE.CPP -- REFCOUNT.CPP w/ trace info
#include <string.h>
#include <fstream.h>
#include <assert.h>
ofstream out("rctrace.out");

class counted {
  class memblock {
    enum { size = 100 };
    char c[size];
    int refcount;
    static int blockcount;
    int blocknum;
  public:
    memblock() {
      memset(c, 1, size);
      refcount = 1;
      blocknum = blockcount++;
    }
    memblock(const memblock& rv) {
      memcpy(c, rv.c, size);
      refcount = 1;
      blocknum = blockcount++;
      print("copied block");
      out << endl;
      rv.print("from block");
    }
    ~memblock() {
    out << "\tdestroying block "
        << blocknum << endl;
  }
  void print(const char* msg = "") const {
```

```
      if(*msg) out << msg << ", ";
      out << "blocknum:" << blocknum;
      out << ", refcount:" << refcount;
    }
    void attach() { ++refcount; }
    void detach() {
      assert(refcount != 0);
      // Destroy object if no one is using it:
      if(--refcount == 0) delete this;
    }
    int count() const { return refcount; }
    void set(char x) { memset(c, x, size); }
    // Conditionally copy this memblock.
    // Call before modifying the block; assign
    // resulting pointer to your block;
    memblock* unalias() {
      // Don't duplicate if not aliased:
      if(refcount == 1) return this;
      --refcount;
      // Use copy-constructor to duplicate:
      return new memblock(*this);
    }
  } * block;
  enum { sz = 30 };
  char id[sz];
public:
  counted(const char* ID = "tmp") {
    block = new memblock; // Sneak preview
    strncpy(id, ID, sz);
  }
  counted(const counted& rv) {
    block = rv.block; // Pointer assignment
    block->attach();
    strncpy(id, rv.id, sz);
    strncat(id, " copy", sz - strlen(id));
  }
  void unalias() { block = block->unalias(); }
  void addname(const char* nm) {
    strncat(id, nm, sz - strlen(id));
  }
  counted& operator=(const counted& rv) {
    print("inside operator=\n\t");
    if(&rv == this) {
      out << "self-assignment" << endl;
```

```
    return *this;
  }
  // Clean up what you're using first:
  block->detach();
  block = rv.block; // Like copy-constructor
  block->attach();
  return *this;
}
// Decrement refcount, conditionally destroy
~counted() {
  out << "preparing to destroy: " << id
      << endl << "\tdecrementing refcount ";
  block->print();
  out << endl;
  block->detach();
}
// Copy-on-write:
void write(char value) {
  unalias();
  block->set(value);
}
void print(const char* msg = "") {
  if(*msg) out << msg << " ";
  out << "object " << id << ": ";
  block->print();
  out << endl;
  }
};

int counted::memblock::blockcount = 0;

main() {
  counted A("A"), B("B");
  counted C(A);
  C.addname(" (C) ");
  A.print();
  B.print();
  C.print();
  B = A;
  A.print("after assignment\n\t");
  B.print();
  out << "Assigning C = C" << endl;
  C = C;
  C.print("calling C.write('x')\n\t");
  C.write('x');
  out << endl << "exiting main()" << endl;
}
```

现在memblock含有一个static数据成员blockcount来记录创建的存储单元号码，为了区分这些存储单元它还为每个存储单元创建了唯一号码（存放在 blocknum中）。在析构函数中声明哪一个存储单元被销毁，print( )函数显示块号和引用记数。

类counted含有一个缓冲器 id用来记录对象信息。 counted构造函数创建了一个新的memblock对象并把结果赋给了block（这个结果是一个堆上指向memblock的指针）。从参数拷贝来的标识符加了一个单词“ copy ”用以显示它是从哪里拷贝来的。函数addname( )也让我们在id（这是实际的标识符，所以我们可以看到它是什么以及从哪里拷贝来的）中加入有关对象的附加信息。

这里是输出结果：

```
object A: blocknum:0, refcount:2
object B: blocknum:1, refcount:1
object A copy (C) : blocknum:0, refcount:2
inside operator=
        object B: blocknum:1, refcount:1
      destroying block 1
after assignment
        object A: blocknum:0, refcount:3
object B: blocknum:0, refcount:3
Assigning C = C
inside operator=
        object A copy (C) : blocknum:0, refcount:3
self-assignment
calling C.write('x')
        object A copy (C) : blocknum:0, refcount:3
copied block, blocknum:2, refcount:1
from block, blocknum:0, refcount:2
exiting main()
preparing to destroy: A copy (C)
      decrementing refcount blocknum:2, refcount:1
      destroying block 2
preparing to destroy: B
      decrementing refcount blocknum:0, refcount:2
preparing to destroy: A
      decrementing refcount blocknum:0, refcount:1
      destroying block 0
```

通过研究输出结果、跟踪源代码和对程序测试，我们会加深对这些技术的理解。

3. 自动创建运算符‘ = ’

因为将一个对象赋给另一个相同类型的对象是大多数人可能做的事情，所以如果没有创建type::operator=(type)，编译器会自动创建一个。这个运算符行为模仿自动创建的拷贝构造函数的行为：如果类包含对象（或是从别的类继承的），对于这些对象，运算符‘ = ’被递归调用。这被称为成员赋值(memberwise assignment)。见如下例子：

```
//: AUTOEQ.CPP -- Automatic operator=()
#include <iostream.h>
```

```
class bar {
public:
  bar& operator=(const bar&) {
    cout << "inside bar::operator=()" << endl;
    return *this;
  }
};

class foo {
  bar B;
};

main() {
  foo a, b;
  a = b; // Prints: "inside bar::operator=()"
}
```

为foo自动生成的运算符‘ = ’调用bar::operator=。

一般我们不会想让编译器做这些。对于复杂的类（尤其是它们包含指针的情况），我们应该显式地创建一个运算符‘ = ’。如果真的不想让人执行赋值运算，可以把运算符‘ = ’声明为private函数。（除非在类内使用它，否则不必定义它。）

11.6 自动类型转换

在C和C++中，如果编译器看到一个表达式或函数调用使用了一个不合适的类型，它经常会执行一个自动类型转换。在C++中，可以通过定义自动类型转换函数来为用户定义类型达到相同效果。这些函数有两种类型：特殊类型的构造函数和重载的运算符。

11.6.1 构造函数转换

如果我们定义一个构造函数，这个构造函数能把另一类型对象（或引用）作为它的单个参数，那么这个构造函数允许编译器执行自动类型转换。如下例：

```
//: AUTOCNST.CPP -- Type conversion constructor

class one {
public:
  one() {}
};

class two {
public:
  two(const one&) {}
};

void f(two) {}
```

```
main() {
  one One;
  f(One); // Wants a two, has a one
}
```

当编译器看到f( )以为对象one参数调用时，编译器检查f( )的声明并注意到它需要一个two对象作为参数。然后，编译器检查是否有从对象 one 到two的方法。它发现了构造函数two::two(one)，two::two(one)被悄悄地调用，结果对象two被传递给f( )。

在这个情况里，自动类型转换避免了定义两个 f( )重载版本的麻烦。然而，代价是隐藏了构造函数对two的调用，如果我们关心f( )的调用效率的话，那就不要使用这种方法。

• 阻止构造函数转换

有时通过构造函数自动转换类型可能出现问题。为了避开这个麻烦，可以通过在前面加关键字explicit [1]（只能用于构造函数）来修改构造函数。上例类two的构造函数作了修改，如下：

```
class one {
public:
  one() {}
};

class two {
public:
  explicit two(const one&) {}
};

void f(two) {}

main() {
  one One;
//!  f(One); // no auto conversion allowed
  f(two(One)); // OK -- user performs conversion
}
```

通过使类two的构造函数显式化，编译器被告知不能使用那个构造函数（那个类中其他非显式化的构造函数仍可以执行自动类型转换）执行任何自动转换。如果用户想进行转换必须写出代码。上面代码f(two(One))创建一个从类型One到two的临时对象，就像编译器在前面版本中做的那样。

11.6.2 运算符转换

第二种自动类型转换的方法是通过运算符重载。我们可以创建一个成员函数，这个函数通过在关键字operator后跟随想要转换到的类型的方法，将当前类型转换为希望的类型。这种形式的运算符重载是独特的，因为没有指定一个返回类型——返回类型就是我们正在重载的运算符的名字。这儿有一个例子：

[1] 写作本书时，explicit是语言中新的关键字。我们的编译器可能还不支持它。

```
//: OPCONV.CPP -- Op overloading conversion

class three {
  int i;
public:
  three(int I = 0, int = 0) : i(I) {}
};

class four {
  int x;
public:
  four(int X) : x(X) {}
  operator three() const { return three(x); }
};

void g(three) {}
main() {
  four Four(1);
  g(Four);
  g(1);  // Calls three(1,0)
}
```

用构造函数技术，目的类执行转换。然而使用运算符技术，是源类执行转换。构造函数技术的价值是在创建一个新类时为现有系统增加了新的转换途径。然而，创建一个单一参数的构造函数总是定义一个自动类型转换（即使它有不止一个参数也是一样，因为其余的参数将被缺省处理），这可能并不是我们所想要的。另外，使用构造函数技术没有办法实现从用户定义类型向内置类型转换，这只有运算符重载可能做到。

• 反身性

使用全局重载运算符而不用成员运算符的最便利的原因之一是在全局版本中的自动类型转换可以针对左右任一操作数，而成员版本必须保证左侧操作数已处于正确的形式。如果想两个操作数都被转换，全局版本可以节省很多代码。这儿有一个小例子。

```
//: REFLEX.CPP -- Reflexivity in overloading

class number {
  int i;
public:
  number(int I = 0) { i = I; }
  const number
  operator+(const number& n) const {
    return number(i + n.i);
  }
  friend const number
    operator-(const number&, const number&);
};
```

```
const number
  operator-(const number& n1,
            const number& n2) {
    return number(n1.i - n2.i);
}

main() {
  number a(47), b(11);
  a + b; // OK
  a + 1; // 2nd arg converted to number
//! 1 + a; // Wrong! 1st arg not of type number
  a - b; // OK
  a - 1; // 2nd arg converted to number
  1 - a; // 1st arg converted to number
}
```

类number有一个成员运算符+号和一个友元(friend)运算符 - 号。因为有一个使用单一int参数的构造函数，int自动转换为number，但这要在正确的条件下。在 main( )里，可以看到增加一个number到另一个number进行得很好，这是因为它重载的运算符非常匹配。当编译器看到一个number后跟一个+号和一个int时，它也能和成员函数number::operator+相匹配并且构造函数把int参数转换为number。但当编译器看到一个int、一个+号和一个number时，它就不知道如何去做，因为它所拥有的是number::operator+，需要左侧的操作数是number对象。因此，编译器发出一个出错信息。

对于友元运算符-号，情况就不同了。编译器需要填满两个参数，它不限定 number作为左侧参数。因此，如果看到表达式 1-a，编译器就使用构造函数把第一个参数转换为 number。有时我们也许想通过把它们设成成员函数来限定运算符的使用。例如当用一个矢量与矩阵相乘，矢量必须在右侧。但如果想让运算符转换任一个参数，就要使运算符为友元函数。

幸运的是编译器不会把表达式 1-1的两个参数转换为number对象，然后调用运算符 - 号。那将意味着现有的C代码可能突然执行不同的工作了。编译器首先匹配“最简单的”可能性，对于表达式1-1将优先使用内部运算符。

11.6.3 一个理想的例子：strings

这是一个自动类型转换对于 string类非常有帮助的例子。如果不用自动类型转换就想从标准的C库函数中使用所有的字符串函数，那么就得为每一个函数写一个相应的成员函数，就像下面的例子：

```
//: STRINGS1.CPP -- No auto type conversion
#include <string.h>
#include <stdlib.h>
#include "..\allege.h"

class string {
  char* s;
public:
  string(const char* S = "") {
```

```
    s = (char*)malloc(strlen(S) + 1);
    allegemem(s);
    strcpy(s, S);
  }
  ~string() { free(s); }
  int Strcmp(const string& S) const {
    return ::strcmp(s, S.s);
  }
  // ... etc., for every function in string.h
  };

  main() {
    string s1("hello"), s2("there");
    s1.Strcmp(s2);
  }
```

这里只写了strcmp( )函数，但必须为STRING.H可能需要的每一个函数写一个相应的函数。幸运的是，可以提供一个允许访问STRING.H中所有函数的自动类型转换：

```
//: STRINGS2.CPP -- With auto type conversion
#include <string.h>
#include <stdlib.h>
#include "..\allege.h"

class string {
  char* s;
public:
  string(const char* S = "") {
    s = (char*)malloc(strlen(S) + 1);
    allegemem(s);
    strcpy(s, S);
  }
  ~string() { free(s); }
  operator const char*() const { return s; }
};

main() {
  string s1("hello"), s2("there");
  strcmp(s1, s2); // Standard C function
  strspn(s1, s2); // Any string function!
}
```

因为编译器知道如何从string转换到char*，所以现在任何一个接受char*参数的函数也可以接受string参数。

11.6.4 自动类型转换的缺陷

因为编译器必须选择如何执行类型转换，所以如果没有正确地设计出转换，编译器会产生

麻烦。类X可以用operator Y( ) 将它本身转换到类Y，这是一个简单且明显的情况。如果类Y有一个单个参数为X的构造函数，也表示同样的类型转换。现在编译器有两个从 X到Y的转换方法，所以发生转换时，编译器会产生一个不明确指示的出错信息：

```
//: AMBIG.CPP -- Ambiguity in type conversion

class Y; // Class declaration

class X {
public:
  operator Y() const; // Convert X to Y
};

class Y {
public:
  Y(X); // Convert X to Y
};

void f(Y);

main() {
  X x;
//! f(x); // Error: ambiguous conversion
}
```

这个问题解决方案不是仅提供一个从一个类型到另一个类型的自动转换路径。

提供自动转换到不止一种类型时，会引发更困难的问题。有时，这个问题被称为扇出(fan-out)：

```
//: FANOUT.CPP -- Type conversion fanout

class A {};
class B {};

class C {
public:
  operator A() const;
  operator B() const;
};

// Overloaded h():
void h(A);
void h(B);

main() {
  C c;
//! h(c); // Error: C -> A or C -> B ???
}
```

类C有向A和B的自动转换。这样存在一个隐藏的缺陷：使用创建的两种版本的重载运算符h( )时问题就出现了。（只有一个版本时，main( )里的代码会正常运行。）

通常，对于自动类型的解决方案是只提供一个从一个类型向另一个类型转换的自动转换版本。当然我们也可以有多个向其他类型的转换，但它们不应该是自动转换，而应该创建显式的调用函数，例如用名字make_A( )和make_B( )表示这些函数。

• 隐藏的行为

自动类型转换会引入比所希望的更多的潜在行为。下面看 11.5节FEEFI.CPP的修改后的例子：

```
//: FEEFI2.CPP -- Copying vs. initialization

class fi {};

class fee {
public:
  fee(int) {}
  fee(const fi&) {}
};

class fo {
  int i;
public:
  fo(int x = 0) { i = x; }
  operator fee() const { return fee(i); }
};

main() {
  fo FO;
  fee fiddle = FO;
}
```

这里没有从fo对象创建fee fiddle的构造函数。然而，fo有一个到fee的自动类型转换。这里也没有从fee对象创建fee的拷贝构造函数，但这是一种能由编译器帮助我们创建的特殊函数之一。（缺省的构造函数、拷贝构造函数、运算符‘ = ’和析构函数可被自动创建）对于下面的声明，自动类型转换运算符被调用并创建一个拷贝函数：

```
fee fiddle = FO ;
```

自动类型转换应该小心使用。它在减少代码方面是非常出色的，但不值得无缘无故地使用。

11.7 小结

运算符重载存在的原因是为了使编程容易。运算符重载没有那么神秘，它只不过是拥有有趣名字的函数。当它以正确的形式出现时，编译器调用这个函数。但如果运算符重载对于类的设计者或类的使用者不能提供特别显著的益处，则最好不要使用，因为增加运算符重载会使问题混淆。

11.8 练习

1. 写一个有重载运算符++的类。试着用前缀和后缀两种形式调用此运算符，看看编译器会给我们什么警告。

2. 写一个只含有单个private char成员的类。重载iostream运算符<<和>>（像在IOSOP.CPP中的一样）并测试它们，可以用fstreams、strstreams和stdiostreams(cin和cout)测试它们。

3. 写一个包含重载的运算符+、-、*、/和赋值符的number类。出于效率考虑，为这些函数合理地选择返回值以便以链式写表达式。写一个自动类型转换运算符 int( )。

4. 合并在UNARY.CPP和BINARY.CPP中的类。

5. 对FANOUT.CPP作如下修改：创建一个显式函数，用它代替自动转换运算符来完成类型转换。