中国科学技术大学软件学院
SCHOOL OF SOFTWARE ENGINEERING OF USTC

第六章 操作符重载

ffh

费飞辉

目录

第六章 操作符重载

- 运算符重载就是赋予已有的运算符多重含义。

  例如：

  a=3+4；

  a=”abc”+”def”;

  同一个运算符“＋”，由于所操作的数据类型不同而具有不同的意义，这就是运算符重载，而且是系统预先定义的运算符重载。

- 通过重新定义运算符，使它能够用于特定类的对象执行特定的功能。增强了语言的扩充能力
- C++本身已对操作符进行了重载

2/3	2.0/3.0
2*3	2.0*3.0
*p	int *p
2&3	&p

6.1 基本操作符重载

- 可以被重载的操作符：

  new　new[ ]　delete　delete[ ]

  +　-　*　/　%　^　&

  |　~　!　=　<　>　+=

  <<　>>　<<=　>>=　++　!=　<=

  >=　&&　||　++　--　,　->*

  ->　( )　[ ]

  编程者只能从上述运算符中选择进行重载的运算符，不能再定义新的运算符。

6.1 基本操作符重载

- 不能被重载的操作符：

运算符	运算符的含义	不允许重载的原因
?:	条件运算符	在C++中没有定义一个三目运算符的语法
.	成员选择符	为保证成员操作符对成员访问的安全性，故不允许重载
*	成员对象指针操作符	同上
::	作用域运算符	因该操作符左边的操作数是一个类型名，而不是一个表达式

6.1 基本操作符重载

- 除了赋值操作符（=）之外，基类中所有被重载的操作符都将被派生类继承。
- 重载语法：

  〈返回值类型〉 operator 〈运算符〉（〈形式参数表〉）

  返回类型可以为任何有效类型，但通常返回它们操作类的对象。

  〈运算符〉为要重载的运算符，〈形式参数表〉中的参数个数与重载的运算符操作数的个数有关。

6.1 基本操作符重载

- 重载为类的成员函数。先看一例。

在类C中重载操作符+,用来对2个该类的对象进行加操作，并将另一个该类的对象作为结果。

```
class C{
  public:
    C  operator+(const C&) const;
  };
```

定义形式：

```
C  C::operator+(const C&c)  const  {          }
```

C的成员函数只有一个参数，但是+操作符需要两个操作数。实际上，第一个操作数就是调用该函数的对象。

6.1 基本操作符重载

调用语法：

C a,b,c;

a=b.operator+(c);

通常形式：

a=b+c;

在语句a = b + c中，b的成员函数operator+被调用，等同于：

a = b.operator + (c)

- 通常，对于二元操作符函数，重载为成员函数时，只能显式声明一个形参，这个形参表示右操作数，左操作数由引用参数提供。重载一个一元操作符,其函数不需要任何参数。

6.1 基本操作符重载

- 例6-1

  以下代码展示了如何重载二元操作符=和一元操作符！

```
class C
{
public:
    C& operator=  (const C&);
    C operator !  ();
    //...
};
```

6.1 基本操作符重载

- 例6-2：封装有序数对的类

```
class Opair{
public:
   Opair(float  f1=0.0, float  f2=0.0)
     {          p1=f1;    p2=f2;
     }
private:
   float p1,p2;
};
```

设当且仅当p1和p2分别相等，二个Opair对象相等。

为判断二个Opair对象相等，可对==重载。

6.1 基本操作符重载

- 对操作符==进行重载，当且仅当数据成员p1和p2分别相等时，两个OPair对象相等。

```
class Opair{
 public:
    bool operator= =(const Opair &) const;  };
```

- 实现：

```
bool Opair::operator= =(const Opair &s)const
  { return p1= =s.p1 && p2==s.p2; }
```

- 使用：

```
Opair s1,s2;
if(s1= =s2)  { //...              }
```

S1==S2 等同于 S1.operator==(S2)

6.1.1操作符的优先级和语法

- 对操作符的重载不改变操作符的优先级和结合性。
- 重载不改变操作符的语法。例如：

```
 int main()
{ Complex c1,c2,c3,ans;
  ans=c1+c2*c3;                    //优先级
}
```

表达式：ans = c1 + c2 * c3 等同于：ans = c1 + (c2 * c3)

重载操作符要注意的问题：

- 如果一个内建操作符是一元的，那么所有对它的重载仍是一元的。
- 如果一个内建操作符是二元的，那么所有对它的重载仍是二元的。

```
class C{
    public: C operator%(  );   //error
};              //操作数
```

目录

6.2 示例程序：复数类

- 要求实现一个表示复数的类，并且完成以下操作：
  - 重载+、-、*和/，以支持复数的算术运算。
  - 设计一个write函数，以输出一个复数至标准输出。
  - 设计默认构造函数，将实部和虚部设为零。
  - 设计拥有一个参数的构造函数，将实部设为该参数，将虚部设为零。
  - 设计拥有两个参数的构造函数，并将两个参数分别赋给实部和虚部。

(complex1)

6.2 示例程序：复数类

- 复数的算术运算定义：

  (a+bi)+(c+di)=(a+c)+(b+d)i

  (a+bi)-(c+di)=(a-c)+(b-d)i

  (a+bi) ×(c+di)=(ac-bd)+(ad+bc)i

  (a+bi) /(c+di)=(ac+bd)/(c^2+d^2)+[(bc-ad)/(c^2+d^2)]i

6.2 示例程序：复数类

```
class Complex {
public:
   Complex();                 // default
   Complex( double );         // real  given
   Complex( double, double ); // both  given
   void write() const;
   // operator methods
   Complex operator+( const Complex& ) const;
   Complex operator-( const Complex& ) const;
   Complex operator*( const Complex& ) const;
   Complex operator/( const Complex& ) const;
private:
   double real;   double imag;
};
```

6.2 示例程序：复数类

```
Complex::Complex() {   // default constructor
   real = imag = 0.0;
}
Complex::Complex( double re ) {  //real given but not imag
   real = re;   imag = 0.0;
}
Complex::Complex( double re, double im ) {
   real = re;   imag = im;
}
void Complex::write() const {
   cout << real << " + " << imag << 'i';
}
```

6.2 示例程序：复数类

```
// Complex + as binary operator
Complex Complex::operator+( const Complex& u ) const
{   Complex v( real + u.real,  imag + u.imag );
    return v;
}

// Complex - as binary operator
Complex Complex::operator-( const Complex& u ) const
{  Complex v( real - u.real,  imag - u.imag );
   return v;
}
```

6.2 示例程序：复数类

```
// Complex * as binary operator :(a+bi)×(c+di)=(ac-bd)+(ad+bc)i
Complex Complex::operator*( const Complex& u )const
{   Complex v( real * u.real - imag * u.imag,
                              imag * u.real + real * u.imag );
   return v;   }
// Complex / as binary operator:
// (a+bi) /(c+di)=(ac+bd)/(c²+d²)+[(bc-ad)/(c²+d² )]i
Complex Complex::operator/( const Complex& u ) const
{   double abs_sq = u.real * u.real + u.imag * u.imag;
   Complex v( ( real * u.real + imag * u.imag ) / abs_sq,
                    ( imag * u.real - real * u.imag ) / abs_sq );
   return v;  }
```

6.2 示例程序：复数类

```
int  main()
{   Complex  c1( 7.7, 5.5 );
    Complex  c2( 4.2, -8.3 );
    Complex  c3;

    c3 = c1 + c2;
    cout<< "c1 + c2 = ";          c3.write();
    cout<< '\n' ;

    c3 = c1 - c2;
    cout<< "c1 - c2 = ";          c3.write();
    cout<< '\n' ;
```

6.2 示例程序：复数类

```
    c3 = c1 * c2;
    cout<< "c1 * c2 = ";          c3.write();
    cout<< '\n' ;

    c3 = c1 / c2;
    cout<< "c1 / c2 = ";          c3.write();
    cout<< '\n' ;
    return  0;
}
```

输出结果：

```
c1 + c2 = 11.9 + -2.8i
c1 - c2 = 3.5 + 13.8i
c1 * c2 = 77.99 + -40.81i
c1 / c2 = -0.153819 + 1.00555i
```

课堂练习

- 下列运算符中，（C ）运算符在C++中不能重载。

  A =　　B ()　　C ::　　D delete

- 下列运算符中，（A ）运算符在C++中不能重载。

  A ?:　　B []　　C new　　D &&

- 下列关于运算符重载的描述中,正确的是（D ） 。

  A 运算符重载可以改变运算符的操作数的个数

  B 运算符重载可以改变优先级

  C 运算符重载可以改变结合性

  D 运算符重载不可以改变语法结构

（continue）

课堂练习

- 关于运算符重载，下列说法正确的是（B）。

  A. 所有的运算符都可以重载。

  B. 通过重载，可以使运算符应用于自定义的数据类型。

  C. 通过重载，可以创造原来没有的运算符。

  D. 通过重载，可以改变运算符的优先级。

目录

6.3 用顶层函数进行操作符重载

- 一个被重载的操作符，就是一个用户自定义的函数，只不过它可以享有操作符语法所带来的便利。
- 一般来说，被重载的操作符要么是类成员函数，要么就是顶层函数。
- 顶层操作符重载函数的调用方式与普通函数相同：

  a = operator+(b, c)； 或者：a = b + c
- 以顶层函数形式被重载的操作符必须在它的参数表中包含一个类对象(理由见下页) 。例如：

  C operator+(const C &c1,const C &c2) { }

  内存管理操作符new 、new[ ] 、delete 和delete[] 除外。

6.3 用顶层函数进行操作符重载

- 操作符以顶层函数实现时， 在他的参数表必须中包含一个类对象。 如果连一个类对象参数都没有，比如定义为

  C operator% (float f1, float f2) { //.... } //Error

  那么对如下表达式（其中x和y为float 型）：

  x % y

  编译系统就不能区别%是内建的，还是用户定义的。

  如果操作符%是类成员函数，或是有一个类对象参数的顶层函数，编译系统就能够根据特定的上下文决定调用哪一个%操作符。

6.3 用顶层函数进行操作符重载

- 通常，如果使用成员函数重载二元操作符，只需要一个参数；而用顶层函数进行二元操作符重载时，需要两个参数，这两个参数分别对应操作符的两个操作数。

  类似地，如果用成员函数重载一元操作符，不需要参数；如果用顶层函数，需要一个参数，该参数对应操作符惟一的操作数。

- 下标操作符[ ]、赋值操作符=、函数调用操作符( )和指针操作符->必须以类成员函数的形式进行重载

  因为操作符以类成员函数的形式重载时，可以保证其第一个操作数是类的对象。因而当x是一个类对象时，像9[x]和6 . 3 2 = x这样的表达式就不会被程序接受。

  ```
  int  operator[ ] (int i,Point &s) {    }   //error
  ```

6.3 用顶层函数进行操作符重载

- 顶层函数进行操作符重载相对于成员函数的优势：

例6-7

```
class Complex{
    Complex(float ) const;
    Complex operator+(const Complex &) const;
};
int main(){
        Complex a,b(4.3,-8.2);
        a=b+54.3 ；//ok，用转型构造
        a=54.3+b; //error，第一个数不是Complex对象
}
```

a=54.3+b; 相当于 a=54.3.operator+(b);

6.3 用顶层函数进行操作符重载

- 顶层函数进行操作符重载相对于成员函数的优势：

例6-8

```
Complex operator+(const Complex &t,const Complex &u)
   {        }
 int main(){
       Complex a,b(4.3,-8.2);
       a=b+54.3;  //ok
       a=54.3+b;  //ok
}
```

a=b+54.3; 相当于 a = operator+(b,54.3);

6.3 用顶层函数进行操作符重载

- 因此，与使用类成员函数重载二元操作符相比，只要定义了可用于转型的构造函数，使用顶层函数进行重载的一个优点就是：
  - 非对象操作数可以出现在操作符的左边。
  - 而使用类成员函数时，第一个操作数必须是类的对象。

6.3 用顶层函数进行操作符重载

- 如用顶层函数重载6.2例中类Complex的操作符+

```
Complex operator+(const Complex &t,const Complex &u)
{   return Complex(t.real+u.real,t.imag+u.imag);    }
```

重载会出编译错，因为real和imag是Complex类的私有成员，顶层函数operator+不能访问它们。

– 有3种解决方案：
  - 将real和imag 设计为Complex 类的的公有成员，但这种方法违背了类的信息隐藏原则；
  - 在Complex类中加入用于访问real和imag的公有成员函数；

    例6-9（见下页）
  - 第三种方法是将operator+声明为Complex类的friend。

friend 的含义是：如果函数f是类C的friend，f就可以存取C的Private和protect成员。

6.3 用顶层函数进行操作符重载

例6-9

```
class Complex
{  public:
    Complex() { real = imag = 0.0; }
    Complex( double ) { real = re; imag = 0.0; }
    Complex( double, double ) { real = re; imag = im; }
    void write() const { cout << real << " + " << imag << 'i'; }
    double get_real() const { return real;}
    double get_imag() const { return imag;}
  private:
    double real;
    double imag;
};
```

6.3 用顶层函数进行操作符重载

```
Complex operator + (const Complex& t, const Complex& u)
{   return Complex( t.get_real() + u.get_real(),
                    t.get_imag() + u.get_imag() );
}
Complex operator - (const Complex& t, const Complex& u)
{   return Complex( t.get_real() - u.get_real(),
                    t.get_imag() - u.get_imag() );
}
Complex operator * (const Complex& t, const Complex& u)
{     return  Complex( t.get_real() * u.get_real()
       - t.get_imag() * u.get_imag(),
      t.get_imag() * u.get_real() + t.get_real() * u.get_imag() );
}
```

6.3 用顶层函数进行操作符重载

```
Complex operator / (const Complex& t, const Complex& u)
{  double abs_sq = u.get_real() * u.get_real()
       + u.get_imag() * u.get_imag();

    return Complex( (t.get_real() * u.get_real()
      + t.get_imag() * u.get_imag()) / abs_sq,
      (t.get_imag() * u.get_real()
      - t.get_real() * u.get_imag()) / abs_sq );
}
```

目录

6.4 friend 函数

- 一个类的friend 函数可以访问该类的私有成员和保护成员，但该friend 函数不是类的成员函数。声明形式：

```
class C{
  //...
  friend int f();
  //...
};
```

该声明的作用是赋予f访问C的私有和保护成员的权力，因为f不是成员函数，该声明可以放在C中的private 、protected 或public的任意部分。

- 从严格意义上说，建议仅在重载操作符时使用friend函数。

```
class Complex
{ public:
    Complex();
    Complex( double );
    Complex( double, double );
    void write() const;
    friend Complex operator + (const Complex&, const Complex&);
    friend Complex operator - (const Complex&, const Complex&);
    friend Complex operator * (const Complex&, const Complex&);
    friend Complex operator / (const Complex&, const Complex&);
private:
    double real;
    double imag;
};
```

6.4 friend 函数

```
Complex operator + (const Complex& t, const Complex& u)
{
    return Complex( t.real + u.real, t.imag + u.imag );
}

Complex operator - (const Complex& t, const Complex& u)
{
    return Complex( t.real - u.real, t.imag - u.imag );
}
```

6.4 friend 函数

```
Complex operator * (const Complex& t, const Complex& u)
{
    return Complex( t.real * u.real - t.imag * u.imag,
                    t.imag * u.real + t.real * u.imag );
}

Complex operator / (const Complex& t, const Complex& u)
{
    double abs_sq = u.real * u.real + u.imag * u.imag;
    return Complex( (t.real * u.real+ t.imag * u.imag) / abs_sq,
                    (t.imag * u.real - t.real * u.imag) / abs_sq );
}
```

6.4 friend 函数

- friend函数不仅可以是顶层函数，也可以是另一个类中的成员函数。

```
class Date
{   public:
      Date(int, int, int)  { Month=m; day = d; year = y; }
      friend void Time::display(Date&);
    private:
      int month
      int day;
      int year;
};
```

6.4 friend 函数

```
class Time
{   public:
      Time(int, int, int) { hour = h; minute = m; sec=s; }
      void display(Date& d)  {
                  cout << d.month<<d.day<<d.year;
                  cout<<hour<<minute<<sec;  }
    private:
        int hour;     int minute;    int sec;
};
```

6.4 friend 函数

```
int main()
{   Time t1(10, 13, 56);
    Date d1(12, 25, 2004);
    t1.display(d1);
    return 0;
}
```

6.4 friend 函数

- 一个函数可以同时被多个类声明为“friend”,这样就可以引用多个类中的私有数据。如：

```
class A { private:  int a;     friend void Test(); };
class B { private:  int b;     friend void Test(); };
void Test()
{   A a;   B b;
    a.a=10;   b.b=20;
    int c = a.a + b.b;
    cout << "c = " << c << endl;}
int main()
{   Test();   return 0; }
```

6.4 friend 函数

- 不仅可以将一个函数声明为一个类的friend, 也可以将一个类声明为另一个类的friend。如：

```
class A  {  private:   int a;       friend class B;  };
class B  {  private:   int b;
          public:void f() { A a; a.a=10; b=20;  cout<<a.a+b<<endl; };
};
int main()
{    B b;    b.f();
       return 0;    }
```

再如，可在前例中将整个Time类声明为Date类的friend。这样time中的所有函数都可以访问Date中的所有成员。

6.4 friend 函数

- friend的关系是单向的，不是双向的，如果声明了A是B的friend,不等于B是A的friend.
- friend的关系不能传递。
- friend函数不是类的成员函数，却可以访问该类的私有和保护成员，从严格意义上来说，这不符合面向对象原则。
- 因此friend函数是存在争议的，且容易被误用。
- 建议仅在重载操作符时使用friend函数。

6.4 friend 函数 举例

- 例：用友元运算符重载函数实现复数的加、减运算。

```
class  Complex
{  private:
     float Real,Image;
   public:
     Complex(float r=0,float i=0)
     {Real=r;Image=i;}
     void Show(int i)
     {  cout<<"c"<<i<<"=" <<Real<<"+"<<Image<<"i"<<endl;}
     friend Complex operator + (Complex & ,Complex &);
     friend Complex operator - (Complex &, Complex &);
     friend Complex operator + (Complex &,float);
};
```

没有转型构造函数，不能调用上面的+函数，所以需要实现这个函数

```
Complex operator + (Complex &c1, Complex &c2)
  {  Complex t;
     t.Real=c1.Real+c2.Real;     t.Image=c1.Image+c2.Image;
     return t;  }
  Complex operator - (Complex &c1, Complex &c2)
  {  Complex t;
     t.Real=c1.Real - c2.Real;     t.Image=c1.Image - c2.Image;
     return t;  }
  Complex operator + (Complex &c,float s)
  {  Complex t;
     t.Real=c.Real+s;     t.Image=c.Image;
     return t;  }
```

```
void main(void)
  {  Complex c1(25,50),c2(100,200),c3,c4;
     c1.Show(1);
     c2.Show(2);
     c3=c1+c2;                    //c3=(25+50i)+(100+200i)=125+250i
     c3.Show(3);
     c4=c2－c1;                   //c4=(100+200i)－(25+50i)=75+150i
     c4.Show(4);
     c1=c1+200;                   //c1=25+50i+200=225+50i
     c1.Show(1);
}
```

2．运算符重载函数为友元函数

- 程序执行后输出：

  c1=25+50i

  c2=100+200i

  c3=125+250i

  c4=75+150i

  c1=225+50i

课堂练习

- 在成员函数中进行双目运算符重载时，其参数表中应带有（ B ）个参数。

  A. 0　　B. 1　　C. 2　　D. 3

- 下列运算符不能用友元函数重载的是（ B ）。

  A +　　B =　　C *　　D <<

- 在重载运算符函数时,下面（D）运算符必须重载为类成员函数形式。

  A +　　B -　　C ++　　D ->

- 如果表达式a+b中的"+"是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为（ A ）。

  A. a.operator+(b)　　B. b.operator+(a)

  C. operator+(a,b)　　D. operator(a+b)

课堂练习

- 友元运算符obj1 > obj2被C++编译器解释为（A ）。

  A operator>(obj1,obj2)　　B >(obj1,obj2)

  C obj2.operator>(obj1)　　D obj1.oprator>(obj2)

- 在表达式x+y*z中，+是作为成员函数重载的运算符，*是作为非成员函数重载的运算符。下列叙述中正确的是（C ）。

  A operator+有两个参数，operator*有两个参数

  B operator+有两个参数，operator*有一个参数

  C operator+有一个参数，operator*有两个参数

  D operator+有一个参数，operator*有一个参数

课堂练习

- 在重载一个运算符时，其参数表中没有任何参数，这表明该运算符是（ B ）。

  A. 作为友元函数重载的1元运算符

  B. 作为成员函数重载的1元运算符

  C. 作为友元函数重载的2元运算符

  D. 作为成员函数重载的2元运算符

- 写出下列程序运行结果

```
class T    {
      public:     T( ) { a=0; b=0; c=0; }
                  T( int i,int j,int k)   {a=i; b =j;c=k;}
                  void get( int &i, int &j, int &k)  {i=a; j=b; k=c;}
                  T operator*(T obj);
      private:     int a,b,c;       };
```

课堂练习

```
T T::operator *(T obj)    {
        T tempobj;                  tempobj.a=a * obj.a;
        tempobj.b=b * obj.b;        tempobj.c=c * obj.c;
        return tempobj;     }
int main() {
        T obj1( 1,2,3), obj2( 5,5,5), obj3;
        int a,b,c;
        obj3=obj1 * obj2;           obj3.get( a, b, c);
        cout<<"1:\ta="<<a<<'\t'<<"b="<<b<< '\t'<<"c="<<c<<endl;
                1：a=5  b=10  c=15
        (obj2 * obj3 ).get( a, b, c);
        cout<<"2:\ta="<<a<<'\t'<<"b="<<b<<'\t'<<"c="<<c<<endl;   }
                2：a=25  b=50  c=75
```

目录

6.5 输入输出操作符的重载

- 程序员可对操作符>>进行重载，以支持用户自定义数据类型。例：

  int i;

  cin>>i;

  可翻译为： cin.operator>>(i);

- >> 的第一个操作数是系统类的对象（如cin 是系统类istream的对象）；
- 在对用户自定义类型重载>>时，如果采用类成员函数的形式，就必须对系统类的源代码进行修改，而这显然是非常不明智的做法，因此, 只能将>>重载函数设计为顶层函数。

6.5 输入输出操作符的重载

- 例6-11(p.229)：重载>>来读取两个浮点数，并将其转换成复数，存入一个Complex对象中，以顶层函数的形式实现operator>>，如下所示：

返回是输入流，可以链式输入

输入流以引用方式

Complex对象以引用方式

```
istream& operator>>(istream & in,Complex  & c) {
    return  in>>c.real>>c.imag;
}
```

在进行操作时， operator>>需要存取Complex的私有成员。为方便存取，将operator>>声明为Complex的friend：

```
class Complex{
    friend istream &operator>>(istream &,Complex &);
};
```

6.5 输入输出操作符的重载

被重载的输入操作符可以这样使用：

```
Complex c_obj;
cin>>c_obj;  //等价于：   operator>>(cin,c_obj);
```

其中第二句当被求值时，等价于

```
cin>>c_obj.real>>c_obj.image;
```

这样，这条语句读入了代表复数的两个浮点数，并存入**Complex**对象**c_obj**中。

- 由于重载函数返回的是该输入流对象，可进行链式输入：

```
Complex c1_obj,c2_obj;
cin>>c1_obj>>c2_obj; //operator>>(operator>>(cin,c1_obj),c2_obj)
```

这就使得**>>**作用于用户定义类型时，其行为和作用与内建类型一样。

输入与输出操作符的重载

- 与>>类似，C++重载了左移操作符<<，使其能输出所有C++内建数据类型及某些系统类的信息。
- 例6-12 （complex2）

  可以用顶层函数的形式对<<进行重载，使其能针对用户自定义类型进行输出。

```
ostream& operator<<(ostream& out,Complex & c) {
    return  out<<c.real<<“+”<<c.imag<<‘i’;
}
```

输入与输出操作符的重载

- 例：设计一个2行3列的矩阵类，使其有矩阵输入、输出及二矩阵求和的功能。

```
#include <iostream>
using namespace std;
class Matrix
{   public:
      friend Matrix operator +(Matrix m1, Matrix m2);
      friend void operator >> (istream&, Matrix &);
      friend ostream&  operator <<(ostream&, Matrix &);
    private:
      int data[2][3] ; //定义矩阵使用的数组
};
```

输入与输出操作符的重载

```
void operator >> (istream& in, Matrix &m)
{     for(int i=0;i<2;i++)
                  for (int j=0;j<3;j++)  in>>m.data[i][j];         }
ostream&  operator << (ostream& out,Matrix &m)
{     for(int i=0;i<2;i++)
      {           for (int j=0;j<3;j++)         out<<m.data[i][j]<<"\t";
                  out<<endl;         }           return out;   }
Matrix operator +(Matrix m1,Matrix m2)
{     Matrix r;
      for(int i=0;i<2;i++)
                  for (int j=0;j<3;j++)
                              r.data[i][j]=m1.data[i][j]+m2.data[i][j];
      return r;   }
```

输入与输出操作符的重载

```
int main()
{   Matrix m1,m2,m3;
    cout<<“input m1:\n”; cin >> m1; //can not：  cin>>m1>>m2;
    cout<<"input m2:\n"; cin >> m2;
    m3 = m1 + m2;
    cout << "m1+m2=\n"<<m3;
   //can: cout<<m1<<m3;
    return 0;
}
```

运行结果：

```
input m1:
1 2 3
4 5 6
input m2:
7 8 9
1 2 3
m1+m2=
8       10      12
5       7       9
```

目录

6.6 赋值操作符的重载

- 运算符函数 operator=()必须被重载为类的非静态的成员函数，而且重载后的operator=()不能被继承。
- 拷贝构造函数和赋值操作符（=），都是用来拷贝一个类的对象给另一个同类型的对象。

  两者的区别在于：

  拷贝构造函数是用已存在的对象创建一个相同的对象。

  而赋值运算符则是把一个对象的数据成员的值赋值给另一个已存在的同类对象。

6.6 赋值操作符的重载

例：为点类Point重载=运算符。

```
class  Point
 {  public:
        Point(int xx=0 , int yy=0) { x=xx;y=yy; }
        Point(Point&  p) {x=p.x;  y=p.y; }       //拷贝构造函数
        ~Point(){ }
        point&  operator=(const Point&  p);     //赋值运算函数
        int getX() const  { return x; }
        int getY() const  { return y; }
    Private:
       int  x,y;
  };
```

6.6 赋值操作符的重载

```
Point&  point::operator=(const   Point&p)
{   x=p.x ;            y=p.y ;
    return    *this;   }
void main()
{    Point  a(5,8);            Point  b(a);
     cout<<"b:"<<b.getX()<<b.getY()<<endl;
     Point  c,d,e;
     c=d=e=a;
     cout<<"c:"<<c.getX()<<c.getY()<<endl;
     cout<<"d:"<<d.getX()<<d.getY()<<endl;
      cout<<"e:"<<e.getX()<<e.getY()<<endl;
}
```

6.6 赋值操作符的重载

程序运行结果为：

```
b:5  8
c:5  8
d:5  8
e:5  8
```

运算符函数operator=()的形参是“=”右边的操作数，而operator=()函数的返回值则将出现在“=”的左边。为了实现向“b=c=d=a”这样的表达式，运算符函数operator=()的返回类型应说明为一个指向所建对象的引用。例如，在本例中被说明为：

```
point&  operator=(const Point&  p);
```

6.6 赋值操作符的重载

- 如果类的声明中没有提供拷贝构造函数，也没有重载赋值操作符，编译器将会给这个类提供一个拷贝构造函数和一个赋值操作符。其功能是把一个对象中的每个数据成员逐个拷贝给另一个对象中的相应成员。

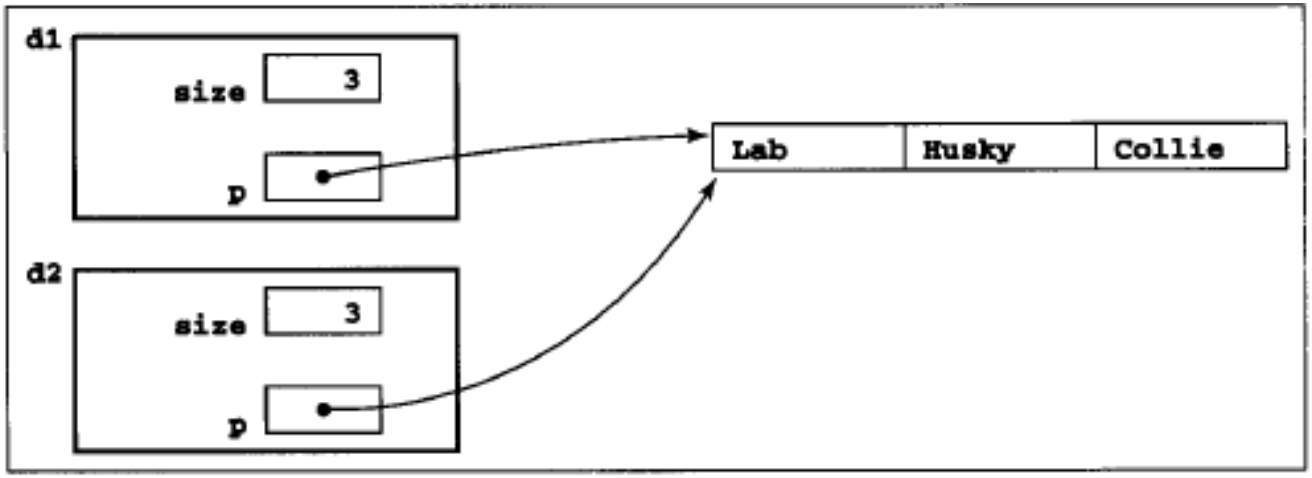
看书上实例 6-13

6.6 赋值操作符的重载

```
class C {
public:
    void set_x( int i ) { x = i; }
    void dump() const { cout << x << '\n'; }
private:
    int x;
};
int main() {
  C c1, c2;
  c1.set_x( -999 );
  c1.dump();  // -999 is printed
  c2 = c1;    // compiler-supplied assignment operator
  c2.dump();  // -999 is printed
  //...
}
```

6.6 赋值操作符的重载

- 一般情况下，缺省的拷贝构造函数和赋值函数都能很好的工作，并不需要程序员去专门定义。但如果一个类的数据成员中有指向动态分配空间的指针，类作者通常应定义拷贝构造函数，并重载赋值操作符。
- 下列（例6-14）如果使用编译器版本的赋值操作符，会出现和3.5节中由于使用编译器版本拷贝构造函数而带来的类似问题。



•指针d1.p和指针d2.p将指向同一块存储空间
•潜在危险：操作d1时可能会改变d2的内容，反之亦然。

例6-14 赋值操作符的重载

```
#include <iostream>    // 例6-14
#include <string>
using namespace std;
class CNameList
{ public:
    CNameList(void) : size(0), p(0) {};
    CNameList(const string [], int);
    CNameList(const CNameList&);
    CNameList& operator= (const CNameList&);
    void set(const string& s, int i) { p[i] = s; }
    void set(const char* s, int i) { p[i] = s; }
```

例6-14

```
    void dump() const;
private:
    int size;
    string* p;
    void copyIntoP(const CNameList&);
};
CNameList::CNameList(const string s[], int si)
{   p = new string[size = si];
    for (int i=0; i<size; i++)
      p[i] = s[i];
}
```

例6-14

```
CNameList::CNameList(const CNameList& d) : p(0)
{   copyIntoP(d);  }
CNameList& CNameList::operator= (const CNameList& d)
{   if (this != &d)
      copyIntoP(d);
    return *this;
}

void  CNameList::dump() const {
    for (int i=0; i<size; i++)
      cout<<p[i]<<'\t';   cout<<endl;
}
```

例6-14

```
void  CNameList::copyIntoP(const CNameList& d) {
    delete [] p;
    if (d.p != 0) {
      p = new string[size = d.size];
      for (int i=0; i<size; i++)
                p[i] = d.p[i];        }
    else        {
      p = 0;
      size = 0;            }
}
```

例6-14

```
int main()
{   string list[] = {"Lab","Zhihua","Song"};
    CNameList  d1(list, 3);
    d1.dump();
    CNameList  d2 ;
    d2 = d1;
    d2.dump();
    d2.set(“Great Dane”, 1);
    d2.dump();
    d1.dump();
    return  0;
}
```

例6-14 赋值操作符的重载

- 赋值操作符重载函数返回一个Namelist对象的引用。

  例如，赋值语句

  d2=d1;

  operator= 返回d2的引用。重载赋值操作符时，通常返回调用该操作符的对象的引用，这样赋值就能连续进行。例如，连续赋值操作

  d3=d2=d1;

  等价于

  d3.operator=(d2.operator=(d1));

6.6 赋值操作符的重载

例 一个在对象中使用动态内存的例子。 (t1.cpp)

```
#include <iostream>
using namespace std;
class  String
{  private:
     char *sPtr;
     int  length;
   public:
     String(const char* s);              //类型转换构造函数
     //const String& operator=(const String& right);
     ~String();                          //析构函数
    void myout()  { cout<<sPtr<<endl; }
};
```

6.6 赋值操作符的重载

```
String::String(const char* s)
{  cout<<"类型转换构造函数called"<<endl;
   length = strlen(s);                    //计算字符串长度
   sPtr = new char[ length+1 ];           //分配内存
   strcpy(sPtr,s);                        //把字符串直接拷贝到对象中
}
String::~String()
{  delete[]  sPtr;                        //收回为对象分配的内存
}
```

6.6 赋值操作符的重载

```
void fun(String &aa)
{   String  c("789");
    c = aa;
}
void main()
{    String a(“12345”), b("xyz");
      b=a;                  //内存泄露， b.sPtr原所指 没有释放
      fun(a);               //a.sPtr将指向一个未定义的区域。
       a.myout();
       a = a;
}
```

6.6 赋值操作符的重载

在用户自定义的String类中含有指向动态存储区的指针，因此，如果直接使用缺省赋值函数，则可能产生以下问题：

1）当执行“b=a”是，由于b.sPtr被直接修改为指向a.sPtr的存储区，而b.sPtr原来所指向的动态存储区并没有释放，从而造成内存漏洞。

2）当执行“c=aa”后，指针a.sPtr和c.sPtr就指向了同一存储区。由于对象c是一个局部变量，那么，当对象c的生命周期结束时，将自动调用析构函数释放c.sPtr所指向的存储区，则此时a.sPtr将指向一个未定义的区域，从而造成指针悬浮。

可见，在以上两种情况下都必须由用户自己定义赋值函数operator=()来复制对象。

6.6 赋值操作符的重载

第一种情况，应在赋值前先释放掉b.sPtr，再分配其独立的内存区，拷贝a.sPtr所指内容；

第二种情况与第一种情况类似，给c.sPtr分配独立的内存区，以便在对象c消失时能正确地释放内存。

但这样做后又会产生一个新问题，即，当执行“a = a”时，如果先释放了左边a.sPtr的内存，则在赋值时右边的a.sPtr就不存在了。为了避免出现这种情况，则在赋值运算中还必须用this指针检测，在赋值号两边的是否是同一对象。如果是，则直接返回这个对象的引用，执行“return *this”。

在String类中，重载后的赋值运算符函数如下所示。

6.6 赋值操作符的重载

```
const String& String::operator=(const String& right)
{  cout<<"operator =()called"<<endl;
   if(this != &right )                          //避免自我赋值
    {  delete[]  sPtr;                         //防止内存漏洞
       length=right.length;                    //取新字符串长度
       sPtr=new char[length+1];                //分配内存
       for (int i=0;i<length;i++)
         sPtr[i]=right.sPtr[i];
       //strcpy(sPtr,right);                   //拷贝字符串
    }
    else       cout<<"Attempted assignment of a String to itself\n";
    return   *this ;                           //保证能够连续赋值
}
```

6.6 赋值操作符的重载

- 运行结果：

  类型转换构造函数called

  类型转换构造函数called

  operator =()called

  类型转换构造函数called

  operator =()called

  12345

  operator =()called

  Attempted assignment of a String to itself

目录

6.7 特殊操作符的重载

- 要介绍的特殊操作符包括：
  - 下标操作符[ ]
  - 函数调用操作符( )
  - 自增操作符++
  - 自减操作符- -
  - 转型操作符

6.7.1下标操作符的重载：

- 对下标运算符［］进行重载，只能重载为类成员函数,不可重载为友元函数。
- 重载下标运算符格式：

```
class C {
    returntype &operator[ ] ( paramtype );
};
```

或

```
class C{
    const    returntype &operator[ ] ( paramtype )  const;
};
```

该重载函数必须且只能带一个形参，且规定其参数值相当于下标值。

6.7.1下标操作符的重载：

- 对于一个类C的对象c，表达式

  c[i]

  将翻译为：

  c.operator[ ](i)

- 应用：

  c++并不检查数组的下标是否越界，但重载下标操作符时可以添加这种检查。

- 例 (下标操作符.cpp)

6.7.1下标操作符的重载：

```
#include <iostream>
using namespace std;
class  CharArray
{  public:
       CharArray(int len);
      ~CharArray();
       char&  operator[ ](int i);
       int getlength() const{ return  length; }
   private:
       int  length;
       char  *buff;    };
```

说明：由于数组函数可能会出现在赋值语句的左端，所以，被重载的下标运算符函数的返回值是一个char型的引用。

6.7.1下标操作符的重载：

```
CharArray::CharArray(int len)
{   length=len;
     buff=new char[length];  }
CharArray::~CharArray()   {    delete[]  buff;   }
char& CharArray::operator[ ](int i)
{   static char  ch=0;
    if(i>=0 && i<length)       return   buff[i];
    else
    {  cout<<"\nIndex out of range.\n“;              return  ch;    }
}
```

说明：在重载下标运算符［］时，首先检查当前下标i的范围。如果下标i越界，则发出一个提示信息，并返回一个对static型变量ch的引用。这样可以避免错误地修改内存区的内容。

6.7.1下标操作符的重载：

```
void main()
{  int j;
   CharArray string1(6);
   char *string2="abcdefg";
   for(j=0;j<7;j++)   string1[j]=string2[j];  // string2[6]保存在ch
   for(j=0;j<7;j++)  cout<<string1[j];
   cout<<"\nlength:"<<string1.getlength()<<endl;
 }
```

说明：在主函数main中，使用string2对string1逐个赋值时，虽然循环变量j的值超出了数组下标的范围，但赋值语句并不会对string1范围之外的内存区进行操作，因为在重载下标运算符［］时进行了下标范围检查。

6.7.1下标操作符的重载：

- 程序的运行结果是：

  Index out of range

  abcdef

  Index out of range

  g

  length:6

6.7.2 函数调用操作符的重载

- 与下标运算符 [ ]一样，函数调用运算符( )只能以成员函数的形式重载。
- 函数调用运算符可以带有一个或多个参数。
- 重载函数调用运算符格式

```
class C {
   returntype operator( )（ paramtypes );
};
```

- 对于一个类C的对象c，表达式

```
c ( x, name );
```

会被翻译为 `c.operator ( ) ( x , name )`

- 例

6.7.2 函数调用操作符的重载

例：假设有如下数学函数表达式： f(x,y)=2x+y; 要求通过重载函数调用运算符来实现以上函数的抽象。 (函数调用操作符.cpp)

```
class  F
{   public :
        double  operator()(double x,double y) const
        {    return    x*2+y;  }      };
void main()
 {  F  f;            cout<<f(1.5,3.2);    // f.operator()(1.5,3.2)      }
```

程序的运行结果为： 6.2

函数调用操作符重载函数 和 普通函数的区别：

普通函数，函数名固定；

函数调用操作符重载函数，“函数名”是类对象，可变化。

6.7.3 自增与自减操作符重载：

- 为区分自增、自减操作符前置和后置两种情形，通过不同的参数表来区分。
  - 默认的带有一个参数的++运算符函数是前缀++运算符。
  - 重载后缀的++运算符时应给函数多增加一个int参数，该int参数不会使用，只是为了让编译器区分++运算符是前缀还是后缀。

  即
  - operator ++( ); // 重载前置自增操作符
  - operator ++(int);//重载后置自增操作符
- 自减操作符的情况类似。

- 例

counter类的定义如下：

```
class counter {
    public:
      counter operator ++ ( );       //前缀操作符
      counter operator ++ ( int );  //后缀操作符
    private:
      unsigned value;    };
counter counter::operator ++ ( ) { value++;  return *this; }
counter counter::operator ++ ( int ) {
    counter t;
    t.value = value ++;
    return t;
}
```

clock类的定义如下：

```
#include<iostream>                //自增.cpp
using namespace std;
class Clock
{ public:
     Clock(int hour=0, int minute=0, int second=0)
     {  m_nHour=hour; m_nMinute= minute; m_nSecond=second; }
     Clock  operator ++();     //前置单目运算符重载
     Clock operator ++(int);  //后置单目运算符重载
private:
     int m_nHour, m_nMinute, m_nSecond;
  friend ostream& operator<<(ostream&,Clock);
};
```

```
Clock Clock::operator ++()   //前置单目运算符重载函数
{   m_nSecond++;
    if (m_nSecond >= 60)
    {    m_nSecond = m_nSecond-60;
         m_nMinute++;
         if(m_nMinute >= 60)
         {    m_nMinute = m_nMinute-60;
              m_nHour++;
              m_nHour = m_nHour%24;
         }
    }
    return *this;
}
```

```
//后置单目运算符重载，注意形参表中的整型参数
Clock Clock::operator ++(int)
{   Clock old=*this;
    ++(*this);
    return old;
}

ostream& operator<<(ostream& out,Clock c)
{return  out << c.m_nHour << ":" << c.m_nMinute
             <<":"<<c.m_nSecond;
}
```

```
int main()
{      Clock myClock(23,59,59);
       cout<<"First time output: "<<myClock<<endl;
       cout<<"Show myClock++\nBefore++: "<<myClock++;
       cout<<"\tAfter++: "<<myClock<<endl;
       cout<<"Show ++myClock: "<<++myClock<<endl;
       return 0;
}
```

运行结果：

```
First time output: 23:59:59
Show myClock++
Before++: 23:59:59      After++: 0:0:0
Show ++myClock: 0:0:1
```

例6-18 重载自增操作符的Clock类（没秒）

```
class Clock {
public:
    Clock( int = 12, int = 0, int = 0 );
    Clock tick();
    friend ostream& operator<<( ostream&, const Clock& );
    Clock operator++();       // ++c
    Clock operator++( int ); // c++
private:
    int hour;
    int min;
    int ap; // 0 is AM, 1 is PM
};

Clock::Clock( int h, int m, int ap_flag ) {
    hour = h;
    min = m;
    ap = ap_flag;
}
```

```
Clock Clock::tick()  {
    ++min;
    if  ( min  ==  60  )  {
        hour++;
        min  =  0;
    }
    if  ( hour  ==  13  )
        hour  =  1;
    if  ( hour  ==  12  &&  min  ==  0  )
        ap  =  !ap;
    return  *this;
 }
Clock Clock::operator++()  {
    return  tick();
}

Clock Clock::operator++( int n )  {
    Clock  c  =  *this;
    tick();

   return  c;
}
```

例6-18（p240）重载自增操作符的Clock类

```
ostream& operator<<( ostream& out, const Clock& c ) {
   out << setfill( '0' ) << setw( 2 ) << c.hour
       << ':' << setw( 2 ) << c.min;
   if ( c.ap )
      out << " PM";
   else
      out << " AM";
   return out;
}
```

后置自加的测试

```
int main() {
   Clock c, d;
   c = d++;
   cout << "Clock c: " << c << '\n';
   cout << "Clock d: " << d << '\n';
   //...
}
```

- 输出为：

```
Clock c: 12:00 AM
Clock d: 12:01 AM
```

前置自加的测试

```
int main() {
   Clock c, d;
   c = ++d;
   cout << "Clock c: " << c << '\n';
   cout << "Clock d: " << d << '\n';
   //...
}
```

- 输出为：

```
Clock c: 12:01 AM
Clock d: 12:01 AM
```

6.7.4 转型操作符重载：

- 数据类型转换

  在程序设计中，常要对数据类型进行转换。

  对于系统预定义的基本数据类型，编译器知道如何进行自动类型转换。但是，如果需要对用户自定义类型的对象进行类型转换的话，则必须在程序中明确定义相应的类型转换函数。

  C++中，定义转换函数可以有两种方式：一是使用类型转换构造函数，另一种是使用类型转换成员函数。

- 类C的转型构造函数：

  只带一个形参的构造函数，可以把一个变量从任何其他数据类型转换成用户自定义的类型，这种构造函数称之为类型转换构造函数。

  注意：这种构造函数带有一个参数，不是C&类型；如果是C&类型，则为拷贝构造函数。

6.7.4 转型操作符重载：

```
class C{
    //...
    C( int );  //类型转换构造函数
    //...    };
int main( ){
     int i=8;
    C c;
    c=i;    //将int类型转换成C类型，再赋值
    //...
}
```

6.7.4 转型操作符重载：

例 一个使用类型转换构造函数的例子。

```
class counter
{   public:
        counter()  { iVal=0; }                  //缺省构造函数
        counter(int  v)  { iVal=v; }  //类型转换构造函数
        ~counter(){ }
    private:
        int iVal;
};
void myfun(Counter  ccc)
{       cout<<"用户自定义的函数\n“;        }
```

6.7.4 转型操作符重载：

```
void main()
{   counter   x=3,y(4);          \\直接调用转换构造函数
    counter   z;    int a=5;
    z=a;                         \\类型交换
    myfun(8);                    \\类型转换   }
```

本例说明了如何把一个int变量，转换为一个counter类对象：

在执行"counter x=3,y(4);"时，系统直接调用类型转换构造函数。

在执行“z=a”时，系统先调用转换构造函数，将int型变量a转换为counter类对象，然后再进行赋值。

在执行函数调用“myfun(8)”时，系统先用类型转换构造函数将实参进行转换，转换后成功再进行函数调用。

可见，系统在需要进程类型转换的时候，会自动地隐式调用类型转换构造函数。

6.7.4 转型操作符重载：

- 转型构造函数可以将其他类型转换成所需的类的类型。
- 如果要进行相反的转型动作，即要将类的类型转换成其他类型，可对转型操作符进行重载（类型转化成员函数）。转型操作符重载函数的声明语法如下：

```
operator othertype () { } ;
```

注意：声明中不能包含形参和返回类型。但函数体中必须包含return语句，用来返回转型的结果。另外，不能定义为类的友元函数。

6.7.4 转型操作符重载：

```
class counter  //转型操作符.cpp
{  public:
      counter()   {     iVal=0;  }            //缺省构造函数
      counter( int  v) { iVal=v; }     //类型转换构造函数
      ~counter(){ }
      operator  unsigned short()    //类型转换成员函数
      {   //返回 转换 类型的对象
          return static_cast<unsigned  short>(iVal);
       }
  private:
      int iVal;
 };
```

6.7.4 转型操作符重载：

```
void main()
{   counter  x(3);          unsigned short  a;
    a=x;  //将赋值号右边的类型转换为左边的类型
          //即将x隐形式转换为unsigned short类型
    // x=x+a              //可能出现二义性!
   cout<<a<<endl;
}
```

类型转换成员函数可以被继承，可以是虚函数，但不能被重载，因为他没有参数。

尽管类型转换在有些场合是必须的，但用户还是应当尽量避免不必要的类型转换。因为，当执行类型转换的时候，所设计的程序实际上正在躲避由C++编译器所提供的防止类型错误的安全检查。

6.7.4 转型操作符重载：

另，如果一个类型既有转型构造函数，又有转型成员函数，那么，在某些时候可能出现语法上的二义性。

对于这种出现二义性的场合，用户应显示的说明所用类型转换方法。例如，要将结果转换为counter类型,则上述语句可改为：

```
counter y=a;
x=x+y;
```

或

```
x=x+counter(a);
```

再看书上示例6-20（p.242）

例子：

- 用整数表示的时间称为军事时间，其规则为：
  - 1300代表1:00 P.M .
  - 1400代表2:00 P.M.
  - 0代表12:00 A.M.等。

  这样，如果时间是8:34 A.M.，其对应的整数值就是834；

  如果时间是4:08 P.M .，则对应1608。

请在前例clock类上添加转型操作符，使其能从clock类转成int类。

原clock类的定义如下：

clock类的定义如下：

```
class Clock {
public:
    Clock( int = 12, int = 0, int = 0 );
    Clock tick();
    friend ostream& operator<<( ostream&, const Clock& );
    Clock operator++();       // ++c
    Clock operator++( int ); // c++
private:
    int hour;
    int min;
    int ap; // 0 is AM, 1 is PM
};

Clock::Clock( int h, int m, int ap_flag ) {
    hour = h;
    min = m;
    ap = ap_flag;
}
```

例子：

```
class Clock {
public:
    operator int();
    //...
};
Clock::operator int() {
    int time = hour;
    if ( time == 12 )
        time = 0;
    if ( ap == 1 )
        time += 12;
    time *= 100;
    time += min;
    return time;
}
```

```
Clock c( 8, 55, 1 );

int i;
i = c;
cout << i << '\n';
```

输出结果为：2055

目录

示例程序：关联式数组

- 建立一个字典类Dict，以支持单词定义对，为其设计适当的成员函数，并重载下标操作符，以支持如下操作：

  d[“dog ”] = “a kind of animal”

  - 该操作表明：d对象的第dog个元素是a kind of animal；

    对字典来说就是dog，其意义是a kind of animal

- 这种下标不是整数的数组称为关联式数组（associative array）。
- 以下语句将打印出dog的意义，前提是dog在字典中，如果dog不在字典中，则打印一条错误消息。

  cout << d[“dog”];

示例程序：测试程序

```
int main() {
    // Create a dictionary of word-definition pairs
    Dict d;
    // Add some pairs
    d[ "residual fm" ] = "incidental fm";
    d[ "pixel" ] = "in a daffy state";
    // Print all pairs in the dictionary.
    cout << "\n\ndump\n" << d;
    // Change definition of pixel
    d[ "pixel" ] = "picture element";
    // Print all pairs in the dictionary.
    cout << "\n\ndump\n" << d;
    // Look up some words
    cout << "\n\nlookup\n\n";
    cout << d[ "residual fm" ] << '\n';
    cout << d[ "pixie" ] << '\n';
    cout << d[ "pixel" ] << '\n';
    return 0;
}
```

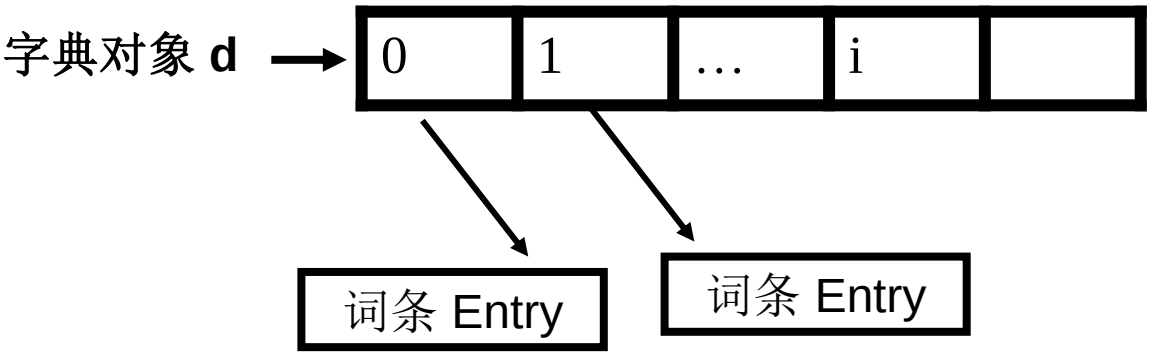
示例程序：输出

```
dump
residual fm defined as: incidental fm
pixel defined as: in a daffy state

dump
residual fm defined as: incidental fm
pixel defined as: picture element

lookup

residual fm defined as: incidental fm
pixie defined as: *** not in dictionary
pixel defined as: picture element
```

示例程序：关联式数组

- 解决方案

  辅助类Entry，用来储存单词、单词的意义和一个有效标记。有效标记代表当前的Entry对象是否存储了有效的单词及意义。




```
d["dog"] = "a kind of animal";
    ^                ^
  word             def
```

```
class Entry
{
private:
 string  word;
 string   def ;
 bool   flag;
};

class Dict
{
private:
   Entry entries[100];
};
```

示例程序：关联式数组

- 解决方案
  - Entry拥有如下成员函数：
    - 默认构造函数，该函数将有效标记初始化为false。
    - 用来添加单词及其定义的成员函数：Add。
    - 用来测试单词是否匹配的成员函数：match。
    - 获取有效标记状态的成员函数:valid。
    - 赋值操作符重载函数，将一个字符串（string或C风格）作为单词意义赋给Entry对象:operator=。
    - 重载操作符<<，以输出单词及其意义，将其作为Entry的friend函数:operator<<。

```
class Entry
{   private:
        string  word;
        string   def ;
        bool   flag;
};
```

示例程序：关联式数组

```
class Entry {
public:
    Entry() { flag = false; }
    void add( const string&, const string& );
    bool match( const string& ) const;
    void operator=( const string& );
    void operator=( const char* );
    friend ostream& operator<<( ostream&, const Entry& );
    bool valid() const { return flag; }
private:
    string word;
    string def;
    bool flag;
};
```

示例程序：关联式数组

```
void Entry::operator=( const string& str ) {
    def = str;
    flag = true;
}

void Entry::operator=( const char* str ) {
    def = str;
    flag = true;
}
ostream& operator<<( ostream& out, const Entry& e ) {
    out << e.word << " defined as: "
        << e.def;
    return out;
}
```

```
class Entry
{   private:
        string  word;
        string   def ;
        bool   flag;
};
```

示例程序：关联式数组

```
void Entry::add( const string& w, const string& d ) {
    word = w;
    def = d;
}

bool Entry::match( const string& key ) const {
    return key == word;
}
```

```
class Entry
{   private:
        string  word;
        string   def ;
        bool   flag;
};
```

示例程序：关联式数组

- Dict类只有一个数据成员，即类型为Entry的数组，用来存储单词及其意义。
- Dict类重载了下标操作符。
- Dict声明了一个enum常量MaxEntries，用来定义Entry数组。
- 采用friend方式为Dict类设计了一个顶层的<<操作符重载函数，用来输出字典中所有的元素。

示例程序：关联式数组

```
class Dict {
public:
    enum { MaxEntries = 100 };
    friend ostream& operator<<( ostream&, const Dict& );
    Entry& operator[ ]( const string& );
    Entry& operator[ ]( const char* );
private:
    Entry entries[ MaxEntries + 1 ];
};
```

示例程序：关联式数组

```
Entry& Dict::operator[ ]( const string& k ) {
    for ( int i = 0; i < MaxEntries && entries[ i ].valid(); i++ )
        if ( entries[ i ].match( k ) )
            return entries[ i ];
    string not_found = "*** not in dictionary";
    entries[ i ].add( k, not_found );
    return entries[ i ];
}

Entry& Dict::operator[ ]( const char* k ) {
    string s = k;
    return operator[ ]( s );
}
```

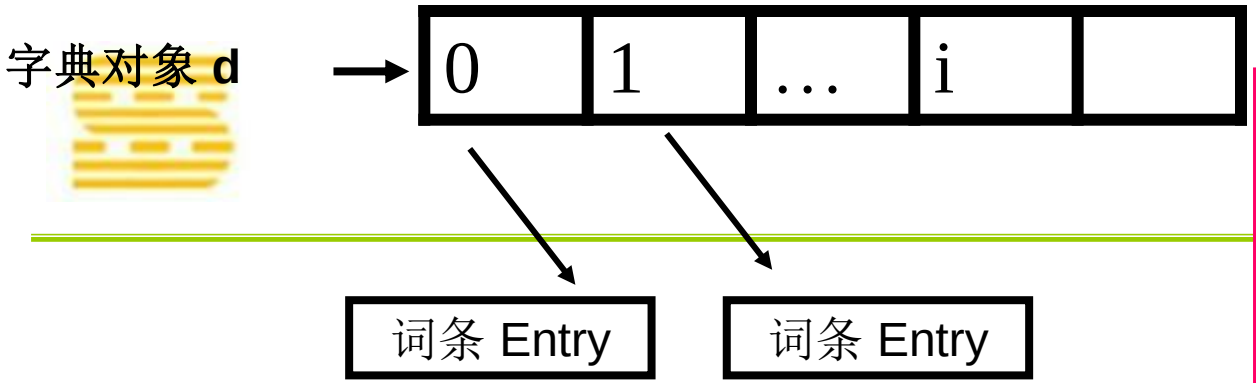
示例程序：关联式数组

```
ostream& operator<<( ostream& out, const Dict& d ) {

    for ( int i = 0; i < MaxEntries; i++ )
        if ( d.entries[ i ].valid() )
            out << d.entries[ i ] << '\n';
    return out;
}
```

示例程序：关联式数组

- 着重理解：

```
d[ "pixel" ] = "picture element";
```




```
class Entry
{
private:
 string  word;
 string   def ;
 bool   flag;
};

class Dict
{
private:
   Entry entries[100];
};
```

d[“pixel”]=“**picture element**”;

d.operator[ ](“pixel”) =“**picture element**”;

d.entries[i]=“**picture element**”;

d.entries[i].operator=(“**picture element**”);

d.entries[i].def =“**picture element**”;

```
Entry & Dict::operator[ ](string k)
{……
 return entries[i];   }
```

```
 void Entry::operator=(string  s)
{ def=s;
 flag=true;    }
```

目录

6.9 内存管理操作符

- 内存管理操作符

  new ，new[] , delete , delete[]

- new 操作符的重载方式有两种：

  void* C::operator new (size_t size)

  { }//重载为成员函数

  和 void* operator new(size_t size)

  { } //重载为顶层函数

  其中，size_t，是为程序移植方便而定义的，通常是

  typedef unsigned int size_t

  这样当不同编译器实现不一致的时候只需要改这一个地方。

  第一个参数必须是size_t类型，数值等于将被创建的对象的大小，对于new[]，数值等于将被创建的对象大小的总和。

6.9 内存管理操作符

- C::new 的调用方式为：

  C *c1=new C;

- operator new的参数size_t什么时候指定？

  是编译器隐藏传递的，是要new的东西所需要的字节数，这个值是编译期间获得的，并不需要程序员来传递。

6.9 内存管理操作符

- delete操作符的两种重载方式：

```
void  C::operator delete(void *objPtr)
  {        }
```

和

```
 void    operator delete(void *objPtr)
{                 }
```

第一个参数必须是void *类型

- C::delete 的调用方式是：

```
C* c1=new C;
delete c1;
```

6.9 内存管理操作符

例：

```
#include <iostream>
    using namespace std;
    class A
    {     public:  void* operator new (size_t size)  {
                                A *temp = ::new A(); cout<<"New...\n";
                                return temp;  }
                   void operator delete (void *p)  {
                                ::delete p; cout<<"Delete...\n"; }      };
    void main()   {  A *p = new A(); delete p;  }
```

运行结果：

```
    New...
    Delete...
```

内存管理操作符重载

- 例6-24：定义一个Frame类，用来描述一个数据帧，数据帧指的是数据通信应用程序中用于数据传输的基本单元
  - 为Frame设计了一个数据成员name ，用来对程序的执行情况进行跟踪。
  - 数据成员data保存了DataSize个字节的数据，用于数据传输。

内存管理操作符重载

```
const int MaxFrames = 4;
const int DataSize = 128;

class Frame {
public:
    Frame() { name = "NoName"; print(); }
    Frame( const char* n ) { name = n; print(); }
    Frame( const string& n ) { name = n; print(); }

    Frame( const string&, const void*, unsigned );
    void print() const;
    void* operator new( size_t );
private:
    string name;
    unsigned char data[ DataSize ];
};
```

内存管理操作符重载

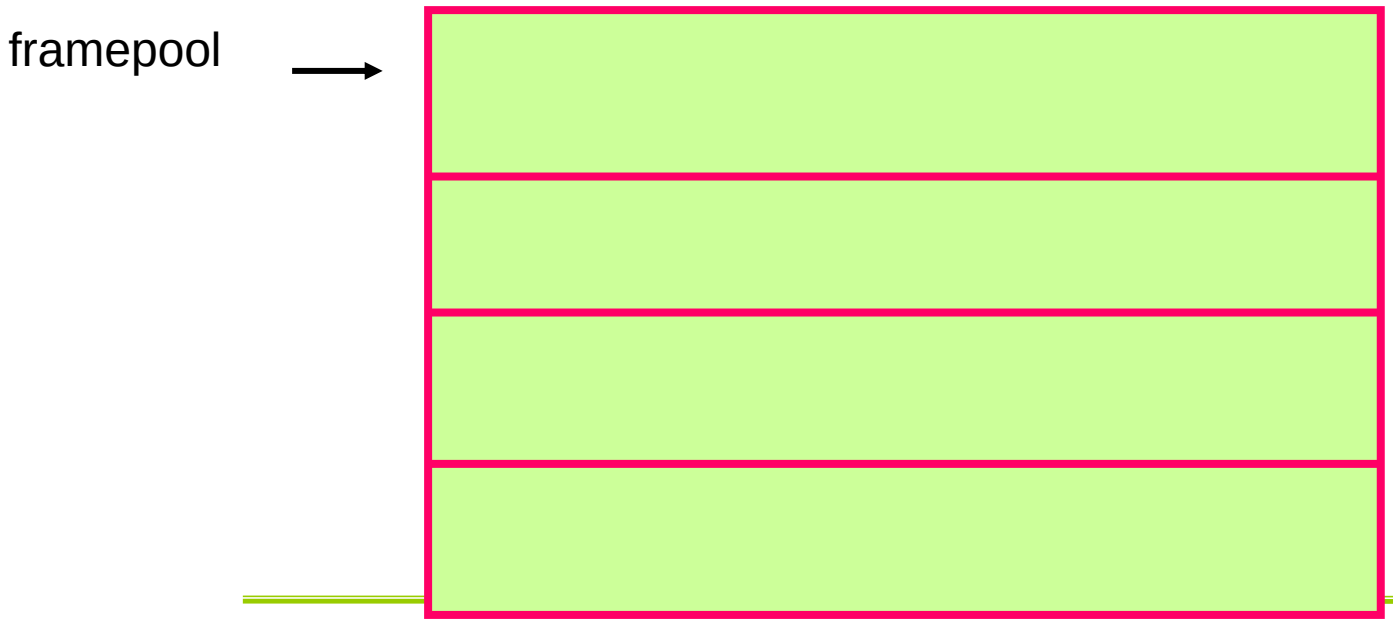
```
Frame::Frame( const string& n, const void* d,
              unsigned bsize ) {
    name = n;
    memcpy( data, d, bsize );
    print();
}

void Frame::print() const {
    cout << name << " created.\n";
}
```

- 解决方案：
  - 所有Frame对象均从framePool获取存储空间，大小为：

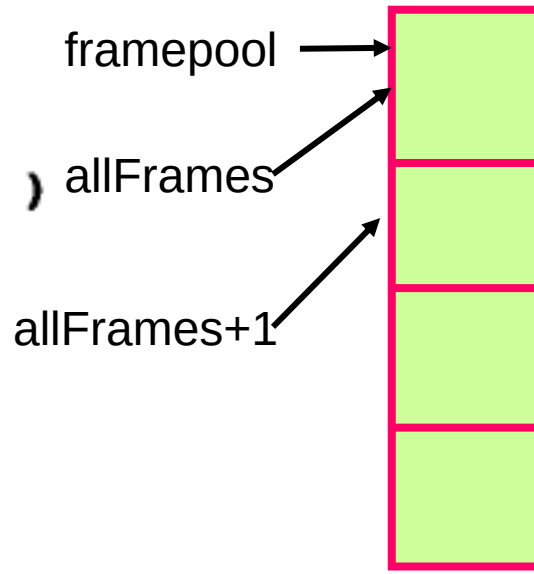
```
MaxFrames * sizeof(Frame)
```

  - 此framePool指向的空间可以保存Frame类型的MaxFrames个对象。对象一次分配一个：

```
Frame *f1 = new Frame("f1");
Frame *f2 = new Frame("f2");
```



```
Frame* allFrames = 0;  // no Frames yet
unsigned char framePool[ MaxFrames * sizeof( Frame ) ];
bool alloc[ MaxFrames ];  //标记是否已使用
void* Frame::operator new( size_t size ) {
    if ( size != sizeof( Frame ) )
        cout << "Not a Frame"
    if ( allFrames == 0 ) {  //初始化
        allFrames = ( Frame* ) framePool;
        for ( int i = 0; i < MaxFrames; i++ )
            alloc[ i ] = false;
    }
    for ( int i = 0; i < MaxFrames; i++ )
        if ( !alloc[ i ] ) {
            alloc[ i ] = true;
            return allFrames + i; //返回分配空间的起始位置
        }
    cout << "Out of Storage"
    return 0;
}
```



Frame的测试程序

```
int main() {
    Frame* a[ 5 ];
    string names[ 4 ] = { "f1", "f2", "f3", "f4" };
        a[ 0 ] = new Frame;
        for ( int i = 0; i < 4; i++ )
            a[ i + 1 ] = new Frame( names[ i ] );
    return 0;
}
```

Frame的测试程序的输出

```
NoName created.
f1 created.
f2 created.
f3 created.
Out of Storage
```

运算符重载小结

1）重载的运算符必须保持原有定义的优先级与结合性，使用语法、操作数个数也不能改变。

2）重载的目的是为了表达行为共享，所以重载的运算符的行为应保持与原有的操作符的行为的一致。

3）数学和物理学领域中的许多知识是用运算符来描述的，当将C++应用于这些领域时，重载相应的运算符是最恰当的，符合使用运算符的习惯。如果不是为了这个目的，则应使用函数

4）将一个运算符作用于对象时，C++编译器解释方式：

- 首先，试图将该运算符解释为类成员运算符；
- 其次，再试图解释为友员（顶层）运算符；
- 试图使用类中定义的转换函数将对象转换为其它的对象，以便能为该运算符调用一个合适的运算符函数；
- 最后，上述努力都失败后，编译器报错。