操作符重载

本章内容

- 操作符重载的需要性
- 作为成员函数重载
- 作为全局（友元）函数重载
- 一些特殊操作符的重载

操作符重载的需要性

- C++语言本身没有提供复数类型，可定义一个类来实现：

```
class Complex        //复数类定义
{public:
    Complex(double r=0.0,double i=0.0)
    { real=r; imag=i;
    }
    void display() const
    { cout << real << '+' << imag << 'i';
    }
    ......
  private:
    double real;
    double imag;
};
```

- 如何实现两个复数（类型为Complex）相加？

- 一种方案：为Complex类定义一个成员函数add，例如：

```
class Complex
{ public:
        Complex add(const Complex& x) const
        {    Complex temp;
             temp.real = real+x.real;
             temp.imag = imag+x.imag;
             return temp;
        }
     ......
};
......
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a.add(b);
```

- 另一种方案：定义一个全局函数，例如：

```
class Complex     //复数类定义
{  ......
   friend Complex complex_add(const Complex& x1,
                              const Complex& x2);
};
Complex complex_add(const Complex& x1,
                    const Complex& x2)
{  Complex temp;
   temp.real = x1.real+x2.real;
   temp.imag = x1.imag+x2.imag;
   return temp;
}
......
Complex a(1.0,2.0),b(3.0,4.0),c;
c = complex_add(a,b);
```

- 上面的加法操作不符合数学上的习惯：c=a+b。
- 如何用+号实现复数加法？

```
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a + b;
```

- 有两种重载方法
  - 成员函数
  - 全局函数

作为成员函数重载操作符

- 双目操作符重载
  - (1) 先定义operator # 函数

    定义格式

    class <类名>

    { .....

    <返回值类型> operator # (<类型>);

    };

    <返回值类型> <类名>::operator # (<类型> <参数>) { ..... }
  - (2) 然后将#作为二元操作符使用

    使用格式

    <类名> a;

    <类型> b;

    a # b 或，a.operator#(b)
  - 事实上， a # b 等价于a.operator#(b)，因此双目操作符重载可以函数重载的一种特殊形式，即先对函数operator#进行重载，然后当作双目操作符使用。

例：实现复数加法

```
class Complex
{ public:
    Complex operator + (const Complex& x) const
    {       Complex temp;
            temp.real = real+x.real;
            temp.imag = imag+x.imag;
            return temp;
    }
    ......
};
……
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a + b;
```

例:实现复数的“等于”和“不等于”操作

```
class Complex
{   double real, imag;
   public:
    ......
    bool operator ==(const Complex& x) const
    {   return (real == x.real) && (imag == x.imag);
    }
    bool operator !=(const Complex& x) const
    {   return !(*this == x);
    }
};
......
Complex c1,c2;
......
if (c1 == c2) //或 if (c1 != c2)
......
```

- 单目操作符重载
  - (1) 首先定义operator #() 函数，不需要参数

    定义格式

    class <类名>

    { ......

    <返回值类型> operator # ();

    };

    <返回值类型> <类名>::operator # () { ..... }
  - (2) 把#作为单目运算符使用

    使用格式

    <类名> a;

    #a

    或，

    a.operator#()

例：实现复数的取负操作

```
class Complex
{ ......
  public:
   ......
   Complex operator -() const
   {  Complex temp;
      temp .real = -real;
      temp.imag = -imag;
      return temp;
   }
};
......
Complex a(1,2),b;
b = -a;  //把b修改成a的负数。
```

操作符++和-- 的重载

- 操作符++和-- 有前置和后置两种用法，如果没有特殊说明，它们的前置用法与后置用法均使用同一个重载函数。
- 为了能够区分++和--的前置与后置用法，可为它们再写一个重载函数用于实现它们的后置用法，该重载函数有一个形式上的int型参数。例如：

```
class Counter
{   int value;
  public:
    Counter() { value = 0; }
    Counter& operator ++()  //前置的++重载函数
    { value++;
      return *this;
    }
    const Counter operator ++(int)  //后置的++重载函数
    { Counter temp=*this;
      ++(*this); //调用前置的++重载函数，或直接写成value++;
      return temp;
    }
};
.....
Counter a,b,c;
b = ++a;  //使用的是上述类定义中不带参数的操作符++重载函数
c = a++;  //使用的是上述类定义中带int型参数的操作符++重载函
  数
```

作为全局（友元）函数重载操作符

- 操作符也可以作为全局函数来重载，要求操作符重载函数的参数类型至少有一个为类、结构、枚举或它们的引用类型。
- 如果在操作符重载函数中需要访问参数类的私有成员（提高访问效率），则需要把它说明成相应类的友元。

作为全局（友元）函数重载操作符（续1）

- 双目操作符重载
  - (1)重载全局函数（友元函数）operator #，至少要有两个参数，定义格式为：

    <返回值类型> operator #(<类型1> <参数1>,
    <类型2> <参数2>)

    { …… }
  - (2)把#当作双目操作符来使用，使用格式为：

    <类型1> a;

    <类型2> b;

    a # b

    或

    operator#(a,b)

例：实现复数加法

```
class Complex
{ ......
   friend Complex operator + (const Complex& c1,
                              const Complex& c2);
};
Complex operator + (const Complex& c1,
                    const Complex& c2)
{ Complex temp;
   temp.real = c1.real + c2.real;
   temp.imag = c1.imag + c2.imag;
   return temp;
}
……
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a + b;
```

作为全局（友元）函数重载操作符（续2）

- 单目操作符重载
  - 定义全局函数（友元函数）operator #，有一个参数，定义格式为：

    <返回值类型> operator #(<类型> <参数>) { …... }
  - 把#当作单目操作符来使用，使用格式为：

    <类型> a;

    #a

    或

    operator#(a)
  - 也可以专门定义一个后置用法的重载函数：

    <返回值类型> operator #(<类型> <参数>,int) { … }

例：重载操作符+，使其能够实现实数与复数的混合运算。

```
class Complex
{     double real, imag;
   public:
      Complex() { real = 0; imag = 0; }
      Complex(double r, double i) { real = r; imag = i; }
      .......
   friend Complex operator + (const Complex& c1,
                              const Complex& c2);
   friend Complex operator + (const Complex& c,
                              double d);
   friend Complex operator + (double d,
                              const Complex& c);
};
```

```
Complex operator + (const Complex& c1,
                                const Complex& c2)
{  return Complex(c1.real+c2.real,c1.imag+c2.imag);
}
Complex operator + (const Complex& c, double d)
{  return Complex(c.real+d,c.imag);
}
Complex operator + (double d, const Complex& c)
//只能作为全局函数重载。
{  return Complex(d+c.real,c.imag);
}
......
Complex a(1,2),b(3,4),c1,c2,c3;
c1 = a + b;
c2 = b + 21.5;
c3 = 10.2 + a;
```

几个特殊操作符的重载

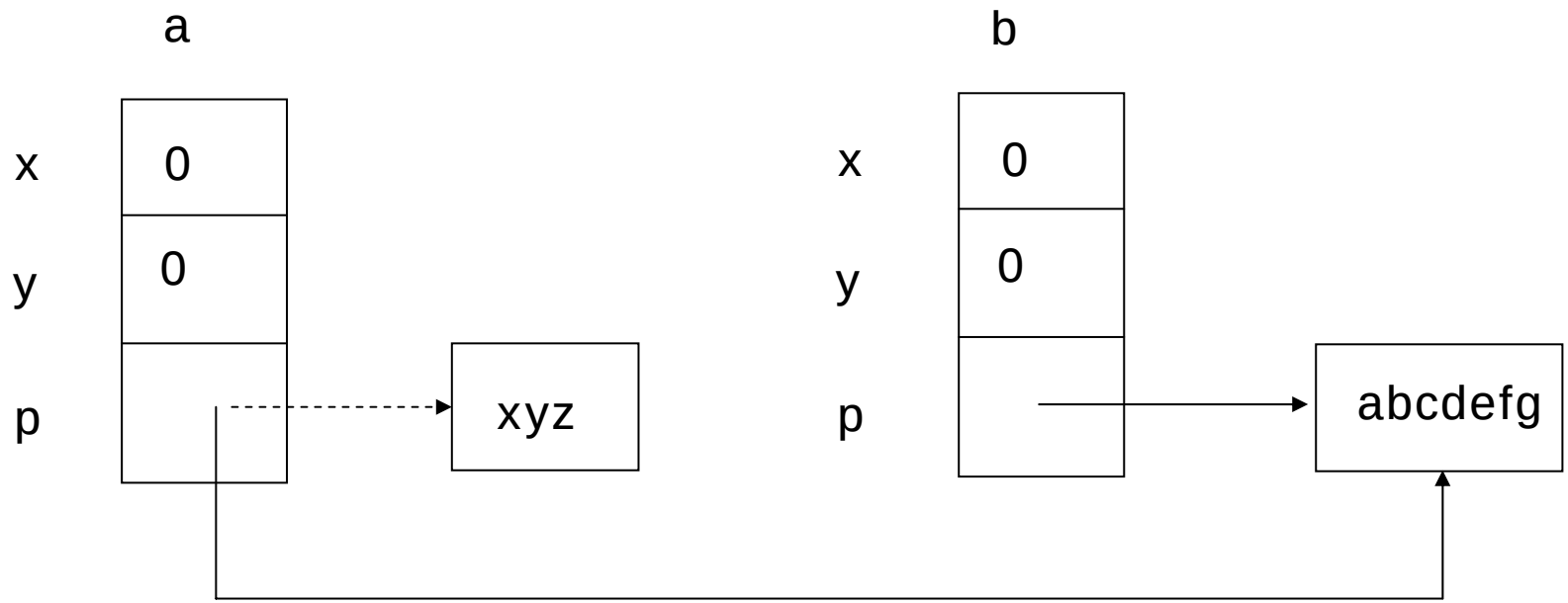
- 赋值操作符=
- 下标操作符[]
- 类成员访问操作符->
- 动态存储分配和去配new与delete
- 自定义类型转换操作符
- 函数调用操作符

赋值操作符=

- 对下面类A的对象进行赋值操作时存在问题：

```
class A
{   int x,y;
    char *p;
  public:
    A() { x = y = 0; p = NULL; }
    A(const char *str)
    {  p = new char[strlen(str)+1];
       strcpy(p,str);
       x = y = 0;
    }
    ~A()
    {  delete []p;
       p = NULL;
    }
};
```

```
A a("xyz"),b("abcdefg");
.......
a = b;   //赋值后，a.p原来所指向的空间成了“孤儿”。
```



- 解决上面问题的办法是自己定义赋值操作符重载函数：

```
A& A::operator = (const A& a)
{  if (&a == this) return *this;  //防止自身赋值。
   delete []p;
   p = new char[strlen(a.p)+1];
   strcpy(p,a.p);
    x = a.x; y = a.y;
   return *this;
}
```

自定义的赋值操作符重载函数不会自动地去进行成员对象的赋值操作，必须要在自定义的赋值操作符重载函数中显式地指出，例如：

```
class A { .......};
class B
{   A a;
    int x,y;
  public:
    ......
    B& operator = (const B& b)
    { a = b.a;//调用A类的赋值操符符重载函数来实现成员对象的赋值。
      x = b.x;
      y = b.y;
      return *this;
    }
};
```

- 赋值操作符只能作为非静态的成员函数来重载。
- 一般来讲，需要自定义拷贝构造函数的类通常也需要自定义赋值操作符重载函数。
- 注意拷贝构造函数和赋值操作符=重载函数的区别：

```
A a;
A b=a; //调用拷贝构造函数，它等价于：A b(a);。
.....
b = a; //调用赋值操作符=重载函数。
```

数组元素访问操作符（或下标操作符）[]

```
class String
{     char *p;
   public:
      ......
      char& operator [](int i)  //操作符[]的重载函数
      {     if (i >= strlen(p) || i < 0)
            {     cerr << "下标越界错误\n";
                  exit(-1);
            }
            return p[i];
      }
};
......
String s("abcdefg");
...s[i]... //访问s表示的字符串中第i个字符。
```

类成员访问操作符->

- “->”为一个双目操作符，其第一个操作数为一个指向类或结构的指针，第二个操作数为第一个操作数所指向的类或结构的成员。
- 通过对“->”进行重载，可以实现一种智能指针（smart pointers）。
- 重载“->”时需要按单目操作符重载形式来实现。

如何实现对A类对象的访问计数？

```
class A
{    int x,y;
  public:
     void f();
     void g();
};
```

- 重新定义A?

```
class A
{     int x,y;
      int count;
  public:
      A(){count=0;}
       void f(){count++;…}
       void g(){count++;…}
      int num_of_access() const {return count;}
};
```

```
class B  //智能指针类
{    A *p_a;
     int count;
   public:
     B(A *p)
     {    p_a = p;
          count = 0;
     }
     A *operator ->()  //操作符“->”的重载函数
     {    count++;
          return p_a;
     }
     int num_of_a_access() const
     {    return count;
     }
};
A a;
B b(&a);  //b为一个智能指针对象，它指向了a。
b->f();  //等价于：b.operator->()->f(); 即访问的是a.f()
b->g();  //等价于：b.operator->()->g(); 即访问的是a.g()
cout << b.num_of_a_access(); // 显示对象a的访问次数
```

操作符new与delete的重载

- 操作符new有两个功能：
  - 为动态对象分配空间
  - 调用对象类的构造函数
- 操作符delete也有两个功能：
  - 调用对象类的析构函数
  - 释放对象的空间
- 系统提供的new和delete操作所涉及的空间是在程序的堆区中分配和去配的，并且，堆空间是由系统管理的。
- 就某一个类的动态对象而言，系统的堆空间管理效率不高：
  - 系统要考虑程序中各种大小的堆空间申请和释放
  - 系统要处理“碎片”问题
- 可以对操作符new和delete进行重载，使得它们能以程序自己的方式实现动态对象的空间分配和释放功能。

- 重载操作符new
  - 操作符new必须作为静态的成员函数来重载（static说明可以不写），其格式为：
    - void *operator new(size_t size,…);
  - 重载函数的返回类型必须为void *，第一个参数表示所需空间的大小，其类型为size_t（unsigned int），其它参数可有可无。
  - new的使用格式与系统提供的基本相同。如果重载的new包含其它参数，其使用格式为：
    - A *p = new (...) A(...);
    - ... 表示提供给new重载函数的其它参数
    - ... 表示提供给构造函数的参数

例：重载new，把程序申请到的对象空间初始化为0

```
#include <cstring>
class A
{     int x,y;
   public:
      void *operator new(size_t size)
      {char *p=new char[size]; //调用系统堆空间分配操作。
        memset(p,0,size); //把申请到的堆空间初始化为全“0”。
        return p;
      }
      ......
};
A *p=new A; //对象的数据成员x和y被初始化为0。
```

```
#include <cstring>
class A
{       ......
   public:
        A(int i) { ... }
        void *operator new(size_t size,void *p)
        {       return p;
        }
};
char buf[sizeof(A)];
A *p=new (buf) A(0);//动态对象的空间分配为buf
.......
p->~A(); //使得p所指向的对象消亡。
```

重载delete

- 重载delete
  - 一般来说，如果重载了new，就应该重载delete
  - 操作符delete也必须作为静态的成员函数来重载（static说明可以不写），其格式为：

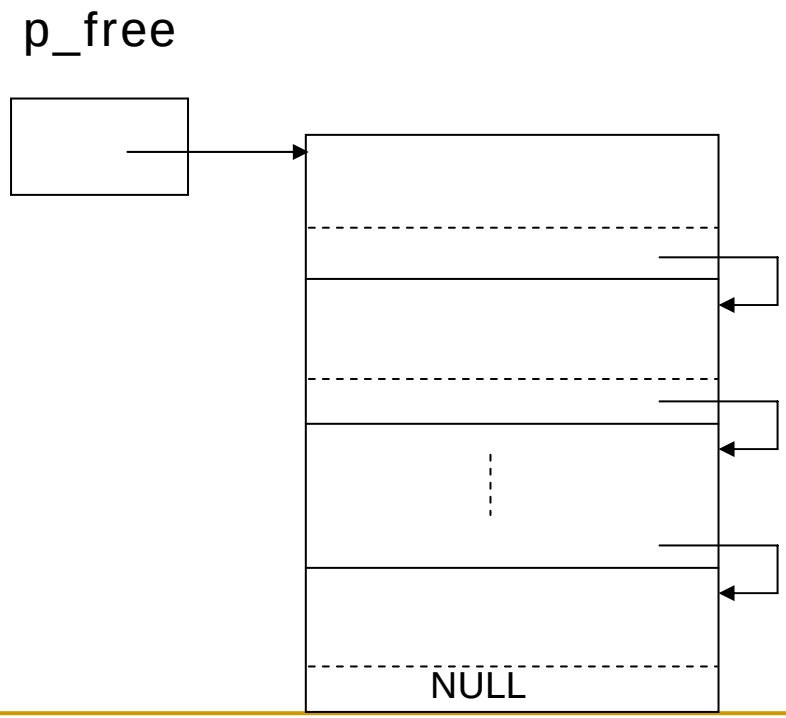
    void operator delete(void *p, size_t size);
  - 返回类型必须为void，第一个参数类型为void *，第二个参数可有可无，如果有，则必须是size_t类型。
  - 重载操作符delete的使用格式与未重载的相同。
  - delete重载只能有一个

例：重载操作符new与delete来管理某类对象的堆空间。

```
#include <cstring>
const int NUM=32;
class A
{   ..... //类A的已有成员说明。
  public:
    static void *operator new(size_t size);
    static void operator delete(void *p);
  private:
    static A *p_free; //用于指向A类对象的自由空间链表。
    A *next; //用于实现自由空间结点的链接。
};
A *A::p_free=NULL;
```

```
void *A::operator new(size_t size)
{  A *p;
   if (p_free == NULL)
   { p_free = (A *)new char[size*NUM];
                //申请NUM个A类对象的堆空间。
     for (p=p_free; p!=p_free+NUM-1; p++)
                //建立自由结点链表。
          p->next = p+1;
     p->next = NULL;
   }
   p = p_free;
   p_free = p_free->next;
   memset(p,0,size);
   return p;
}
```

```
void A::operator delete(void *p)
{ ((A *)p)->next = p_free;
  p_free = (A *)p;
}
```



自定义类型转换操作符

- 带一个参数的构造函数可以用作从一个基本数据类型或其它类到一个类的转换。 例如：

```
class Complex
{   double real, imag;
  public:
    Complex() { real = 0; imag = 0; }
    Complex(double r)  //一个参数的构造函数可兼作类型转换用
    {   real = r;
        imag = 0;
    }
    Complex(double r, double i) { real = r; imag = i; }
    ......
  friend Complex operator + (const Complex& x,
                             const Complex& y);
};
```

```
Complex operator + (const Complex& x,
                    const Complex& y)
{ Complex temp;
  temp.real = x.real+y.real;
  temp.imag = x.imag+y.imag;
  return temp;
}
.....
Complex c1(1,2),c2,c3;
c2 = c1 + 1.7;   //1.7隐式转换成一个复数对象Complex(1.7)
c3 = 2.5 + c2;   //2.5隐式转换成一个复数对象Complex(2.5)
```

- 自定义类型转换

```
class A
{  int x,y;
  public:
    ......
    operator int() { return x+y; }  //。
};
.....
A a;
int i=1;
int z = i + a; //将调用类型转换操作符int的重载函数
                    //把对象a隐式转换成int型数据。
```

```
class A
{    int x,y;
   public:
     A() { x =0;  y = 2; }
     A(int i) { x = i; y = 2; }
     operator int() { return x*y; }
     A operator +(const A &a1, const A &a2)
      { return a1.x+a2.x;
      }
};
......
A a(3);
int i=1,z;
z = a + i;  //是a转换成int呢，还是i转换成A?
```

对上面的问题，可以用显式类型转换来解决：

```
z = (int)a + i;    //这时,z=3*2+1=7
```

或

```
z = a + (A)i;    //这时,z=(3+1)*2=8
```

也可以通过给A类的构造函数A(int i)加上一个修饰符explicit来解决：

```
class A
{ ......
   explicit A(int i) //禁止把它当作隐式类型转换符来用。
   { x = i; y = 0; }
   ......
};
```

函数调用操作符

- 函数调用()可以看做一个操作符，并可以针对类进行重载
- 这样，相应类的对象就可以当作函数使用

```
class A
{ int value;
  public:
  A() {value=0;}
  A(int x){value=x;}
  int operator () (int x) //()的重载函数
   {  return x+value;
   }
};
…
A a(3);
cout <<a(10)<<endl;     //输出13
```

操作符重载的基本原则

- 只能重载C++语言中已有的操作符，不可臆造新的操作符。
- 可以重载C++中除下列操作符外的所有操作符：“.”，“.*”，“?:”，“::”，“sizeof”
- 重载不改变原操作符的优先级和结合性。
- 重载不能改变操作数个数。
- 尽量遵循已有操作符的语义（不是必需的）。
- 重载操作符时，其操作数中至少应该有一个是类、结构、枚举以及它们的引用类型。