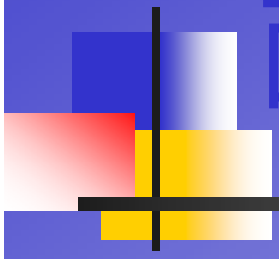
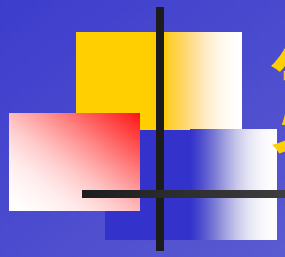


面向对象的程序设计

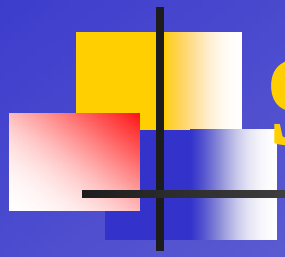
华中科技大学计算机学院
李瑞轩



第9章　运算符重载

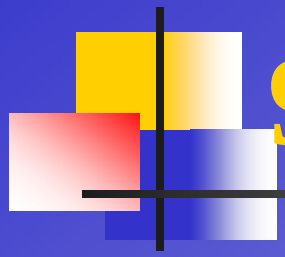
本章内容：

- 9.1 概述
- 9.2 运算符函数参数
- 9.3 赋值与调用
- 9.4 强制类型转换
- 9.5 重载new和delete
- 9.6 表运算实例



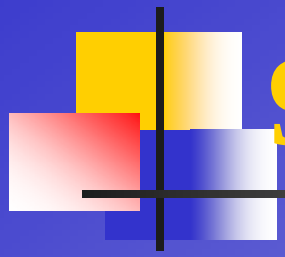
9.1 概述

- 运算符：也称为运算符函数，运算符的操作数则相当于函数的参数。
- C++根据操作数个数的不同将运算符分为单目、双目和三目运算符：
  - 纯单目运算符，只能有一个操作数，包括：
    !、~、sizeof、new、delete 等
  - 纯双目运算符，只能有两个操作数，包括：
    [ ]、->、%、= 等
  - 三目运算符，有三个操作数，包括：
    ?:
  - 既是单目又是双目的运算符，包括：
    +、-、&、* 等



9.1 概述

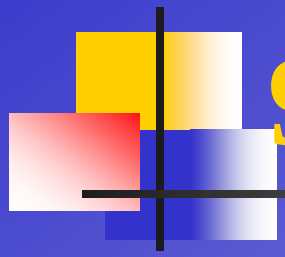
- 左值运算符：是运算结果为左值的运算符。其构成的表达式可以出现在等号左边。如前置运算++、--以及赋值运算=、+=、*=和&=等均为左值运算符。
- 某些运算符要求操作数为左值，如前置运算++和--。
- 还有一些运算符要求第一个操作数为左值，如赋值运算=、+=、*=和&=等。



9.1 概述

【例9.1】左值运算符的用法。

```
void main(void)
{
    int x=0;
    ++x;
    ++ ++x;      //++x仍为左值，故可连续运算，x=3
    --x=10;      //--x仍为左值，故可再次赋值，x=10
    (x=5)=12;  //x=5仍为左值，故可再次赋值，x=12
    (x+=5)=7;  //x+=5仍为左值，故可再次赋值，x=7
}
```



9.1 概述

- 运算符重载：将运算符看成是预定义的关于简单类型的运算函数，运算符重载就是对这些函数进行的关于类的对象的运算的重载。
- 运算符重载：用关键字operator加上参数(主要是)参数类型的不同来说明重载。
- 【例9.2】将加法运算符重载为普通函数。

```
#include <iostream.h>
class A{
    int x;
public:
    int getx () const{ return x; }
    A(int x) { A::x=x; }
};
```

9.1 概述

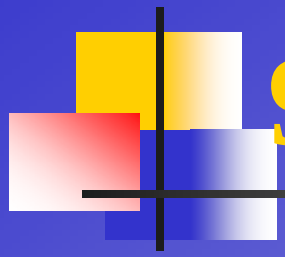
```
int operator+(const A &x, int y){  return x.getx()+y; }
int operator+(int y, A x){ return x.getx()+y; }
void main(void)
{
    A a(6);
    cout<<"a+7="<<a+7<<"\n";
    //调用int operator+(const A&, int)
    cout<<"8+a="<<8+a<<"\n";
    //调用int operator+(int, A)
}
```

必须有关于类的对象

输出：

a+7=13

8+a=14



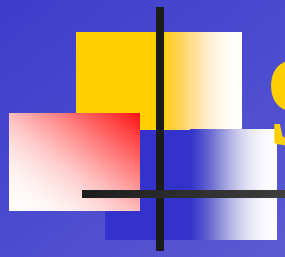
9.1 概述

- 根据能否重载及重载的函数类型，可将运算符分为：
  - 不能重载的运算符：

    sizeof、.、.*、::、?:
  - 只能重载为普通函数成员的运算符：

    =、->、()、[]
  - 不能重载为普通函数成员的运算符：

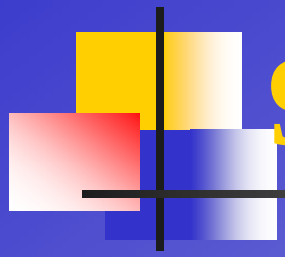
    new、delete
  - 其他运算符：都不能重载为静态函数成员，但可以重载为普通函数成员和普通函数。



9.1 概述

- 运算符=、–>、( )和[ ]不能重载为普通函数，没有普通函数也就不能定义普通友元：

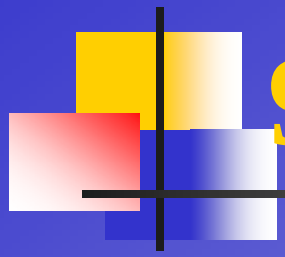
```
int operator=(int, A);                  //错误，不能重载为普通函数
int operator( )(A, int);                //错误，不能重载为普通函数
int operator[ ](A, int);                //错误，不能重载为普通函数
class A{
    friend int operator=(int,A);        //错误，不能定义为普通友元
    friend int operator()(A,int);       //错误，不能定义为普通友元
    friend int operator[](A,int);       //错误，不能定义为普通友元
    friend int operator += (int,A);     //正确
};
```



9.1 概述

- 运算符=、–>、( )和[]也不能重载为静态函数成员，因此，不能在类中声明或定义如下静态函数成员：

```
class B{
    static int operator+=(int, B);    //错误，不能定义为静态函数成员
    static int operator=(int, B);     //错误，不能定义为静态函数成员
    static int operator( )(B, int);   //错误，不能定义为静态函数成员
    static int operator[ ](B, int);   //错误，不能定义为静态函数成员
    static operator int(B);           //错误，不能定义为静态函数成员
};
```



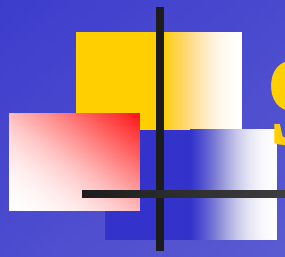
9.1 概述

- 运算符重载约定：关于单个对象的运算
  - 将运算符函数重载为普通函数(C函数)时，参数中至少要有一个类或类的引用类型；

```
int operator+(int, A);          //正确
int operator+(const A&, int); //正确
```

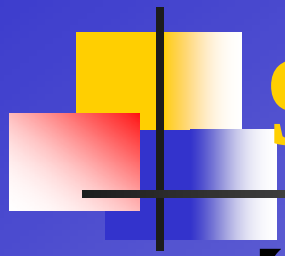
  - 不能将重载运算符函数的参数类型定义为对象指针(简单类型)或者对象数组类型；

```
int operator+(A *, int);        //错误
int operator+(A[6], int);       //错误
```



9.1 概述

- 运算符重载约定（续）：
  - 若运算符为左值运算符，则重载后运算符函数最好返回非只读引用类型；
  - 重载为普通函数(C函数)的运算符可以声明为类的普通友元；
  - 重载不改变运算符的优先级和结合性；
  - 重载一般也不改变运算符的操作数个数。（特殊的运算符除外:->，++，--）



9.1 概述

【例9.3】重载运算符为普通函数并声明为类的普通友元。

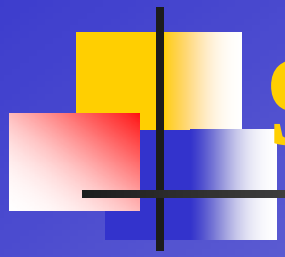
```
class A{
    int x, y;
public:
    A(int x, int y) { A::x=x; A::y=y; }
    A &operator=(A);                    //等号运算符重载为左值运算符
    friend A operator–(A);
    friend A operator+(const A &, const A &);
};
A operator–(A a) {                      //重载为单目运算符（右值）
    return A(–a.x, –a.y);               //A(–a.x, –a.y)为A类常量对象
}
A &A::operator=(A y) {                  //this指向等号左边的操作数（左值）
```

9.1 概述

```
    A::x=y.x;
    A::y=y.y;
    return  *this;      //返回一个引用表示等号为左值，可连续运算(m=n)=p
}
A operator +(const A&x, const A&y) {//重载双目加法为(右)值运算符函数
    int u=x.x+y.x, v=x.y+y.y;
    return A(u, v);  //A(x.x+y.x, x.y+y.y)为类A的常量对象
}
void main(void){
    A a(2,3), b(4,5), c(1, 9);
    c=a+b;
    (c=a+b)=b+b;   //赋值运算重载为左值，可出现在等号左边(连续运算)
    c= –b;
}
```

9.2 运算符函数参数

- 不同的重载形式函数的参数表列出的参数个数不同。
  - 重载为普通函数：
    参数个数 = 运算符目数
  - 重载为类的普通函数成员：（存在this指针）
    参数个数 = 运算符目数 - 1
  - 重载为类的静态函数成员：（没有this指针）
    参数个数 = 运算符目数
- 注意有的运算符既为单目，又为双目的：*, +,-
- 对于某些特殊的运算符，其重载函数参数个数与其目数之间可能不满足上述关系。



9.2 运算符函数参数

- 在重载运算符函数时：
  - 如果函数的参数实际有两个（包括隐含this参数），则重载的运算符函数为双目运算符函数；
  - 如果函数的参数实际只有一个（包括隐含this参数），则重载的运算符函数为单目运算符函数。

9.2 运算符函数参数

- 运算符++和--有前置运算和后置运算两种运算形式，重载时必须使用不同的函数原型加以区分。
- 将后置运算重载为返回右值的双目运算符函数：
  - 如果重载为类的普通函数成员，则该函数只需定义一个int类型的参数；(包含不用const修饰的this参数)
  - 如果重载为普通函数(C函数)，则最好声明不用const引用类的和int类型的两个参数。(无this参数)
- 将前置运算重载为返回左值的单目运算符函数：
  - 前置运算是先运算再取值，运算结果为左值，因此其返回类型应该定义为非只读类型的引用类型。
  - 如果重载为普通函数(C函数)，则最好声明不用const的引用类的一个参数。(无this参数)

9.2 运算符函数参数

- 无论是前置运算还是后置运算，运算符++和--都会改变对象的值，在重载时，最好将与该对象相应的参数定义为非只读引用类型，以便函数返回时能通过换名变量带回执行结果。
- 【例9.4】重载运算符++和--运算。

```
#include <iostream.h>
class A{
    int  a;
    friend A &operator--(A&);
    //仅一左值参数，返左值，前置运算
    friend A operator--(A&,int);
    //两个参数，为后置运算
public:
    int get( ) { return a; }
    A &operator++( );
    //只有this，返回左值，前置运算
    A operator++(int);
   //包括this共两个参数，为后置运算
    A(int x) { a=x; }
};
```

9.2 运算符函数参数

```
A &operator--(A&x) { //前置--，操作数x左值，返回引用对象(左值)
   x.a--;                    //先运算后取值，返回左值可连续运算
   return x;
}
A operator--(A&x, int)     //后置--，操作数x左值引用，返回右值对象
{//后置运算先取值x.a，构造的常量A(x.a)做右值返回，
 //后置运算后运算，x.a--响当前操作数x，返回时当前对象x被修改
   return  A(x.a--);         //先取值后运算，返回右值不可连续运算
}
A &A::operator++( )        //前置++，this无const修饰，返回左值引用对象
{//前置运算结果的左值等于当前对象，且返回的引用应为当前对象，
 //以便连续运算，不能引用局部对象，故必须用return *this返回
   a++;                      //前置++，先运算，后取值返回
   return  *this;            //返回值为当前对象的引用，左值可连续运算
}
```

9.2 运算符函数参数

```
A A::operator++(int){       //后置++，返回对象(右值)
   return  A (a++);          //后置++，先取值构造，后运算
}
void  main(void){
   A a(5);

   cout<<"a.a="<<(--a).get( )<<"\n";
   cout<<"a.a="<<(++a).get( )<<"\n";

   cout<<"a.a="<<(a--).get( )<<"\n";
   cout<<"a.a="<<(a++).get( )<<"\n";
}
```

输出：
a.a=4
a.a=5
a.a=5
a.a=4

9.2 运算符函数参数

- 重载双目运算符–>：
  - 只能重载为有一个参数，且返回类型必须为指针或引用类型的普通函数成员。
  - 该运算符是唯一可以重载为单目运算的纯双目运算符。
- 【例9.5】重载运算符–>，使其返回类型为指针类型。

```
struct A{
    int a;
    A(int x) { a=x; }
};
class  B{
    A  x;
public:
    A *operator –>( );
    B(int v):x(v) { }
};
```

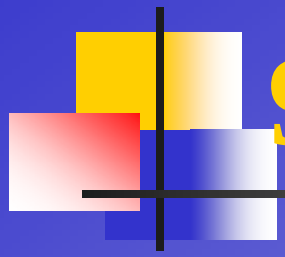
9.2 运算符函数参数

```
A *B::operator–>( )          //只有一个this参数，故重载为单目函数
{
   return &x;
}
void  main(void)
{
   B  b(5);
   int i=b–>a;                //i=b.x.a=5
   b->a=i+5;                  //b.x.a=10
   i=(*b.operator->( )).a;    //i=b.x.a=10
   i=b.operator –>( )–>a;     //i=b.x.a=10
}
```

9.2 运算符函数参数

- 运算符–>除了可以重载为指针外，还可以重载为引用。
- 【例9.6】重载运算符–>，使其返回类型为引用类型。

```
struct A{
    int a;
    A(int x) { a=x; }
};
class  B{
    A  x;
public:
    A &operator –>( );
    B(int x):x(x) { };
};
A &B::operator–>( ) {
//有一个this参数，
//故重载为单目运算符
    return x;
}
void  main(void){
    B  b(5);
    A &x=b.operator–>( );
    //等价于A &x=b.x
    int i=x.a;
    //i=x.a=b.x.a=5
    x.a=i+10;
    //b.x.a=x.a=15
    i=b.operator –>( ).a;
    //i=b.x.a=x.a=15
}
```



9.2 运算符函数参数

- 有些运算符既可作单目运算又可作双目运算，重载时必须注意运算符函数的实际参数个数。

- 对于纯双目运算符：重载为普通函数成员时，由于普通函数成员都有一个隐含参数this，因此重载的运算符函数成员仅能另外定义一个参数。

- 对于纯单目运算符：重载为普通函数成员时，则不能另外定义参数。

9.2 运算符函数参数

【例9.7】重载纯单目和纯双目运算符 。

```
class A{
    int x;
public:
    A (int y) { x=y; }
    A operator %(A m) const { return A(x%m.x); }
    A operator !( ) const { return A(!x); }
};
void main(void)
{
    A a(5), b(3);
    b=a%b;
    b=!a;
}
```

9.2 运算符函数参数

- 对于既能作单目运算又能作双目运算的运算符：若同时将其重载为普通函数和普通函数成员，则必须根据实参和形参的匹配结果，确定参与计算的运算符是在作单目运算还是在作双目运算。
- 【例9.8】运算符函数的重载与调用。

```
class POINT{
    int x, y;
public:
    POINT operator–( );
    POINT operator–(POINT);
    POINT operator+(POINT);
```

9.2 运算符函数参数

```
friend POINT operator+(POINT);
        friend POINT operator+(POINT, POINT);
    };
//在此进行成员函数的定义
void main( ){
    POINT p1, p2;
    POINT p3= –p1;                          //调用POINT operator–( )
    POINT p4=p1–p2;                         //调用POINT operator–(POINT)
    POINT p5=p1.operator– (p2);             //调用POINT operator–(POINT)
    POINT p6=+p1;                           //调用friend POINT operator+(POINT)
    POINT p7=operator+(p1,p2);//调用friend POINT operator+(POINT,POINT)
    POINT p8=p1.operator+(p2);              //调用POINT operator+(POINT)
    POINT p9=p1+p2;                         //错误，无法确定调用友元还是函数成员
}
```

9.3 赋值与调用

- 重载“=”运算符：
  - 编译程序为每个类提供缺省的赋值运算符函数；
  - 缺省的赋值运算实现数据成员的复制或浅拷贝；
  - 如果类自定义或重载了赋值运算函数，则优先调用类自定义或重载的赋值运算函数 。
- 【例9.9】编译程序提供了缺省赋值运算函数。

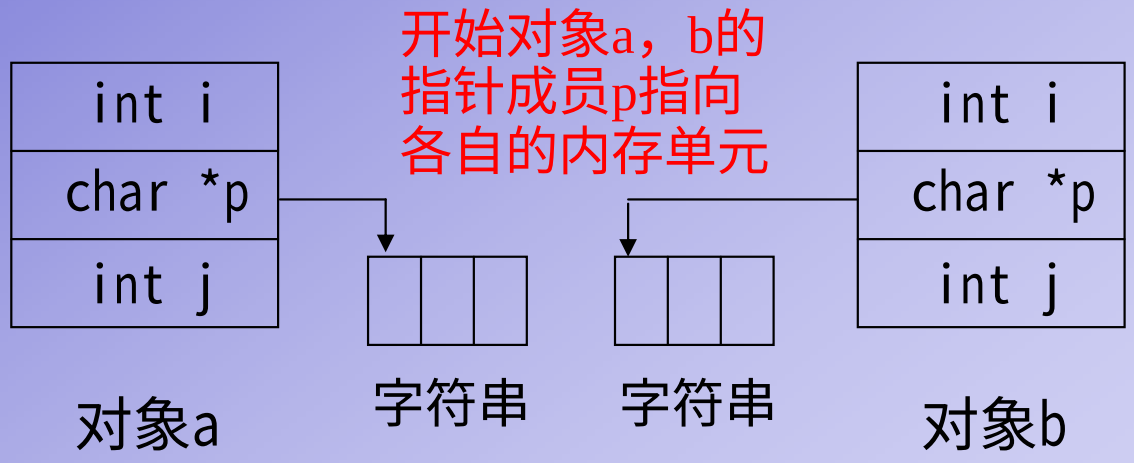
```
#include <iostream.h>
class A{
    int  a;
public:
    A(int x) { a=x; }
    A &operator =(A x){
        a=x.a; return *this;
    }
    int geta( ){ return a; }
};
```

9.3 赋值与调用

```
void main(void){
    A  a1(1), a2(2), a3(3);
    A &(A::*f)(A);              //定义一个函数成员指针
    A &(A::*g)(const A&); //定义一个函数成员指针
    f=&A::operator =;          //指向自定义的A &operator=(A)
    g=&A::operator =;          //指向缺省的A &A::operator=(const A &)
    cout<<"\na1="<<a1.geta( )<<"a2="<<a2.geta( )<<" a3="<<a3.geta( );
    (a1.*f)(a2);                    //等价于a1=a2
    cout<<"\na1="<<a1.geta( )<<" a2="<<a2.geta( )<<" a3="<<a3.geta( );
    ((a1.*f)(a2).*f)(a3);       //等价于(a1=a2)=a3
    cout<<"\na1="<<a1.geta( )<<" a2="<<a2.geta( )<<" a3="<<a3.geta( );
    (a1.*f)((a2.*f)(a3));       //等价于a1=a2=a3
    cout<<"\na1="<<a1.geta( )<<" a2="<<a2.geta( )<<" a3="<<a3.geta( );
}
```

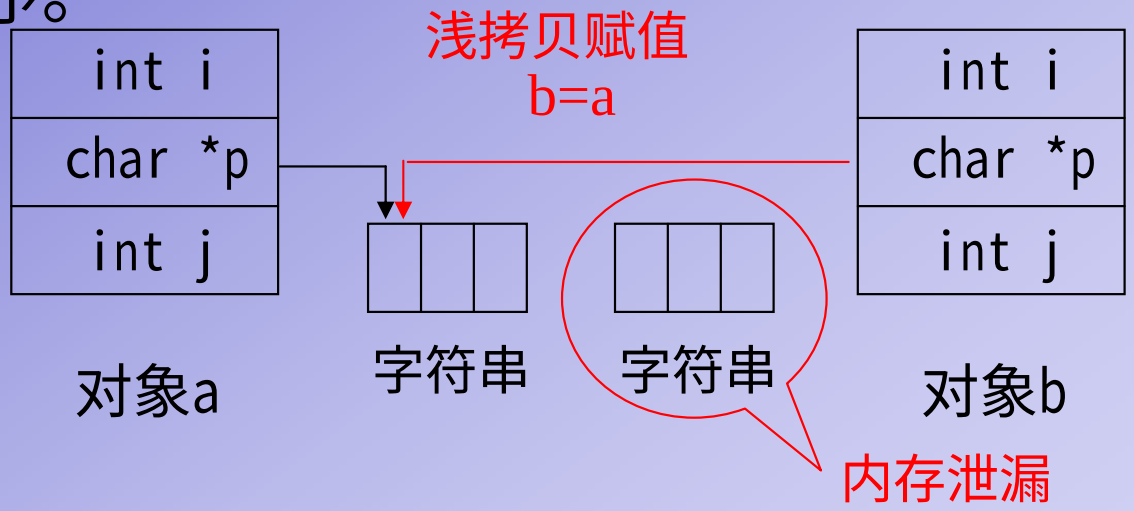
9.3 赋值与调用

- 浅拷贝问题：
  - 当对象的数据成员不含指针成员时，浅拷贝赋值不会出现任何问题；
  - 当对象的数据成员包含指针成员时，浅拷贝赋值可能造成内存泄漏，导致内存保护错误。



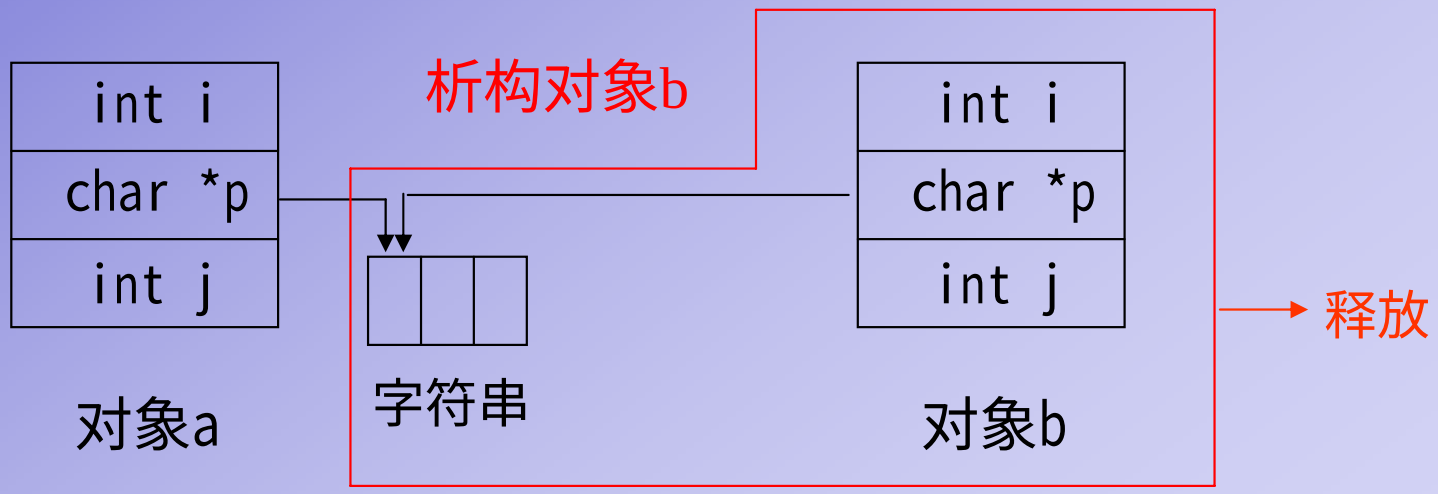
9.3 赋值与调用

- 浅拷贝问题：
  - 浅拷贝赋值造成b对象指向的字符串内存块泄露；
  - 当内存泄漏累计过多时，执行新的应用程序，操作系统会提示内存不足，请你退出一些应用程序。



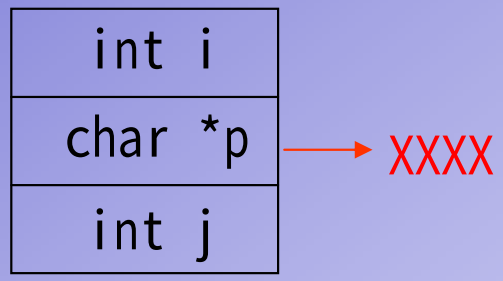
9.3 赋值与调用

- 浅拷贝问题：
  - 对象b析构时，释放了a、b共同指向的存放字符串的内存，这块内存很可能被其他程序分配；
  - 若这块内存被其他程序分配，本程序通过a访问这块内存时将导致内存页面保护错误。



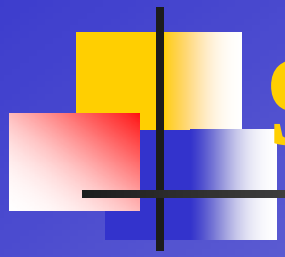
9.3 赋值与调用

- 浅拷贝问题：
  - 如果本程序分配b释放的内存块，并且分配后对内存块进行了初始化，对a来说产生了副作用；
  - 同理，a也可能对本程序其他指针变量分配的内存产生副作用。



对象a

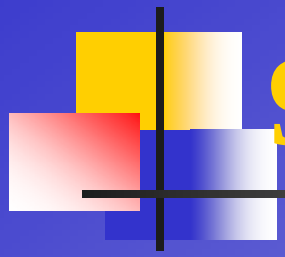
对象b的释放使得对象a的指针成员p指向的存储单元被释放。若该地址的内存被分配给其他程序，继续访问对象a的指针成员p指向的内容时将出现页面保护错误



9.3 赋值与调用

【例9.10】定义一个字符串类，实现字符串连接运算。

```
#include <iostream.h>
#include <string.h>
class STRING{
    char *s;
public:
    STRING(char *s);
    STRING operator+(const STRING &)const;
    ~STRING( );
};
STRING::STRING(char *str){
    s=new char[strlen(str)+1];
```



9.3 赋值与调用

```
    strcpy(s, str);
    cout<<"Construct: "<<str<<"\n";
}
STRING::~STRING( ){
    cout<<"Delete: "<<s<<"\n";
    delete [ ]s;
}
STRING STRING::operator+(const STRING &str)const{
    char *t=new char[strlen(s)+strlen(str.s)+1];
    STRING r(strcat(strcpy(t,s),str.s));
    delete [ ]t;
    return r;
}
```

9.3 赋值与调用

```
void func(STRING  s){ return; }
void main(void){
    STRING s1("S1");      //等价于s1="S1"
    STRING s2="S2";       //等价于s2("S2")
    STRING s3="S3";
    s1=s2;                //浅拷贝赋值s1
    cout<<"Call func\n";
    func(s3);             //浅拷贝构造参数s
    cout<<"Exit main\n";
}
```

输出：

Construct: S1

Construct: S2

Construct: S3

Call func

Delete: S3

Exit main

Delete: S3

Delete: S2

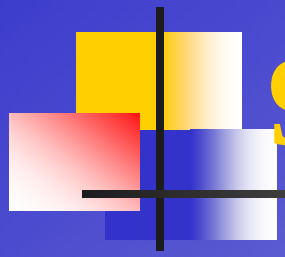
Delete: S2

9.3 赋值与调用

- 问题的分析：

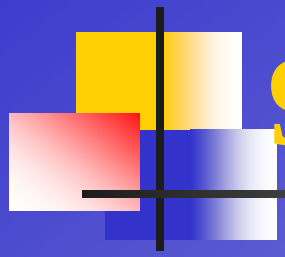
```
void main(void)
{
    STRING s1("S1");   //s1.s指向某块内存A
    STRING s2="S2"     //s2.s指向某块内存B
    STRING s3="S3";    //s3.s指向某块内存C
    s1=s2;             //s1.s指向s2.s的内存B
    cout<<"Call func\n";
    func(s3);          //s.s=s3.s

    cout<<"Exit main\n";
    }
```

```
void func(STRING  s)
{//s.s指向s3.s的内存
   return;
 //释放s.s即s3.s的内存
}
```



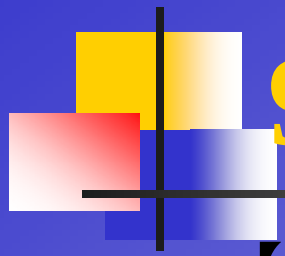
9.3 赋值与调用

- 浅拷贝问题的解决：
  - 当类的对象包含成员指针时，必须重载拷贝构造函数和等号运算符，以便该类的对象进行深拷贝构造和赋值操作。
- 安全的类：
  - 如果要定义包含指针成员的类将作为其他类的基类，并且希望通过父类指针进行多态析构，则基类的析构函数就应该声明为虚函数，其他普通函数成员也尽可能地定义为虚函数，这样定义的类才是一个安全的类。



9.3 赋值与调用

- 具体来说，定义包含指针成员的类T应注意以下几点：
  - 应定义T(const T &)等形式的深拷贝构造函数；
  - 应定义virtual T &operator=(const T &)等形式的深拷贝赋值运算函数；
  - 应定义virtual ~T( )形式的虚析构函数；
  - 在定义引用T &p=*new T( )后，要使用delete &p释放对象占用的内存；
  - 在定义指针T *p=new T( )后，要使用delete p释放对象占用的内存。



9.3 赋值与调用

【例9.11】定义字符串类，实现字符连接、比较以及字符取值赋值运算。

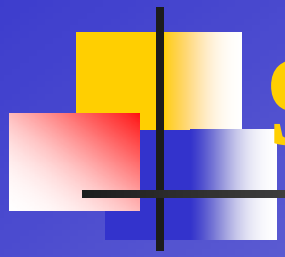
```
#include <iostream.h>
#include <string.h>
class STRING{
    char *s;
public:
   virtual int operator>(const STRING &c)const
   { return strcmp(s,c.s)>0;}
   virtual int operator==(const STRING &c)const
   { return !strcmp(s,c.s);}
   virtual int operator<(const STRING &c)const
   { return strcmp(s,c.s)<0;}
   virtual char &operator[ ](int x){ return s[x]; }
```

9.3 赋值与调用

```
    virtual operator const char *( )const        //强制类型转换重载
    { return s; }
    STRING(const char *c){ strcpy(s=new char[strlen(c)+1], c); }
    STRING(const STRING &c)
    { strcpy(s=new char[strlen(c.s)+1], c.s); }
    virtual STRING operator+(const STRING &)const;
    virtual STRING &operator=(const STRING &);
    virtual STRING &operator+=(const STRING&s)
    { return *this=*this+s;}
    virtual ~STRING( ) { if(s){ delete [ ]s; s=0; }}
};
STRING STRING::operator+(const STRING &c)const{
    char *t=new char[strlen(s)+strlen(c.s)+1];
    STRING r(strcat(strcpy(t,s),c.s));
```

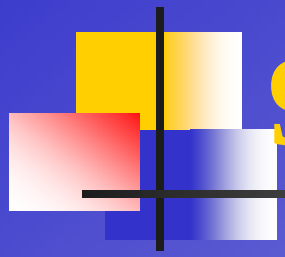
9.3 赋值与调用

```
    delete [ ]t;    return r;
}
STRING &STRING::operator=(const STRING &cs){
    delete [ ]s;
    strcpy(s=new char[strlen(cs.s)+1], cs.s);
    return *this;
}
void main(void){
    STRING s1("S1"), s2="S2",s3("S3");
    s1=s1+s2;
    s1+=s3;
    s3[0]='T';                          //调用char &operator[ ](int x)
    cout<<"s1="<<s1<<"\n";  //调用operator const char *( )得到s1.s
    cout<<"s3="<<s3<<"\n";  //同上得到s3.s
}
```



9.4 强制类型转换

- 函数调用是通过参数匹配完成的，运算符函数的调用也是这样。
- 如没有定义合适的类型转换函数，想要对象能与不同类型的数据进行运算，并且对象能随意出现在双目运算符的左边和右边，可能要重载很多运算符函数才能满足运算需求。



9.4 强制类型转换

【例9.12】定义复数类型，实现加法运算。

```
class COMPLEX{
    double r, v;
public:
    COMPLEX(double r1, double v1){ r=r1; v=v1; }
    COMPLEX operator+(const COMPLEX &c)const
    { return COMPLEX(r+c.r, v+c.v); };
    COMPLEX &operator+=(const COMPLEX &);
    COMPLEX operator+(double d)const
    { return COMPLEX(r+d, v); };
    COMPLEX &operator+=(double);
};
```

9.4 强制类型转换

```
COMPLEX &COMPLEX::operator+=(const COMPLEX &c){
    r+=c.r;  v+=c.v;    return *this;
}
COMPLEX &COMPLEX::operator+=(double d){
    r+=d;    return *this;
}
void main(void){
    COMPLEX c1(2, 4), c2(3, 4);
    c1=c1+c2;   //调用COMPLEX operator+(const COMPLEX &)
    c1=c1+'2';  //调用COMPLEX operator+(double d)
    c1+=c2;     //调用COMPLEX &operator+=(const COMPLEX &)
    c1+=2.5;    //调用COMPLEX &operator+=(double d)
}
```

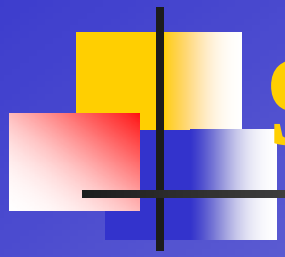
9.4 强制类型转换

- 为了减少运算符函数的重载个数，可以定义只有一个参数的构造函数。T类单参数的T::T(A)、T::T(const A)、T::T(const A&)相当于A类到T类的强制转换函数。
- 【例9.13】定义复数类型，实现加法运算。

```
class COMPLEX{
    double r, v;
public:
    COMPLEX(double r1){ r=r1; v=0; }
    COMPLEX(double r1, double v1){ r=r1; v=v1; }
    COMPLEX operator+(const COMPLEX &c)const
    { return COMPLEX(r+c.r, v+c.v); }
```

9.4 强制类型转换

```
    COMPLEX &operator+=(const COMPLEX &c);
};
COMPLEX &COMPLEX::operator+=(const COMPLEX &c){
    r+=c.r;  v+=c.v;    return *this;
}
void main(void){
    COMPLEX c1(2, 4), c2(3, 4);
    c1=c1+c2;  //调用COMPLEX operator+(const COMPLEX &)
    c1=c1+'2';  //调用COMPLEX operator+(const COMPLEX &)
    c1+=c2;     //调用COMPLEX &operator+=(const COMPLEX &)
    c1+=2.5;    //先将2.5转为常量对象COMPLEX(2.5)，再作+=运算
}               //调用COMPLEX &operator+=(const COMPLEX &)
```



9.4 强制类型转换

- 有时需要将类A转换为类B，或者将类A转换为简单类型，为此必须重载强制类型转换。由于转换后的类型就是函数的返回类型，所以强制类型转换函数不需要定义返回类型。
- 重载强制类型转换使数据在必要时能自动转换类型，从而使程序代码显得更加简洁完美。
- 按照C++的约定，类型转换的结果做为右值，故最好不要将类型转换函数的返回值定义为左值 (如非只读类型的引用)

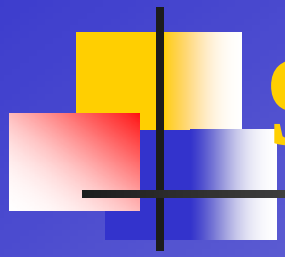
9.4 强制类型转换

【例9.14】本例介绍了强制类型转换函数的定义及用法。

```
struct A{
    int  i;
    A(int v) { i=v; }
    operator int( ) const{ return  i; }  //返回右值
    //const表示强制转换不改变当前对象的值
};
struct B{
    int  i, j;
    B(int x, int y) { i=x; j=y; }
    operator int( ) const{ return  i+j; } //返回右值
    //const表示强制转换不改变当前对象的值
```

9.4 强制类型转换

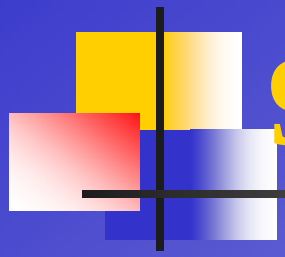
```
    operator A( ) const{ return A(i+j); } //返回右值
    //const表示强制转换不改变当前对象的值
};
void  main(void){
    A  a(5);
    B  b(7, 9);
    B  c(3, a);        //调用A::operator int( )转换a，等价于B  c(3, 5)
    int  i=1+a;        //调用A::operator int( )转换a，i=6
    i=b+3;             //调用B::operator int( )转换b，i=19
    i=a=b;             //调用B::operator A( )和A::operator int( )，i=16
    B d(i, b);         //调用B::operator int( )转换b
}
```



9.5 重载new和delete

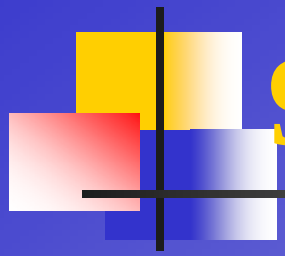
- 运算符函数new和delete定义在头文件new.h中，其函数原型为：
  - extern void * operator new(unsigned bytes);
  - extern void operator delete(void *ptr);

- 函数new的参数就是要分配的内存的字节数。在使用运算符new分配内存时，是用类型而不是用内存大小作为new的参数，编译程序会根据类型计算内存大小并调用上述new函数。



9.5 重载new和delete

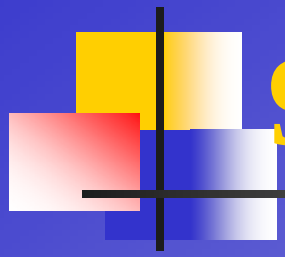
- 可以重载new和delete函数来实现内存的自我管理：
  - 重载内存管理函数既可以在整个程序范围内，也可以只针对某个类。后者更为常见，因为不同的类具有不同的结构和不同的内存需求，在整个程序范围内重载并不能保证每个类的内存分配都能达到最优。
- new和delete不能高效管理特别小的对象的内存：
  - 对于特别小的对象，由于内存管理产生的额外内存开销可能会超过对象所需的内存；太多的小块内存也会严重降低内存的分配和释放速度。这些问题要求重载运算符函数new和delete。



9.5 重载new和delete

【例9.15】重载new和delete以便更好地为POINT分配空间。

```
class  POINT{
    int x, y;
    static struct Block{
        int xy[32][2];
        unsigned long used;
        Block *next;
    } *list;
public:
    static void *operator new(unsigned);
    static void operator delete(void *);
};
```

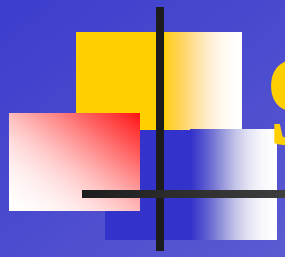


9.5 重载new和delete

```
POINT::Block *POINT::list=0;
void *POINT::operator new(unsigned bytes){
    Block *r;
    for(r=list; r!=0; r=r–>next) {
        if(r–>used==0xFFFFFFFFL) continue;
        for(int i=0; i<32; i++)
            if(!i&&!(r–>used&1)||!(r–>used>>i&1)) {
                r–>used+=1L<<i;
                return r–>xy[i];
            }
    }
    r=new Block;    r–>next=list;    list=r;
    r–>used=1;
    return r–>xy[0];
}
```

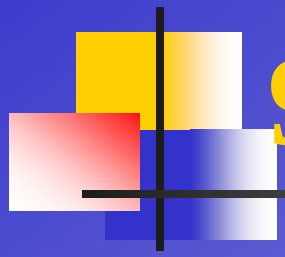
9.5 重载new和delete

```
void POINT::operator delete(void *p){
    Block *r, *q;
    for(r=list; r!=0; q=r, r=r–>next)
        for(int i=0; i<32; i++)
            if(((r–>used>>i)&1)&&(r–>xy[i]==p)) {
                r–>used-=1L<<i;
                if(r–>used!=0) return;
                if(r==list)
                { list=q–>next;  delete q;   return; }
                q–>next=r–>next; delete r;   return;
            }
}
void main(void){
    POINT *p=(POINT *)POINT::operator new(sizeof(POINT));
    POINT::delete(p);     //不能忘记释放p指向的内存空间
}
```



9.6 表运算实例

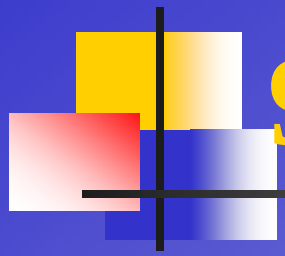
- 符号表由一组符号名及其属性值组成，常见的运算包括查表、插入、删除等运算。
- 通过定义这些运算的运算符函数，程序将变得更加清晰可读。
- 查表、插入以及删除运算可以分别定义成[]、()和–运算。重载( )时可以定义任意个参数，也可以省略参数。



9.6 表运算实例

【例9.16】定义符号表及其查表、插入及删除运算。

```
#include <string.h>
#include <iostream.h>
class SYMTAB;
struct SYMBOL{
//由于构造和析构函数是私有的，故本类不能定义全局对象
    char *name;
    int      value;
    SYMBOL *next;
    friend SYMTAB;
private:
    SYMBOL(char *, int, SYMBOL *);
    ~SYMBOL( ) { delete name; }
};
```



9.6 表运算实例

```
SYMBOL::SYMBOL(char *s, int v, SYMBOL *n){
    strcpy(name=new char[strlen(s)+1], s);
    value=v;    next=n;
}
class SYMTAB{
    SYMBOL *head;
public:
    SYMTAB( ) { head=0; };
    const SYMBOL *operator( )(char *, int);
    const SYMBOL *operator[ ](char *);
    int operator–(char *);
    ~SYMTAB( );
    };
SYMTAB::~SYMTAB( ){
    SYMBOL *p=head;
    while(p) { head=p –>next;   delete p;   p=head;  }
}
```

9.6 表运算实例

```
SYMTAB::~SYMTAB( ){
    SYMBOL *p=head;
    while(p) { head=p –>next;   delete p;   p=head;  }
}
const SYMBOL *SYMTAB::operator( )(char *s, int v){ //插入运算
    return head=new SYMBOL(s, v, head);
}
const SYMBOL *SYMTAB::operator[ ](char *s){ //查表运算
    for(SYMBOL *h=head; h!=0; h=h–>next)
        if(strcmp(s, h–>name)==0)  break;
    return h;
}
int SYMTAB::operator–(char *s){
    SYMBOL *p, *q;
    for(p=q=head; p!=0; q=p, p=p–>next)
        if(strcmp(s, p–>name)==0)  break;
```

9.6 表运算实例

```
    if(p==0) return 0;
    if(q==head) head=p–>next; else q–>next=p–>next;
    delete p;    return 1;
}
void main(void){
   SYMTAB tab;
   const SYMBOL *s;
   s=tab(“a”, 1);              //插入元素，包括this实际有三个参数
   s=tab("b", 2);              //插入元素，包括this实际有三个参数
   tab–"b";                    //删除元素
   if(s=tab["a"])              //查表运算
   cout<<"Symbol a="<<s–>value;
   if(!tab["b"])               //查表运算
   cout<<"\nSymbol b not found";
}
```