華中科技大學

第11章 运算符重载

许向阳

xuxy@hust.edu.cn

大纲

華中科技大學

11.1 运算符概述

11.2 运算符参数

11.3 赋值与调用

11.4 强制类型转换

11.5 重载new和delete

11.6 运算符重载实例

难点

运算符重载函数的正确写法

强制类型转换

11.1 运算符概述

- 什么是运算符重载?

  函数重载
- 运算符有哪些?
- 为什么要重载?
- 如何重载?

11.1 运算符概述

- 纯单目运算符，只能有一个操作数

  !、~、sizeof、new、delete 、++、--等

- 纯双目运算符，只能有两个操作数

  [ ]、->、%、=

- 三目运算符，有三个操作数，如“? :”

- 既是单目又是双目的运算符

  +、-、&、*

- 多目运算符，如函数参数表 “( )”。

11.1 运算符概述

11.1.1 结果为左值的运算符

- 左值运算符是运算结果为左值的运算符

  其表达式可出现在等号左边

  如前置++、--以及赋值运算=、+=、*=和&=等。

- 右值运算符是运算结果为右值的运算符

  如+、-、>>、 %、后置++、--等。

- 某些运算符要求第一个操作数为左值

  如 ++、-- 、=、+=、&=等。

11.1 运算符概述


```
#include <stdio.h>
void main(int argc, char *argv[ ])
{    int x=0;
     ++x;              //++x的结果为左值（可出现在等号左边）
     ++  ++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
     printf("%d %d", x, x++); //( )可看作任意目运算符
} //(x--)++是错的，x--的结果为右值，而++要求一个左值
```

11.1.1 结果为左值的运算符

```
class MAT {
    int*  e;        //指向所有整型矩阵元素的指针
    int r, c;
public:
    MAT(int r, int c);  //矩阵定义
    MAT(const MAT& a);  //深拷贝构造
    MAT(MAT&& a) ;
    MAT& operator=(const MAT& a); //深拷贝赋值运算
    MAT& operator=(MAT&& a) ;      //移动赋值运算
    MAT& operator+=(const MAT& a);// “+=” 运算
    MAT& operator-=(const MAT& a);// “-=” 运算
    MAT& operator*=(const MAT& a);
};
```

11.1 运算符概述

11.1.1 结果为左值的运算符

赋值函数的正确写法：

ClassName & operator =(const ClassName &obj);

MAT a(3,4); MAT b(3,4) ; MAT c(3,4);

a= b = c; // a.operator= (b.operator =(c));

(a=b) = c; // (a.operator= (b)).operator =(c) ;

有问题的一些写法:

void operator =(ClassName obj);

void operator =(ClassName &obj);

void operator =(const ClassName &obj);

ClassName operator =(const ClassName &obj);

Q: 各自的执行过程是什么？

11.1 运算符概述

总结

赋值函数的恰当写法：

ClassName & operator =(const ClassName &obj);

- 参数有 & 及 const 的原因

  引用参数 比 对象参数有更高的效率；

  = 之右可以是有名对象，也可以是临时对象，const 通配

- 返回有 引用

  连续赋值

在函数返回引用时，增加const 修饰，结果如何？

（a=b）=c; 编译有错误。

11.1 运算符概述

左值运算符 += 函数的正确写法：

```
ClassName & operator +=(const ClassName  &obj);
```

11.1.2 运算符重载的分类

- 有些运算符不能重载
- 重载为普通函数
- 重载为实例函数成员
- 重载为静态函数成员

11.1.2 运算符重载的分类

- C++预定义了简单类型的运算符重载

  如3+5、3.2+5.3分别表示整数和浮点加法。

- 运算符重载必须针对类的对象

  重载时至少有一个参数代表对象(类型如A、const A&)。

- C++用“operator 运算符”进行运算符重载。

- 对于运算符实例函数成员，隐含参数this代表第一个操作数对象。

11.1 运算符概述

- sizeof　.　.*　::　?　:　不能重载
- 运算符（+　-　*　/、+=、*= 等）

  可以重载为普通函数、实例函数成员

  不能重载为静态函数成员
- =、->、( )、[ ]

  可以重载为类中的实例函数成员

  不能重载为普通函数、静态函数成员
- new、delete

  可以重载为普通函数、静态函数成员

  不能重载为类中的实例成员函数

11.1 运算符概述

普通函数	实例函数成员	静态函数成员
+、-、*、/ 等 +=、*= 、/=等 new、delete	+、-、*、/ 等 =、-> () [] += 、 *= 、/=等	new delete

双目算术运算：+、-、*、/、%

关系运算：==、!=、<、>、<=、>=

逻辑运算：||、&&、！

单目运算：+ /正、-/负、*/指针、&/取地址

自增自减：++、--

位运　算：|、&、~、^、<<、>>

赋值运算：=、+=、-=、……

空间申请与释放：new,delete、new[], delete []

其他运算：　()(函数调用)、[](下标)、-> 、，(逗号)

11.1.3 成员与非成员重载

```
MAT operator+(const MAT  & a, const MAT  &b)
{  MAT    temp(a);
   ……
   return temp;
}

MAT MAT::operator+(const MAT  & a)
{  MAT    temp(a);
   ……
   return temp;
}
MAT a(3,4), b(3,4),c(3,4);
c=a + b;     // 优先使用实例函数成员
c = a. operator + (b);   // 使用实例函数成员
c = operator +(a,b);    // 使用普通重载函数
```

重载 + 为普通函数
VS 实例函数成员

11.1 运算符概述

双目 加法 函数：

ClassName operator +(const ClassName &obj);

- 参数有 & 及 const 的原因

  引用参数 比 对象参数有更高的效率；

  = 之右可以是有名对象，也可以是临时对象，const 通配

- 返回是对象，而不是引用

  相加的结果一般放在局部对象中，不应返回局部对象引用。若在函数中申请空间，又难于控制空间的释放。

- 返回的类型前不加 const

  返回结果是一个临时对象 temp，若 之后 有 a=temp;

  等同 a = const temp; 此时会执行以对象有址引用为参数的赋值函数。

11.1 运算符概述

方括号 [] 的重载

```
class STRING {
private:
     char *s;
public:
    STRING(const char *s)  {
          s=new char[strlen(str)+1];
          strcpy(s,str);
    }
    char operator [](int i) {
          return *(s+i);
     }
};
```

```
STRING  s1("S1 hello");
char  t;
t=s1[5];
cout<<s1[0]<<s1[1]<<endl;
```

s1[5] 相等于
s1.opearator [] (5)

方括号 [] 的重载

```
class  MAT {
public:
      int  *e;
      int   r,   c;
            //取矩阵 i行的第一个元素地址
      int* const operator[ ](int i) {
          return  e+i*c;
      }
};
```

11.1 运算符概述

方括号 [] 的重载

```
MAT  a（3， 4）;
 a[0][1]=1;
```

先执行 a[0], 得到了 int * 的地址， 相当于 int *p;
p=a[0]; 之后 p[1] 则是访问第 0行的第1个元素了。

Q:两个[ ][ ]的处理方式为何不同？

它的解析顺序是 (a[0])[1], 在执行 a[0] 时， 隐含的参数类型是 a 的类型，即 MAT 类型，此时使用 类 的运算符函数[], 但其返回类型是 int*， 之后的[] , 相当于 p[]，类型不再是 MAT， 故不会调用l类中的 运算符 [] 。

11.1 运算符概述

圆括号 （） 的重载

```
class STRING {
private:
    char *s;
public:
   STRING(const char *s)  {
        s=new char[strlen(str)+1];
        strcpy(s,str);
   }
   int operator ()(int i) {
         s[0]=s[0]+i;
         return i+10;
    }
};
```

```
STRING  s1("S1 hello");
int  x;
t=s1(1);
cout<<s1[0]<<s1[1]<<endl;
```

s1(1) 相等于
s1.opearator () (1)

定义对象时构造函数的（）不受影响；
紧跟在对象之后的（）重载

11.1 运算符概述

- 若运算符为左值运算符，则重载后运算符函数返回非只读引用类型(左值)。当运算符要求第一个参数为左值时，不能使用const说明第一个参数指向的对象(含this)，例如++、--、=、+=等的第一个参数；
- 重载运算符函数可以声明为类的友元；重载的普通运算符成员函数也可定义为虚函数；
- 重载运算符函数一般不能缺省参数，只有任意目的运算符( )省略参数才有意义；
- 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。

11.1 运算符概述


```
class A{
    int x, y;
public:
    A(int x, int y) { A::x=x; A::y=y; }
    A &operator=(const A&m)//返回类型为左值，即返回值可再被修改赋值
     { x=m.x; y=m.y; return *this;}; //左值引用当前对象，赋值后还可赋值
    friend A operator-(const A&); //返回右值，参数也为右值，不可修改
    friend A operator+(const A&, const A&); //const表示不能修改两个加
    数
} a(2,3), b(4,5), c(1, 9);

a = b  = c;        //  (b=c)  → t    a=t;

(a=b) = c;
```

11.1 运算符概述


```
class A{
    int x, y;
public:
    A(int x, int y) { A::x=x; A::y=y; }
    A &operator=(const A&m)//返回类型为左值，即返回值可再被修改赋值
     { x=m.x; y=m.y; return *this;}; //左值引用当前对象，赋值后还可赋值
    friend A operator-(const A&); //返回右值，参数也为右值，不可修改
    friend A operator+(const A&, const A&); //const表示不能修改两个加
    数
} a(2,3), b(4,5), c(1, 9);
A operator -(const A&a){ return A(-a.x, -a.y); }//普通函数返回右值
A operator +(const A&x, const A&y){//返回右值,(被)加数、结果均不能修
    改
    return A(x.x+y.x,  x.y+y.y); //A(x.x+y.x,  x.y+y.y)为类A的常量
}
void main(void){  (c=a+b)=b+b /*c=a+b, c=b+b*/;  c= -b;  }
```

11.2 运算符参数

- 重载函数种类不同，参数表列出的参数个数也不同。
  - 重载为普通函数：参数个数=运算符目数
  - 重载为普通成员：参数个数=运算符目数－1 (即this指针)
  - 重载为静态成员：参数个数 = 运算符目数(没有this指针)
- 有的运算符既可以是单目，也可是双目，如*, +, -等。
- 特殊运算符不满足上述关系：->双目重载为单目，前置++和--重载为单目，后置++和--重载为双目、函数( )可重载为任意目。
- ( )表示强制类型转换时为单参数；

  表示函数时可为任意个参数。

11.2 运算符参数


```
#include <string.h>
class SYMTAB;
struct SYMBOL{
    char *name;  int  value;  SYMBOL *next;   friend SYMTAB;
private:
    SYMBOL(char*s,int v, SYMBOL *n){/*...*/}; ~SYMBOL( )
    { /*...*/ }
} *s;
class SYMTAB{
    SYMBOL *head;
public:
    SYMTAB( ) { head=0; };    ~SYMTAB( ){/*...*/  }
    SYMBOL *operator( )(char *s, int v, int w){ /*...*/};
} tab;
void main(void){ s=tab("a", 1, 2);} //包括this(指向tab)实际有四个
    参数
```

11.2 运算符参数

- 运算符++和--都会改变当前对象的值，重载时最好将参数定义为非只读引用类型(左值)，左值形参在函数返回时能使实参带出执行结果。前置运算是先运算再取值，后置运算是先取值再运算。
- 后置运算应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(已包含一个不用const修饰的this参数)；
  - 如果重载为普通函数(C函数)，则最好声明非const引用类型和int类型的两个参数(无this参数)。
- 前置运算应重载为返回左值的单目运算符函数：
  - 前置运算结果应为左值，其返回类型应该定义为非只读类型的引用类型；左值运算结果可继续++或--运算。
  - 如果重载为普通函数(C函数)，则最好声明非const引用类型一个参数(无this参数)。

11.2 运算符参数


```
class A{
    int  a;
    friend A &operator--(A&x){x.a--; return x; }//自动内联，返回左值
    friend A operator--(A&, int); //后置运算，返回右值
public:
    A &operator++( ){ a++;   return *this; }//单目，前置运算
    A operator++(int){ return  A(a++); }//双目，后置运算
    A(int x) { a=x; }
};//A m(3); (--m)-- ;
    可以，因为--m左值，其后--要求左值操作数
A operator--(A&x, int){
                              // x左值引用，实参被修改
    return  A(x.a--);      // 先取x.a返回A(x.a)右值，再x.a--
} //A m(3); (m--)-- ;
  // 不可，因为m--右值，其后--要求左值操作数
```

11.2 运算符参数

重载 ++，时钟，时间增加1秒

```
#include <iostream>
using namespace std;
class Clock {
private:
    int  hour;
    int  minute;
    int  second;
public:
    Clock() {
hour=minute=second=0;
    }
    Clock(int h, int m, int s) {
            hour = h;
            minute = m;
            second = s;
    }
    void display() {
            cout << hour << " : " << minute <<
    " : " << second << endl;
    }
     Clock & operator++(); // 前置运算++
};
```

11.2 运算符参数


```
Clock & Clock::operator++()
{
    second++;
    if (second == 60) {
        second = 0;
        minute++;
        if (minute == 60) {
            minute = 0;
            hour++;
            if (hour == 24)  hour = 0;
        }
    }
    return *this;
}
```

11.2 运算符参数

```
int main()
{
    Clock t(10, 59, 50);
    for (int i = 0;i < 100;i++) {
        ++t;
        t.display();
    }
    return 0;
}
```

从例子中看，operator++的返回值未使用，
能否用void operator++(); 代替 Clock& operator++();？？

单从例子中看，是可以用void operator++(); 来代替。
但是 对于 t = ++t； 或者 another = ++t; 就会报错。

11.2 运算符参数


```
    int xx = 5;
    int yy;
    yy = xx++;     //  yy=5;   xx=6;
          // 重载后置++
  Clock Clock::operator++(int )
{
    Clock temp(*this);
    second++;
    if (second == 60) {
          second = 0;
          minute++;
           ……
    }
    return temp;
}
```

```
Clock t1(10, 59, 50);
Clock t2;
t2 = t1++;
// 执行后
// t1 = (10, 59, 51)
// t2 = (10, 59, 50)
```

11.2 运算符参数


```
//重载双目->，使其只有一个参数(单目)，返回指针类型
struct A{ int a; A(int x) { a=x; } };
class  B{
    A  x;
public:
    A *operator ->( ){ return &x; }; //只有一个参数this，故重载为单目
    B(int v):x(v) { }
}b(5);
void  main(void){
    int i=b->a;  //等价于下一条语句，i=b.x.a=5
    i=b.operator ->( )->a;  //i=b.x.a=5
    i =( b.operator ->( )) ->a;
    i=*(b.operator->( )).a; //i = b.operator ->( )->a
}
```

11.3 赋值与调用

区别 赋值与定义对象


```
Array   a=b;
    //  定义对象 a, 即为 a 分配空间
    //  调用以对象为参数的构造函数，初始化对象 a;

a=b;
    //  a 的空间已存在
    //  要考虑释放 a 已有的体外空间
    //  要考虑 为 a 中的指针分配新的体外空间
    //   将 b 中的数据 深拷贝到 a 中
```

11.3 赋值与调用

区别 深拷贝赋值与移动赋值

a=b;　　// b 是一个命名的对象

　　// 在赋值后，b 应该保持不变

a=临时对象;

例：　a = f(...) ;　　f(...) 返回对象

例：　a = u + v;　　u+v的结果是一个临时对象

// 临时对象在赋值语句执行后会被析构

// 考虑 a 中的指针直接指向临时对象中指针指向的体外空间，然后将临时对象中的指针置空

11.3 赋值与调用

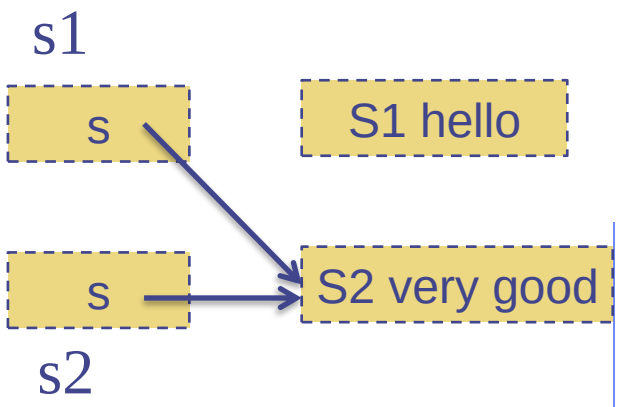
- 运算符 =、+=、*=、&=、|= 等是左值运算法
- 编译程序为每个类都提供了缺省赋值运算符函数
- 缺省赋值运算实现数据成员的复制或浅拷贝赋值，
  对指针类型的数据成员，不复制指针所指存储单元的内容。
  若类不包含指针，浅拷贝赋值不存在问题。

11.3 赋值与调用

```
class STRING {
private:  char *s;
public: STRING(char *s);
        ~STRING();
};

STRING::STRING(const char *str) {
        s=new char[strlen(str)+1];
        strcpy_s(s,str);
        cout<<"Construct :"<<str<<endl;
}
STRING::~STRING() {
        cout<<"Delete : "<<s<<endl;
        if (s) { delete s; s=NULL;}
}
```

```
void main()
{
  STRING  s1("S1 hello");
  STRING  s2("S2 very good");
  s1=s2;
  mov   eax,dword ptr [ebp-20h]
  mov   dword ptr [ebp-14h],eax
}
```



浅拷贝
析构s1时出错

11.3 赋值与调用

实现独立对象的拷贝方案

friend void Copy_S (STRING &dest, STRING sour);

```
void Copy_s(STRING &dest, STRING sour)
{
    dest.s= new char[strlen(sour.s)+1];
    strcpy_s(dest.s, sour.s);
}
void main()
{
   STRING  s1("S1 hello");
   STRING  s2("S2 very good");
   Copy_s(s1,s2);      // s1=s2;
}
```

可行吗？

分析参数 **sour**的传递方式

11.3 赋值与调用

friend void Copy_S (STRING &dest, STRING sour);

```
void main()
{
   STRING  s1("S1 hello");
   STRING  s2("S2 very good");
   Copy_s(s1,s2);
mov  eax,dword ptr [ebp-20h]
push eax
lea    ecx,[ebp-14h]
push  ecx
call   Copy_s (2C126Ch)
}
```

浅拷贝
析构s2时出错

在退出Copy_s 时，会析构参数对象sour，导致
即s2中指针指向的单元被释放

11.3 赋值与调用

friend void Copy_S (STRING &dest, STRING &sour);

```
void main()
{
   STRING  s1("S1 hello");
   STRING  s2("S2 very good");
   Copy_s(s1,s2);
   lea     eax,[ebp-20h]
   push    eax
   lea     ecx,[ebp-14h]
   push    ecx
   call    Copy_s (981271h)
}
```

无错误，参数是对象的地址；
无析构

11.3 赋值与调用

void CopyString (STRING &sour); 成员函数

```
void CopyString(STRING &sour)
{
    if (s)  delete s;
    s= new char[strlen(sour.s)+1];
   strcpy_s(s, sour.s);
}
void main()
{
  STRING  s1("S1 hello");
  STRING  s2("S2 very good");
  s1.CopyString(s2);
}
```

重载赋值运算，实现深度拷贝

```
void operator = (STRING &sour)
{
   if (s)  delete s;
   s= new char[strlen(sour.s)+1];
   strcpy_s(s, sour.s);
}

s1=s2;
```

11.3 赋值与调用

void CopyString (STRING &sour); 成员函数

```
STRING & operator = (STRING &sour)
{
    if (s)  delete s;
    s= new char[strlen(sour.s)+1];
    strcpy(s,sour.s);
    return *this;
}
```

定义恰当的返回类型

支持： s1=s2;

支持： s1=s2=s3;

等同于 s1=(s2=s3);

不同于 (s1=s2)=s3;

重载赋值运算，实现深度拷贝

```
void operator = (STRING &sour)
{
    if (s)  delete s;
    s= new char[strlen(sour.s)+1];
    strcpy(s,sour.s);
}
```

支持：s1=s2;

不支持：s1=s2=s3;

11.3 赋值与调用

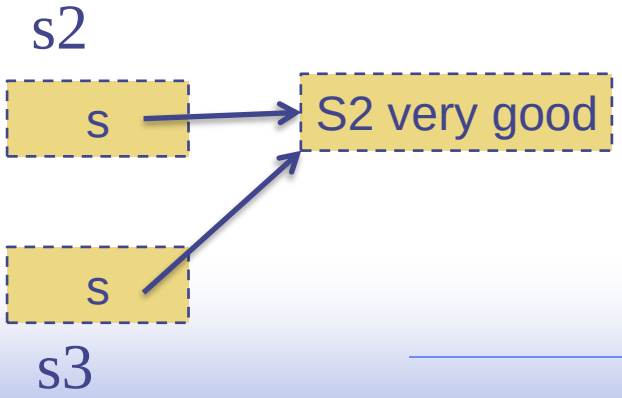
以对象为参数的构造函数

```
void main()
{
   STRING  s1("S1 hello");
   STRING  s2("S2 very good");
   STRING  s3 (s2);
   mov   eax,dword ptr [ebp-14h]
   mov   dword ptr [ebp-2Ch],eax
}
```

有缺省的以对象为参数的构造函数。

实现方式是浅拷贝

程序运行结束时出错



11.3 赋值与调用

```
STRING(STRING  &os)  {
    s=new char[strlen(os.s)+1];
    strcpy(s,os.s);
}
void main()
{
   STRING  s1("S1 hello");
   STRING  s2("S2 very good");
   STRING  s3 (s1);
  lea     eax,[ebp-14h]
  push  eax
  lea     ecx,[ebp-2Ch]
  call    STRING::STRING
}
```

以对象为参数的构造函数。

复制构造函数

参数必须是对象的引用

```
STRING(STRING os)  {
    s=new char[strlen(os.s)+1];
    strcpy(s,os.s);
}
```

非法的复制构造函数: 第一个参数不应是“STRING”

11.3 赋值与调用

```
STRING::STRING(STRING &os)  {
    s=new char[strlen(os.s)+1];
    strcpy(s,os.s);
}
friend void Copy_s(STRING &dest,
                   STRING sour);
void main()
{
   STRING  s1("S1 hello");
   STRING  s2("S2 very good");
   Copy_s(s1,s2);
}
```

以对象为参数的构造函数。

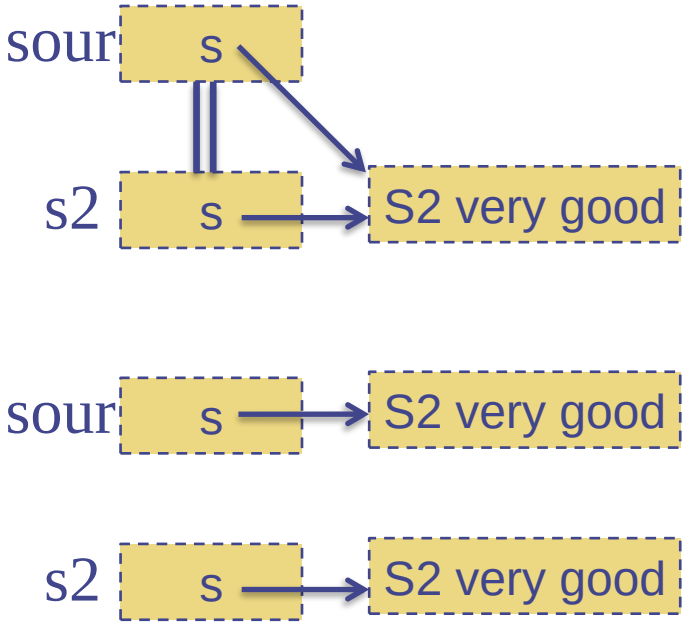
比较有/无复制对象构造函数时，对象参数传递的不同

对于 对象参数**sour**

- 无对象复制构造函数时
  - 使用缺省的构造函数；
  - 按字节内容，简单复制
- 有对象复制构造函数时
  - 自动使用该构造函数
  - 完整的深度复制对象

11.3 赋值与调用

friend void Copy_s(STRING &dest, STRING sour);
Copy_s(s1,s2);



对于 对象参数**sour**

- 无对象复制构造函数时
  - 使用缺省的构造函数；
  - 按字节内容，简单复制
- 有对象复制构造函数时
  - 自动使用该构造函数
  - 完整的深度复制对象

11.3 赋值与调用

STRING s3(s1);

```
00E24712  lea     eax,[ebp-14h]
00E24715  push    eax
00E24716  lea     ecx,[ebp-2Ch]
00E24719  call    STRING::STRING (0E2128Fh)
00E2471E  mov     byte ptr [ebp-4],2
```

STRING s4=s1;　写在定义处的 = 与赋值语句不同

调用的是构造函数

```
00E24722  lea     eax,[ebp-14h]
00E24725  push    eax
00E24726  lea     ecx,[ebp-38h]
00E24729  call    STRING::STRING (0E2128Fh)
00E2472E  mov     byte ptr [ebp-4],3
```

11.3 赋值与调用

赋值运算符重载和复制对象构造函数的比较

- 两者的功能相似
- 实现方式相似
- 赋值运算符显式调用，有 =
- 复制对象构造函数自动调用

  以一个对象为参数，生成另一个对象
- 两个都有默认的函数，都是浅拷贝

11.3 赋值与调用

- 编译程序为每个类提供了缺省赋值运算符函数

  对类A而言，其成员函数原型为A&operator=(const A&)。

- 缺省赋值运算实现数据成员的复制或浅拷贝赋值，

  对指针类型的数据成员，不复制指针所指存储单元的内容。

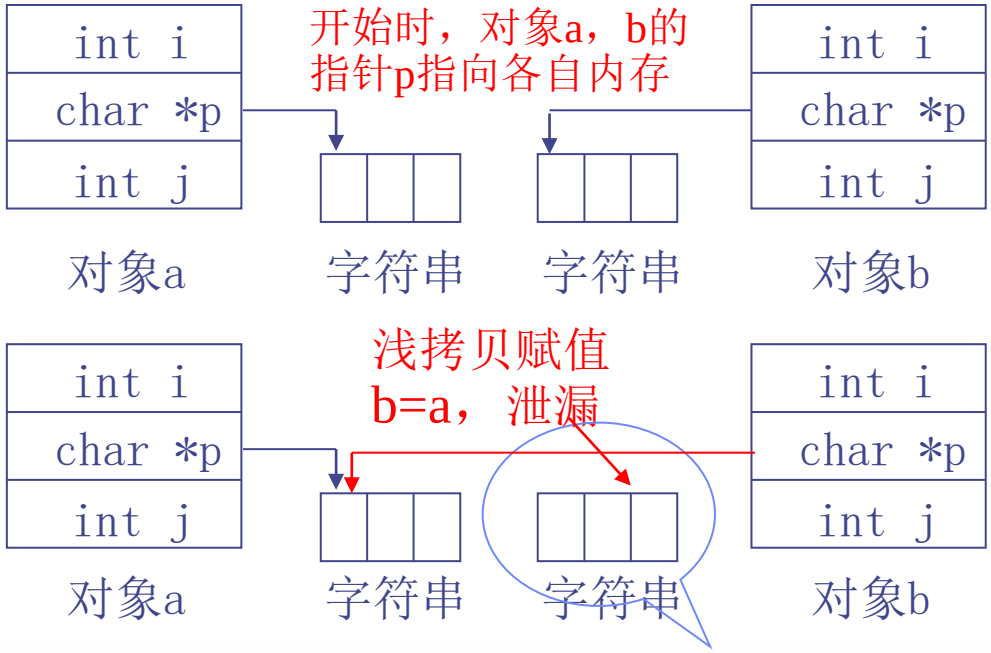
  若类不包含指针，浅拷贝赋值不存在问题。

- 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(不管是否取代型定义)。

- 如果函数参数要值参传递一个对象，当实参传值给形参时，若类A没有定义A(const A&)形式的构造函数，则值参传递也通过浅拷贝赋值实现。

11.3 赋值与调用

当类包含指针时，浅拷贝赋值可造成内存泄漏，并可导致页面保护错误或变量产生副作用。



11.3 赋值与调用


```
#include <string.h>
class STRING{
    char *s;
public:
    virtual char &operator[ ](int x){ return s[x]; }
    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); }
    //返回左值用STRING&，而STRING和const  STRING&都是返回右值
    virtual STRING operator+(const STRING &)const;//返回右值，因此，
    //加数、被加数及和(即函数返回值)都不能被修改或赋值
    virtual STRING&operator=(const STRING &); //返回左值可连续赋值
    virtual STRING&operator+=(const STRING&s){return *this=*this+s;}
    virtual ~STRING( ) { if(s){ delete [ ]s; s=0; }}
} s1("S1"),  s2="S2", s3("S3");//s2="S2"等价于是s2("S2")
```

11.3 赋值与调用


```
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)); //strcpy、strcat返回t
    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){
    (s1=s1+s2)=s2; //重载“=”返回左值，可连续赋值否则不可
                   //等价于s1=s1+s2; s1=s2;s1被连续赋值
    s1+=s3;
    s3[0]='T';// s3[0]=调用char &operator[ ](int x)返回左值
}
```

11.3 赋值与调用

对于类T，防止内存泄露要注意以下几点：

- 不要随便使用exit和abort退出程序；
- 定义T(const T &)等形式的深拷贝构造函数；
- 定义virtual T &operator=(const T &)等形式的深拷贝赋值运算符函数；
- 定义virtual ~T( )形式的虚析构函数；
- 在定义引用T &p=*new T( )后，使用delete &p析构并释放对象占用的内存；
- 在定义指针T *p=new T( )后，使用delete p析构并释放对象占用的内存。

11.4 强制类型转换

- C++是强类型的语言，运算时要求类型相容或匹配。
- 定义合适的类型转换函数，可完成操作数的类型转换；

```
int  x = 3;      double y=4.6;
x = y;  // 警告：从double 转换到 int，可能丢失数据
x = (int)y;
x = int(y);  // 等价写法
   cvttsd2si   eax, mmword ptr [y]
   mov         dword ptr [x], eax
```

讨论：数据类型转换 与地址类型转换有何差别？

```
x=int(y);          // x=4
x =*(int *)&y;  // x =1717986918
```

11.4 强制类型转换

设有复数类 COMPLEX {double r, v; .....};

要求完成如下功能，如何实现？

复数 C3 = 复数 C1 + 复数 C2

复数 C3 = 复数C1 + 浮点数 r2

复数 C3 += 复数 C1

复数 C3 += 浮点数 r2

方法1：定义合适的构造函数，可以构造符合类型要求的对象，构造函数可以起到类型转换的作用。

方法2：定义类型转换函数，实现类型转换

11.4 强制类型转换

定义“复数+复数”、“复数+实数”、“复数+整数”、“复数-复数”、“复数-实数”、“复数-整数”几种运算（还有复数同实数乘除运算等等，实在太多）：

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

11.4 强制类型转换

- 单参数的构造函数具备类型转换作用，必要时能自动将参数类型的值转换为要构造的类型。
- C++会自动将 int 转为 double

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

COMPLEX m(3);

m + 2 转换为 m + 2.0; 再转换为 m + COMPLEX(2.0)

11.4 强制类型转换

- 表面上是多参数，但是有缺省参数时，等同单参数的构造函数具备类型转换作用，必要时能自动将参数类型的值转换为要构造的类型。
- C++会自动将 int 转为 double

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

COMPLEX m(3);

m + 2 转换为 m + 2.0; 再转换为 m + COMPLEX(2.0)

11.4 强制类型转换

- 单参数的构造函数相当于类型转换函数

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。

COMPLEX(double r1); // 由浮点数类型转换为COMPLEX类型

11.4 强制类型转换

- 用operator定义强制类型转换函数。 operator 类型(...)

  由于转换后的类型就是函数的返回类型，

  强制类型转换函数不需要定义返回类型。

- 不能同时定义T::T(A) 和T::T(const A&)

  表面上看，两者是不同的。

  但若有语句 T(A)，编译报错：对重载的调用不明确。

- 按照C++约定，类型转换的结果通常为右值，故最好不要将类型转换函数的返回值定义为左值，也不应该修改当前被转换的对象 (参数表后用const说明this)。

11.4 强制类型转换


```
struct A{
      int  i;          A(int v) { i=v; }
      virtual operator int( ) const{ return  i; } //类型转换返回右值
}a(5);
struct B{
      int  i, j;       B(int x, int y) { i=x; j=y; }
      operator int( ) const{ return  i+j; }      //类型转换返回右值
      operator A( ) const{ return A(i+j); }      //类型转换返回右值
}b(7, 9),  c(a, b);

void  main(void){
   int  i=1+(int)a; //强制转换，调用A::operator int( )转换a，i=6
   i = a;           //  i = a. operator int( ) ;
   i=b+3;           //自动转换，调用B::operator int( )转换b，i=19
   i=a=b;           //调用B::operator A( )和A::operator int( )，i=16
                    //  i = a = b.operator A();
}
```

11.5 重载new和delete

- 运算符函数new和delete定义在头文件new.h中

  extern void * operator new(unsigned bytes);

  extern void operator delete(void *ptr);

- 运算符new分配内存的大小

  类型表达式而不是值表达式作为实参

  new long[20]　　// sizeof(long)*20

- new和delete可重载为普通函数，

  也可重载为静态函数成员。

总结

- 运算符重载的语法

  与函数类比， 逆波兰的表达形式
- 重载运算符为普通函数
- 重载运算符为成员函数
- 对象复制构造函数

  浅拷贝 深拷贝

  对象作为参数时的构造函数
- 赋值运算符的重载
- 参数、返回结果的传递基理

运算符重载练习


```
class complex
{
 private:
        double real;
        double imag;
public:
     complex(double r, double i){
         real = r;
         imag = i;
     }
     complex() { }
};
```

```
complex c1(1.2 , 3.4);
complex c2(1.0, 2.0);
complex c3;   // 无参构造
```

练习：
完成复数c1和c2的加法；
结果在c3中。

程序要做哪些修改？

运算符重载练习

思路一：普通函数

```
complex   complex_add(complex x, complex y)
{   complex  z;   // 无参构造
    z.real = x.real + y.real;
    z.imag =x.imag+ y.imag;
    return  z;
}
```

```
c3 = complex_add(c1, c2);
```

问题：

- 该函数不是complex的成员函数，而是一个普通函数
- 该函数中不能访问complex的私有成员
- 定义为类的普通友元

```
friend  complex  complex_add(complex x, complex y)
```

运算符重载练习

执行过程分析：

c3 = complex_add(c1, c2);

(1) 浅复制对象c2 到参数 y中，拷贝所有数据成员

(2) 浅复制对象c1 到参数 x中，拷贝所有数据成员

(3) 临时返回对象的存放地址作为参数压栈

(4) CALL complex_add

构造局部对象 z;

执行加法；

对象z中的内容拷贝到 临时返回对象中

(5) 临时返回对象存放地址中的数据拷贝给 c3

运算符重载练习

优化 I：采用引用传递实参地址，减少对象构造

```
complex   complex_add(complex &x, complex &y)
{   complex  z;   // 无参构造
    z.real = x.real + y.real;
    z.imag =x.imag+ y.imag;
    return  z;
}

c3 = complex_add(c1, c2);
    //    c2 的地址压栈
    //    c1 的地址压栈
    //    其他相同
```

运算符重载练习

优化 II：增加 const约束，防止修改参数，
同时可适配临时对象作为参数

```
friend  complex  complex_add(const complex &x,
                              const complex &y);
```

```
complex   complex_add(const complex &x,
                      const complex &y)
{   complex  z;   // 无参构造
    z.real = x.real + y.real;
    z.imag =x.imag+ y.imag;
    return  z;
}
c3 = complex_add(c1, c2);
```

运算符重载练习

能否改成如下形式？

```
friend  complex  & complex_add(const complex &x,
                               const complex &y);

complex &  complex_add(const complex &x,
                       const complex &y)
{   complex  z;   // 无参构造
    z.real = x.real + y.real;
    z.imag =x.imag+ y.imag;
    return  z;                  // 有警告，返回局部变量的地址
}
c3 = complex_add(c1, c2);
  //  无法保证结果的正确性
  //  实现时，直接将 z的地址放在 eax中返回子
```

运算符重载练习


```
friend  complex  & complex_add(const complex &x,
                               const complex &y);
```

如何修改程序，使程序无警告？

```
complex &  complex_add(const complex &x,
                       const complex &y)
{   complex  *z = new complex;
    z->real = x.real + y.real;
    z->imag =x.imag+ y.imag;
    return  *z;
}
c3 = complex_add(c1, c2);
```

存在的问题：　　new申请的空间泄露，未释放

运算符重载练习

优化 III：返回结果也采用引用，减少了2次对象拷贝

```
friend  void complex_add(complex &z,
                         const complex &x,  const complex &y);

void   complex_add(complex &z,  const complex &x,
                                const complex &y)
{   z.real = x.real + y.real;
    z.imag =x.imag+ y.imag;
}

complex_add(c3, c1, c2);
```

运算符重载练习

思路一：普通函数，重载运算符+ 为普通函数

```
// 等同于 complex   complex_add(complex x, complex y)
complex  operator +(complex x,  complex y)
{   complex  z;
    z.real = x.real + y.real;
    z.imag =x.imag+ y.imag;
    return  z;
}
```

```
c3 = complex_add(c1,c2);
```

- 在complex中定义 “operator +”为类的普通友元

```
    friend complex operator +(complex x, complex y);
     c3=c1+c2;
// operator+ 等同于函数 complex_add
```

运算符重载练习

思路一：普通函数，重载运算符+为普通函数

```
// 等同于 complex   complex_add(const complex &x,
                                const  complex &y)

complex  operator +(const complex &x,  const complex &y)

c3 =c1+c2;
c3=complex_add(c1,c2);
```

运算符重载练习

思路二：成员函数

当前对象数据与另一个对象数据运算，结果在第3个对象中

```
complex complex :: complex_add(complex x)
{
    complex z;
    z.real=this->real+x.real;
    z.imag=this->imag+x.imag;
    return z;
}
```

c3 = c1.complex_add(c2);

- 进入函数时 有 x的构造，函数运行结束有x的析构。
- 有 z的构造和析构
- 返回结果的析构

运算符重载练习

思路二：重载为成员函数
重载运算符 +

```
// complex complex :: complex_add(complex x)

complex complex :: operator +(complex x)
{
    complex z;
    z.real=this->real+x.real;
    z.imag=this->imag+x.imag;
    return z;
}
```

c3 = c1.complex_add(c2); c3=c1+c2;

运算符重载练习

比较几种不同的写法

complex complex :: complex_add(complex x);

complex complex :: complex_add(complex &x);

complex complex :: complex_add(const complex &x);

complex & complex :: complex_add(complex &x);

complex complex :: operator +(complex x);

complex complex :: operator +(const complex &x); // 推荐写法

complex & complex :: operator +(complex &x);

运算符重载练习

complex complex_add(complex x);

```
c3 = c1.complex_add(c2);
sub         esp,10h
mov         eax,esp
mov       ecx,dword ptr [ebp-38h]  // c2的地址，即c2.real的地址
mov       dword ptr [eax],ecx        // [ebp-38h] ~ [ebp-30h]
mov       edx,dword ptr [ebp-34h]
mov       dword ptr [eax+4],edx    // 一个double型数据，分2次取
mov  ecx,dword ptr [ebp-30h]     // 第二个double型数据
mov  dword ptr [eax+8],ecx       //  c2.imag
mov  edx,dword ptr [ebp-2Ch]
mov  dword ptr [eax+0Ch],edx  // c2的两个数据成员放入了堆栈
lea     eax,[ebp-134h]  // 不是c3 的地址，而是一个临时对象的地址
push      eax
lea         ecx,[ebp-20h]       // c1的地址放在 ecx 中
call        complex::complex_add (12D127Bh)
```

运算符重载练习


```
call        complex::complex_add (12D127Bh)
mov         dword ptr [ebp-13Ch],eax  // eax 返回结果对象地址
mov         ecx,dword ptr [ebp-13Ch]
mov         edx,dword ptr [ecx]       // [ebp-50h]是c3的首地址
mov         dword ptr [ebp-50h],edx  // 拷贝2个double型数据
mov         eax,dword ptr [ecx+4]
mov         dword ptr [ebp-4Ch],eax
mov         edx,dword ptr [ecx+8]
mov         dword ptr [ebp-48h],edx
mov         eax,dword ptr [ecx+0Ch]
mov         dword ptr [ebp-44h],eax
lea         ecx,[ebp-134h]  // 析构临时对象
call        complex::~complex (12D1262h)
```