華中科技大學

面向对象程序设计

第四章 作用域及成员指针

许向阳

xuxy@hust.edu.cn

内容

華中科技大學

4.1 作用域

4.2 名字空间

4.3 成员指针

4.4 const、volatile和mutable

4.5 引用对象

4.1 作用域

标识符：变量名、函数名、参数名、类型名、常量名……

可以在什么范围内被访问?

- 全局变量
- 局部变量（包括参数变量/形参）
- 语句内的变量

```
for (int i=0; i<10; i++) {...}
```

- 静态变量（static 变量）：全局、局部、类的成员

4.1 作用域

面向过程(C传统)的作用域

从小到大可以分为四级：

① 作用于表达式内 (常量)

② 作用于函数内 (函数参数、局部变量、局部类型)

③ 作用于程序文件内 (static变量和函数)

④ 作用于整个程序 (全局变量、函数、类型)

整个程序 》一个文件内 》函数内 》表达式内

4.1 作用域

面向对象的作用域

从小到大可以分为五级：

① 作用于表达式内 (常量)

② 作用于函数成员内 (函数参数、局部变量、局部类型)

③ 作用于类或派生类内 (数据/函数/类型 成员)

④ 作用于基类内 (数据/函数/类型 成员)

⑤ 作用于虚基类内 (数据/函数/类型 成员)

虚基类 》基类 》类 /派生类 》成员函数 》表达式内

4.1 作用域

有同名符号时，该符号优先解释成什么？

全局变量 **int x;**

在一个函数中又有局部变量 **int x;**

该函数中 有 **x=5;** **x**是指的全局变量还是局部变量？

- 标识符作用域越小，被访问优先级就越高。

问：当函数成员的参数和数据成员同名时，优先访问谁？

4.1 作用域

有同名符号时，指定是用哪儿定义的符号 (::)

单目:: 指定为全局标识符

全局类型名、全局变量名、全局函数名等

```
int  x;
void  f ()
{
    int  x;
    :: x = 10;   // 全局变量
    x =20;       // 局部变量
}
```

4.1 作用域

有同名符号时，指定是用哪儿定义的符号(::)

双目:: 指定类或者名字空间中的枚举元素、数据成员、函数成员、类型成员等。

用法：类名::成员名

::的优先级为最高级，结合性自左向右。

```
class STACK{
        struct NODE { NODE(int v); };
};
STACK::NODE::NODE(int v) { } //自左向右结合
```

4.1 作用域


```
class  POINT2D{
    int  x, y;
public:
    int  getx( ) {return x; };
    POINT2D (int x, int y){
        POINT2D::x=x;
        this->y=y;                    //等价于POINT2D:: y=y
    }                                 //  *this.y = y;
} p(3,5);
static  int  x=7;
void main(void) {
    int  x=p.POINT2D::getx( ); //等价于x=p. getx( )
    ::x=POINT2D(4,7).getx( );
                //常量对象POINT2D(4,7)作用域局限于表达式
}
```

4.2 名字空间

- 名字空间可分多次和嵌套地用namespace定义

```
namespace A {
    int x;
    namespace B {
        namespace C {
            int k=4;
        }
    }
}
namespace AB=A::B;
 using namespace A::B::C;
   using namespace AB;//A::B无成员可用
```

4.2 名字空间

- 直接访问成员

  std::cout <<“hello”<<std::endl;

- 引用名字空间的某一个成员

  using std::cout;

  cout<<“hello”<<std::endl;

- 引用名字空间

  using namespace std;

  cout<<“hello”<<endl;

- 先定义、后引用

4.2 名字空间


```
namespace ALPHA {          //初始定义ALPHA
   int x;
   void g(int t);          //声明void g(int)
    g(long t){ ……  };      //定义void g(long)
}
namespace ALPHA {          //扩展定义ALPHA
   int y=5;                //定义整型变量y
   void g(void);           //新函数void g(void)
}
 using ALPHA::g;           //声明引用名字空间void g(int)和g(long)
void main(void) {
   g(ALPHA::x);            //调用函数void g(int)
}
```

4.2 名字空间


```
namespace A { int x=1; };
namespace B { int y=2; };
namespace C { int z=3; }
namespace   { int m=4; }
using namespace A; //此用法允许在全局作用域定义新x
using B::y;              //此用法不允许在全局作用域定义y
int z=x+3;  //访问A::x
int x=y+2;  //访问B::y，，此时定义了一个全局变量 x
int v=::x+A::x;  //用::区分全局变量x和名字空间成员x
//int y=4;    //错误，当前作用域有变量y
int main(void){   return  z;  }    //优先访问全局变量::z
```

4.3 成员指针

- 类数据成员指针
  - 什么是类数据成员指针？
  - 与普通的数据指针有何差别？
  - 使用数据成员指针有何优点？
  - 如何使用数据成员指针？
  - 数据成员空指针与普通数据空指针的差别
- 类成员函数指针

4.3 成员指针


```
Class Student {
public:
    int  number;
    char name[15];
    float score;
public: ……
};
Student xu(123,"Xuxiangyang",100);
Student zhang(456,”Zhangsan”,99);
```

p、q 普通的数据指针

p = &xu.number;

若想知道number 在类Student中偏移地址，怎么做？

为何要知道某一个成员的偏移地址？

```
int *p=&xu.number;        // p 指向对象 xu中的number
int *q=&zhang.number;     // q 指向对象 zhang中的number
cout<< *p << endl;        // 输出 123
cout<< *q << endl;        // 输出 456
```

4.3 成员指针


```
Class Student {
public:
    int  number;
    char name[15];
    float score;
public: ……
};
Student xu(123,"Xuxiangyang",100);
Student zhang(456,”Zhangsan”,99);

int x=offsetof(Student, number);   //取偏移地址
Student * p;                       // 注意与成员指针的差别
p = &xu;        p=&zhang;
*(int *)((char *)p + x)  =100 ;
```

4.3 成员指针


```
Class Student {
public:
    int  number;
   ……
};
Student xu(123,"Xuxiangyang",100);
Student zhang(456,”Zhangsan”,99);
int *p=&xu.number;        // p 指向对象 xu中的number
int Student::*q = &Student::number;  // q 数据成员指针
     //int Student::*q;      q= &Student::number;
cout << xu.*q<<endl;   // cout << xu.number <<endl;
cout << zhang.*q <<endl;
int *p=&Student::number; //无法从 Student::* 转换为 int *
```

若想知道number 在类Student中偏移地址，怎么做？

指向的类型 类名::* 变量名

4.3 成员指针

私有数据成员，数据指针

```
Class Student {
private:
    int  number; …
public:
    int  * getaddress1()
    {return   &number; }

    int   getaddress2()
    { return  int (&number); };
};
Student xu(123,"Xu",100);
```

```
void main()
{
    int  *p;
    int    q;
    p  =  xu.getaddress1();
    q  =  xu.getaddress2();
    cout<< *p << endl;
    cout<< *(int *)q;
}
```

结果皆为 123

p, q为xu.number的地址

不是number 在类中的偏移地址

4.3 成员指针

数据成员指针 VS 数据指针

私有成员的访问

```
Class Student {
private:
    int  number; …
public:
    int *  getaddress1()
    { return  &number; };
    int Student::*getaddress2() {
        return &Student::number;  }
};

Student xu(123,"Xu",100);
```

```
void main()
{
    int  *p;
    int  Student::*q;
    p=xu.getaddress1();
    q = xu.getaddress2();
   cout<< *p;
   cout<<xu.*q;
}
```

结果皆为 123

将 q=… 带入 xu.*q

```
cout<<xu.*xu.getaddress2();
```

4.3 成员指针

数据成员指针

```
Class Student {
public:
    int *  getaddress1()
    { return  &number; };
    int * Student::getaddress2()
   {  return  &number; };
    int Student::*getaddress3() {
          return &Student::number;  }
 int  *Student::getaddress4() {
          return &Student::number;  }
};

Student xu(123,"Xu",100);
```

getaddress1
与 getaddress2等价

getaddress3 : 成员指针

getaddress4: 无法从 Student::* 转换为 int *

4.3 成员指针

数据成员指针　　类内写法 与 类外写法

```
Class Student {
public:
int Student::*getaddress3() {
        return &Student::number;  }
};

int Student::* Student::getaddress3() { // 类成员 Student::get….
                                        // 指向 一个成员
        return &Student::number;
}

Student xu(123,"Xu",100);
```

4.3 成员指针

数据成员空指针

```
int    *p;

int  Student::*q;
```

p=0;	// p中的值为 0	*p 不可用，程序异常
q=0;	// q 中的值为 0xffffffff	s.*q ， 结果不正确 程序可继续运行

4.3 成员指针

数据成员空指针

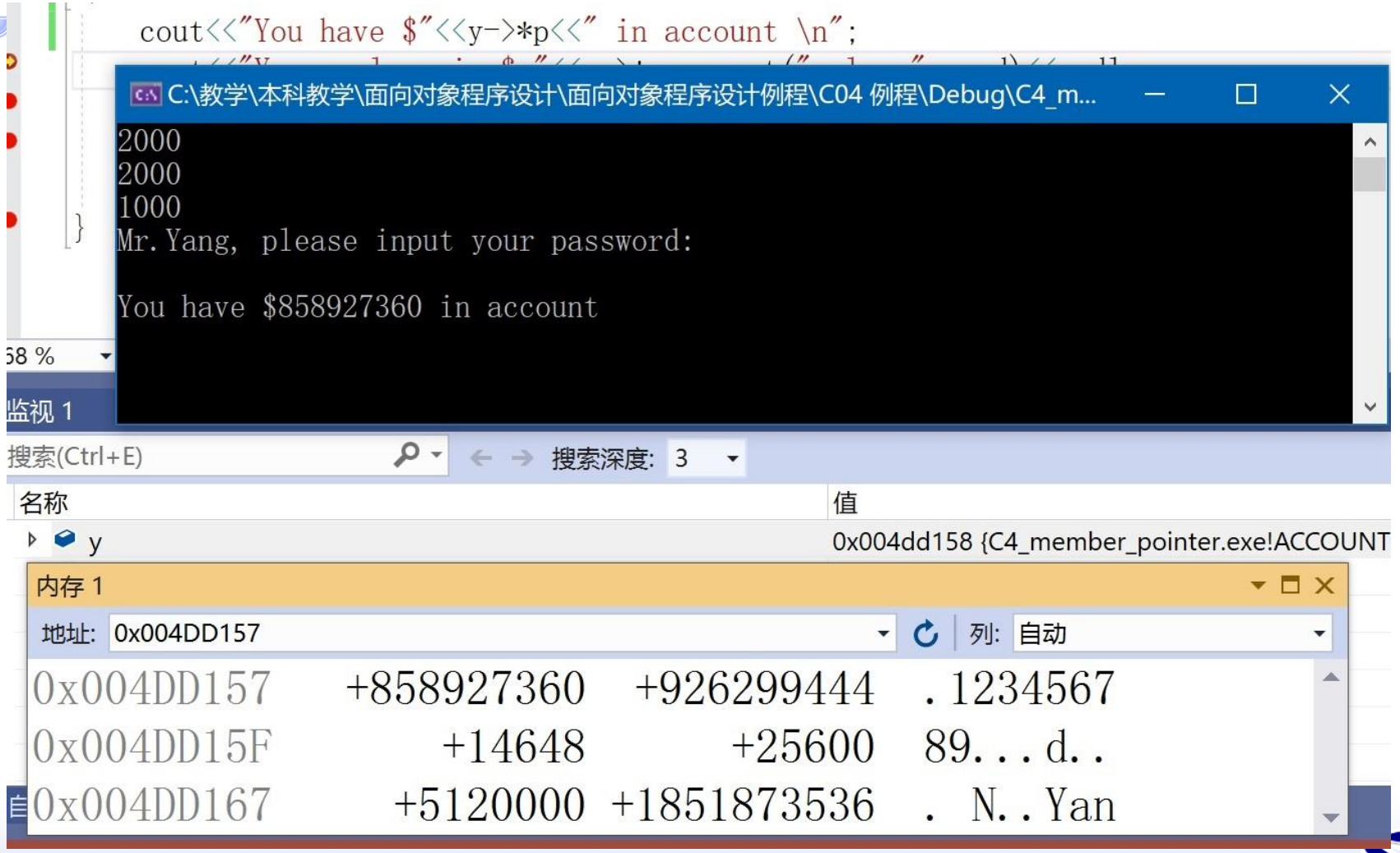
书 P70 – P71

```
char *getpswd(const char *name)   // 返回输入的密码

int  ACCOUNT::*ACCOUNT::get(char *item,char *pswd)
{
    if (stricmp(pswd,password))  return 0;
    ……
}
ACCOUNT wang(“Wang”,”abcdefghi”,1000,10000);
char *pswd = getpswd(wang.name);
cout<<wang.*wang.get(“salary”,pswd)<<endl;
```

当输入的密码不正确时，程序运行的结果是什么？

4.3 成员指针


cout<<"You have $"<<y->*p<<" in account \n";
C:\教学\本科教学\面向对象程序设计\面向对象程序设计例程\C04 例程\Debug\C4_m...
2000
2000
1000
Mr.Yang, please input your password:
You have $858927360 in account
监视 1
搜索(Ctrl+E)
搜索深度: 3
名称
值
y
0x004dd158 {C4_member_pointer.exe!ACCOUNT
内存 1
地址: 0x004DD157
列: 自动
0x004DD157 +858927360 +926299444 .1234567
0x004DD15F +14648 +25600 89...d..
0x004DD167 +5120000 +1851873536 . N..Yan

4.3 成员指针

类成员函数指针

- 指针函数与函数指针
- 普通的函数指针的用法与优点
- 类成员函数指针的定义和使用

4.3 成员指针——函数指针

指针函数 ： 返回结果是一个指针

```
char * fun(……);
```

函数指针：**是一个指针，指向一个函数**

```
int  fadd(int x, int y)
{   return x+y;   }
```

```
int  fsubtract(int x, int y)
{   return x-y;  }
```

```
int  (*fp)(int, int);
fp=&fadd;
result=(*fp)(10,20);
```

```
fp=fadd;
result=fp(10,20);

fp=fsubtract;
result=fp(10,20);
```

4.3 成员指针——函数指针


```
int  fadd(int x, int y);

Result=fadd(22,33)

002043B0  push      21h
002043B2  push      16h
002043B4  call   fadd
```

```
fp = fadd;
result=fp(22,33);

mov    dword ptr [fp],offset fadd
call     dword ptr [fp]
```

```
int  (*fp)(int, int);
fp=&fadd;
mov  dword ptr [fp],offset fadd

result=(*fp)(22,33);

003543C1  push      21h
003543C3  push      16h
003543C5  call    dword ptr [fp]
```

取函数地址时，有无 & 一样；
调用函数指针是，有无 * 一样

4.3 成员指针


```
Class Student {
public:
        void  SetNumber(int  x) { number=x; }
};
Student xu(123,"Xu",100);

void  (Student:: *pf) (int) ;   //成员函数指针
void  (*pq)(int) ;              // 函数指针
pf=&Student::SetNumber;
(xu.*pf)(200);
xu.SetNumber(200);     // 等同语句
```

4.3 成员指针——函数指针

不能取构造函数的地址，否则通过函数指针就可实现手动调用；

可以取析构函数的地址，通过函数指针可以手动调用析构函数。

4.3 成员指针

成员指针：指向类的**普通成员指针**

静态成员指针

普通成员指针

数据成员指针

函数成员指针

静态成员指针

静态数据成员指针

静态函数成员指针

数据成员、函数参数和返回类型都可定义为成员指针类型。

4.3 成员指针

- 使用普通成员指针访问成员时必须和对象关联
- 使用静态成员指针时不必和对象关联
- 普通成员指针通过.*和->*访问对象成员
- .*和->*优先级高， 结合性自左至右
- .*左操作数为类的对象，右操作数为成员指针

  int Student::*p;　　xu.*p
- ->*左操作数为对象指针，右操作数为成员指针

  Student *q;　　q->*p

4.3 成员指针

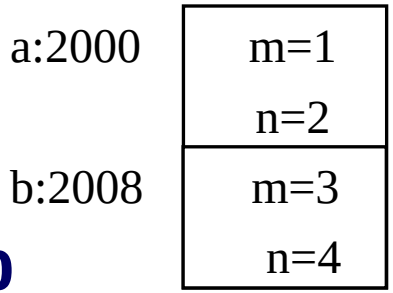
- 普通成员指针是一个偏移量，存放的不是成员地址，故不能移动：

```
int  Student::*p;
p=p+1;        // 非法，不能移动指针
int  *q;
q = q+1;
```

- 普通成员指针不能进行类型转换：
  - 防止通过类型转换间接实现指针移动。

4.3 成员指针

```
struct A{   //普通成员指针是偏移量
   int m, n;
}a={1,2}, b={3,4};
void main(void){
                //以下p=0表示偏移，实现时实际<>0
   int x, A::*p=&A::m;    //p=0：m相对结构体的偏移
   x=a.*p;                //x=*(a的地址+p)=*(2000+0)=1
   x=b.*p;                //x=*(b的地址+p)=*(2008+0)=3
   p=&A::n;               //p=4：n相对结构体的偏移
   x=a.*p;                //x=*(a的地址+p)=*(2000+4)=2
   x=b.*p;                //x=*(b的地址+p)=*(2008+4)=4
}
```



4.3 成员指针

```
struct A{   //struct定义的类，进入类体缺省权限为public
    int i, f( ){ return 1; } ;    //公有成员i、f( )
private:
    long j; //私有的成员j
}a;
void main(void){
   int  A::*pi=&A::i;          //普通数据成员指针pi指向public成员A::i
   int(A::*pf)( )=&A::f;       //普通函数成员指针pf指向函数成员A::f
   long  x=a.*pi+(a.*pf)( ); //等价于x=a.*(&A::i)+(a.*(&A::f))( )=a. i+a.f( )
   pi++;   pf+=1;  //错误，  pi不能移动，否则指向私有成员j，pf不能移动
   x=(long) pi;                //错误，pi不能转换为长整型
   pi=(int A::*)x;             //错误，x不能转换为成员指针
}
```

4.4 const、volatile和mutable

const 可修饰

- 普通变量
- 类的数据成员
- 函数的参数，成员函数的参数
- 函数成员
- 对象
- 函数的返回类型

4.4 const、volatile和mutable


```
class TUTOR{
    char        name[20];
    const       char  sex;      //性别为只读成员
    int         wage;
public:
    TUTOR(const char *n, char g, int s): sex(g), wage(s)
        { strcpy_s(name,n); }
    const char *getname( ) const{ return name; }
          //函数体不能修改当前对象   函数的返回类型有 const 修饰
    char *setname(const char *n)
        { strcpy_s(name, n);      return name;  }
};
TUTOR xu("xuxy",'M',2000);
*xu.getname()='X';    //  不能给常量赋值
*xu.setname("xuxiangyang") ='X';  // name 的首字母变成X
strcpy_s(xu.setname("xu123"), 6, "hello"); //name 改为hello
```

4.4 const、volatile和mutable


```
int                 const  int
int *               const  int  *
                    const char *getname( );
                    const char *pc;      // *pc=…   错误
                                         // *getname() = …
```

const 变量是右值

4.4 const、volatile和mutable

volatile：

不稳定的、易挥发的、变化无常的

变量可能会被意想不到的改变；

优化器不对该变量的读取进行优化，用到该变量时重新读取。

正因为变化无常，而不能对涉及到该变量的语句进行优化。

4.4 const、volatile和mutable

```
#include <iostream>
using namespace std;
int main()
{        int  i = 10;
         int  a = i;
         cout << "a = " << a << endl;
         __asm {
            mov dword ptr [ebp-8], 123
         }
         int b = i;
         cout << "b= " << b << endl;
         return 0;
}
```

在Debug 版本下，
a =10
b = 123

在Release 版本下，
a =10
b = 10

4.4 const、volatile和mutable

```
#include <iostream>
using namespace std;
int main()
{       volatile  int  i = 10;
        int  a = i;
        cout << "a = " << a << endl;
        __asm {
            mov dword ptr [ebp-8], 123
        }
        int b = i;
        cout << "b= " << b << endl;
        return 0;
}
```

在Debug 版本下，
a =10
b = 123

在Release 版本下，
a =10
b = 123

4.4 const、volatile和mutable

编译器对程序的优化

```
int main()
{
    int   i, j;
    i=1;
    i=2;
    i=3;
        // 上面的程序等价于 i=3;
    for (i=0;i<10000;i++)  j=0;
        // 延时程序，可优化掉无用的循环
}
```

4.4 const、volatile和mutable

```
int   a;
const  int  x=100;

cout<<x<<endl;

*(int *)&x =200;

cout <<x<<endl;
```

```
int   a;
a=100;
const  int  x=a;

cout<<x<<endl;

*(int *)&x =200;

cout <<x<<endl;
```

4.4 const、volatile和mutable

```
int  flag=0;
int main()
{
    while (1)  {
      if (flag)
        dosomething();
    }
}

// 中断服务程序   程序2
void ISR()
{    flag=1;
}
```

编译器可能认为main 函数中未修改过 flag;

将 flag 读入到一个寄存器中；后面只用寄存器中的副本；导致flag 修改后未发现；dosomething 不被执行

4.4 const、volatile和mutable

volatile：

优化器不对该变量的读取进行优化，用到该变量时重新从它所在的内存读取数据。

修饰的变量可由操作系统、硬件、并发执行的线程在程序中进行修改。

以下情况下，应在变量前 加 **volatile**

- 多任务环境下，各任务间共享的变量；
- 中断服务程序中修改的供其他程序检测的变量；
- 存储器映射的硬件寄存器；

4.4 const、volatile和mutable

- **volatile**可以修饰变量、类的数据成员、函数成员及普通函数的参数和返回类型。
- 构造或析构函数的参数表后不能出现**const**或**volatile**

  classname(….) const;　// error

  构造或析构时，**this**指向的对象应能修改且不随便变化。

- 静态函数成员参数表后不能出现**const**或**volatile(**无隐含**this)**。

  static … ff(…) const;　// error

4.4 const、volatile和mutable

mutable:

可变的

- 是const 的反义词
- 为突破 const的限制而设置的
- 被mutable 修饰的变量永远处于可变得状态，即使在const函数中
- mutable只能用来修饰数据成员
- 不能与 const、volatile 或 static 同时出现

4.4 const、volatile和mutable


```
class A {
        mutable int  x;
public:
        void  f() const
        {
                x = x + 1;
        }
};
//  若 x的定义改为      int  x;
//  编译时，语句  x=x+1; 报错，无法修改 x
```

4.4 const、volatile和mutable

- 参数表后出现const、volatile或const volatile会影响函数成员的重载：
  - 普通对象应调用参数表后不带const和volatile的函数成员；
  - const和volatile对象应分别调用参数表后出现const和volatile的函数成员。
- 参数表后出现volatile，表示调用函数成员的对象是挥发对象，意味存在并发执行的进程，正在修改当前对象。

4.4 const、volatile和mutable


```
class  A{
   int a;  const  int  b; //b为const成员
public:
   int f( ){a++; return a; } //this类型为： A * const  this
   int f( )const{return a; } //this类型为：const A * const  this。
   int f( )volatile{return a++; } //this类型为：volatile A * const this
   int f( )const volatile{ return a; }//this类型为：const volatile A* const this
   A(int x) ： b(x)  { a=x; } //不可在函数体内对b赋值修改
} x(3);                          //等价于A  x(3)
const A y(6);                    //  y 不可修改
const volatile A z(8);           //  z  不可修改
void main(void) {
    x.f( ); //普通对象x调用int f( ): this指向的对象可修改
    y.f( ); //只读对象y调用int f( )const:this指向的对象不可修改
    z.f( ); //只读挥发对象z调用int f( )const volatile:this指向的对象不可修改、挥发
}
```

4.5 引用对象

左值表达式　　VS　右值表达式

引用变量　-》　引用对象

左值引用　　VS　右值引用

移动构造函数

移动赋值函数

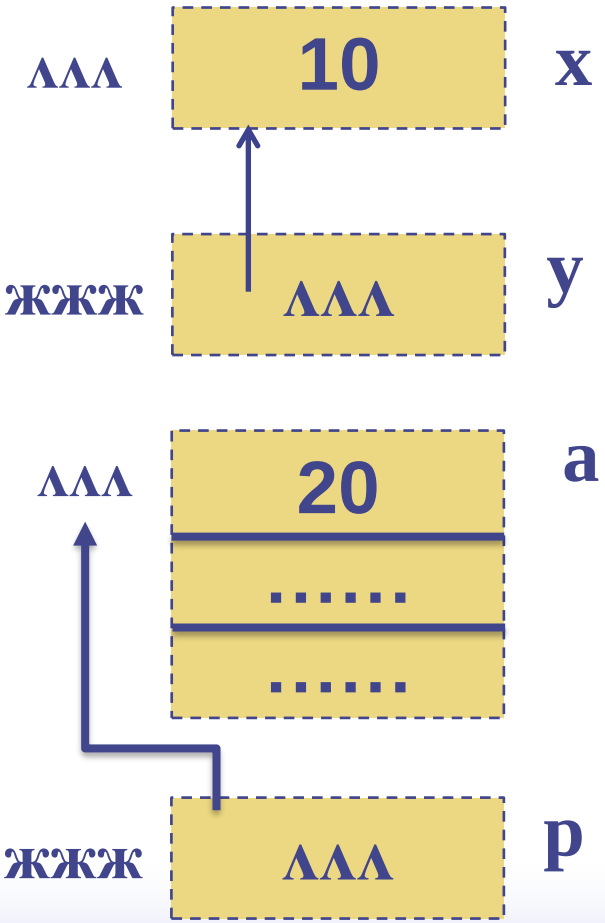
4.5 引用对象

引用简单类型的变量

```
int  x;
int  &y = x;
y=10;
```

引用对象

```
struct A { int x;
           …… };
A   a(……);
A   &p = a;
p.x = 20;
```



引用变量、引用对象 在定义时赋初值

4.5 引用对象

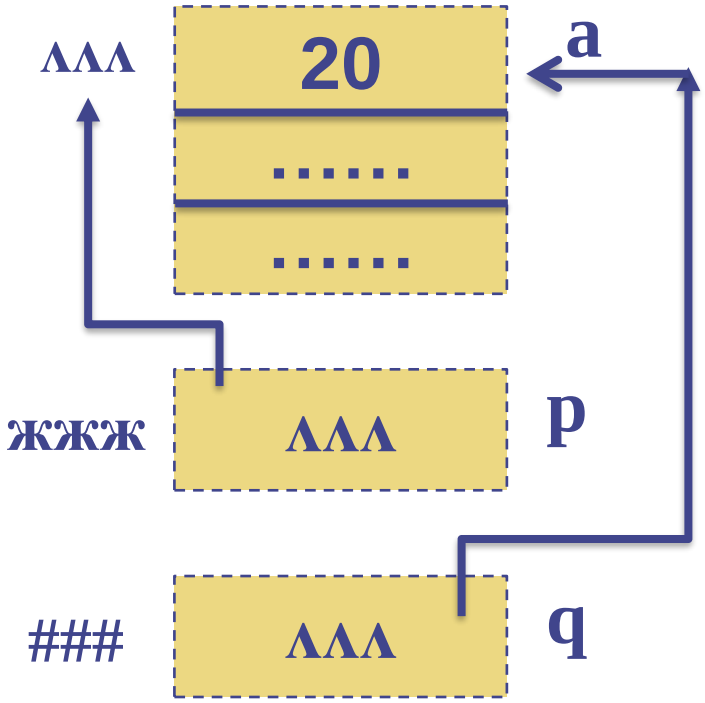
华中科技大学

引用对象

```
struct A { int x;
           ..... };
A    a(.....);
A    &p = a;
p.x = 20;
```

对象指针

```
A    *q ;
q= &a;
q->x = 20;
(*q).x=20;
```



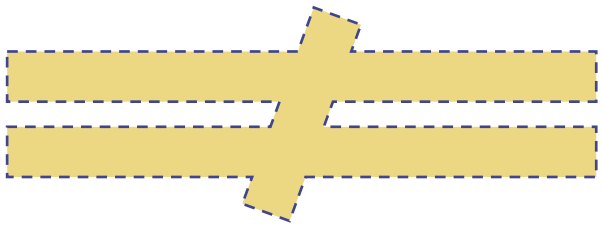
引用对象 在定义时赋初值

对象指针定义、赋值可分离

4.5 引用对象

引用对象作为函数参数：传入的是对象地址

引用对象作为函数返回值：传出对象地址



以对象作为函数参数

以对象作为函数的返回值

4.5 引用对象

复习：以对象作为函数参数时应注意的问题

- **值参(非左值引用形参)** 相当于函数体内的局部变量。

  f(classname a)

- 值参对象的构造在函数调用时通过值参传递赋值完成

  (浅拷贝)

  或通过调用以对象为参数的复制的构造函数完成

  以对象为参数的复制的构造函数：A(const A & a)

- 值参为对象则在函数返回前析构
- 若对象包含指针类型的数据成员，浅拷贝只复制指针的值而未复制指针所指的单元内容，实参和形参两变量的指针成员指向同一块内存。

4.5 引用对象

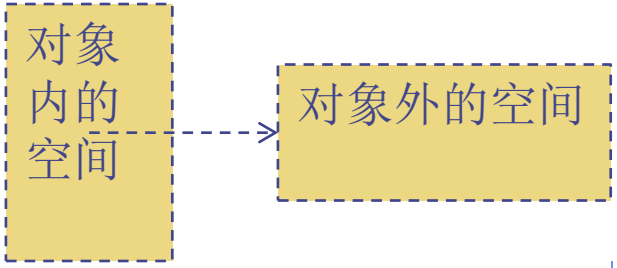

```
class A {
    int  size, *p;
public:
    int  get(int x) { return p[x]; };
    A(int s);
    A(A &r); //A (const A&r)深拷贝构造，调用f时不会浅拷贝传递值参
    ~A( ){
       if(p){delete p; p=0;} cout<<"Destruct A("<<size<<")\n";
    }
};
A::A(int s){
    p=new int[size=s];
    for(int i =0; i<s; i++)  p[i]=1;
    cout<<"Construct A("<<s<<")\n";
}
```

4.5 引用对象

- 引用对象可视为被引用对象的别名
- 一般情况下，有被引用的对象负责构造和析构
- 如果定义引用变量时，使用了构造函数，则应注意析构

```
A  &r = *new A(.....),
delete  &r;
```

否则可能造成内存泄漏。



- 注意比较差异

r.~A( )仅析构被引用对象，而不释放其所占内存；

free(&r)仅释放了被引用对象所占内存而未对其析构。

```
delete &r;
```

4.5 引用对象

- 引用变量必须在定义的同时初始化
- 引用参数则是在调用函数时初始化
- 左值引用变量和参数

  应使用同类型的左值表达式初始化

4.5 引用对象

左值引用 VS. 右值引用

```
int  x=10;
int  &y =x;   // x 是一个左值
int  &z = x*2;  //错误，无法从 int 转换为 int &
   // z 定义为一个引用，z 中应存放一个地址
   // 而 x*2 会对应一个值，无法取其地址，送入 z 中。

int  && w = x*2; // 右值引用；x*2是一个右值；
   // 将 x*2 存放在一个临时单元中；将该单元的地址送入 w 中
```

4.5 引用对象

右值引用

```
void f(int&& r)
{       cout << “r=” << r << endl;
        r = 30;
}

int w1 = 10;
int w2 = 25;
f(w1 + w2);     // 显示 r=35
f(10);          // 显示 r=10
f(w1++);        // 显示 r = 10, w1 为11
```

4.5 引用对象

右值引用

```
        f(w1 + w2);
mov     eax,dword ptr [w1]
add     eax,dword ptr [w2]
mov     dword ptr [ebp-0E0h],eax
lea     ecx,[ebp-0E0h]
push    ecx
call    f (02C12A8h)
```

传递给函数f 的是临时空间的地址

4.5 引用对象

右值引用

```
        f(10);
mov     dword ptr [ebp-0ECh],0Ah
lea     eax,[ebp-0ECh]
push    eax
call    f (02C12A8h)
```

传递给函数f 的是临时空间的地址

4.5 引用对象

右值引用

```
void f(int&& r)
{       ……;
        r = 30;
}
```

```
void f(int  r)
{       ……;
        r = 30;
}
```

```
int w1 = 10,   w2 = 25;
f(w1 + w2);    // 显示 r=35
f(10);         // 显示 r=10
f(w1++);       // 显示 r = 10, w1 为11
```

疑问：用右边的函数代替左边的函数，

运行结果相同，为什么要用右值引用？

4.5 引用对象

右值引用

移动语义：

通过移动构造函数与移动赋值运算符来实现。

```
STACK(STACK &&s);      //移动构造
virtual STACK &operator=(STACK &&s); //移动赋值

STACK(const STACK &s);  //对象构造
virtual STACK & operator=(const STACK &s);
                                    // 对象赋值
```

4.5 引用对象

右值引用 的作用

移动语义：

通过移动构造函数与移动赋值运算符来实现。

将内存的所有权从一个对象转移到另外一个对象，高效的移动用来替换效率低下的复制，对象的移动语义需要实现移动构造函数（move constructor）和移动赋值运算符（move assignment operator）。

4.5 引用对象

右值引用

- 以右值引用为参数的构造函数
- 以右值引用为参数的构造函数的调用
- 右值引用的应用场景
- 如何实现移动语义?

4.5 引用对象


```
class  STACK {
    private:
        int * const elems;  // 元素数据区指针
        const  int  max;    //  数组元素最大个数
        int      pos;
Public:
     STACK  CreateTemp();
};

STACK  c = a.CreateTemp();  // 调用移动构造
```

4.5 引用对象

右值引用　移动构造

```
STACK  CreateTemp() {
    cout << "Enter CreateTemp" << endl;
    STACK  temp(max);
    int i;
    for  (i = 0;i < max;i++)
        temp.elems[i] = 1;
    temp.pos = max;
    cout << "Exit CreateTemp" << endl;
    return temp;
}  // 先构造对象temp。将temp返回时，调用移动构造
```

4.5 引用对象

STACK c = a.CreateTemp();

在调用时，对应的汇编语句：

```
lea     eax,[c]
push    eax
lea     ecx,[a]
call    STACK::CreateTemp (0F21500h)
```

STACK c = a.CreateTemp();

构造函数，将被构造对象c 的地址压入了堆栈中

4.5 引用对象

STACK CreateTemp() { …… 最后一条语句

```
return temp;
 lea        eax,[temp]
 push       eax          // 传递局部对象的地址
 mov        ecx,dword ptr [ebp+8]   // 待构造对象的地址
 call       STACK::STACK (0F21505h)   // 移动构造
 mov        ecx,dword ptr [ebp-104h]
 or         ecx,1
 mov        dword ptr [ebp-104h],ecx
 mov        dword ptr [ebp-4],0FFFFFFFFh
 lea        ecx,[temp]                // 析构 temp
 call       STACK::~STACK (0F214DDh)
```

4.5 引用对象

右值引用 -> 移动构造

```
STACK(STACK  && rr) : max(rr.max),
                              elems(new int[rr.max]) {
   cout << "Const. STACK(STACK  && rr)" << endl;
   pos = rr.pos;
   memcpy(elems, rr.elems, max * sizeof(int));
}
STACK  c = a.CreateTemp();  // 调用移动构造
```

注意：上面的函数，只是有一个右值引用的外表，其实现与左值引用的函数体写法形同。还不是真正的移动构造。

4.5 引用对象

讨论：在函数中能否直接将 elems 指向 rr.elems?
并删除 memcpy 语句？

原移动构造函数

```
STACK(STACK  && rr) : max(rr.max),
                      elems(new int[rr.max])
{   cout << "Const. STACK(STACK  && rr)" << endl;
   pos = rr.pos;
   memcpy(elems, rr.elems, max * sizeof(int));
}
```

4.5 引用对象

讨论：在函数中能否直接将 elems 指向 rr.elems?

并删除 memcpy 语句？

```
STACK(STACK  && rr) : max(rr.max),
                      elems(rr.elems)
{   cout << "Const. STACK(STACK  && rr)" << endl;
   pos = rr.pos;
}
```

修改后，语法上没有错误，但执行有错误

实现的是一种浅拷贝，同一片区域会被释放多次。

CreateTemp() 中要析构局部对象；

此处被构造的对象，以后也要析构

4.5 引用对象

讨论：在函数中能否直接将 elems 指向 rr.elems?

并删除 memcpy 语句？

```
STACK(STACK  && rr) : max(rr.max),
                      elems(rr.elems)
{   cout << "Const. STACK(STACK  && rr)" << endl;
   pos = rr.pos;
}
```

修改后，语法上没有错误，但执行有错误

能不能在 CreateTemp 中的局部对象采用指针，这样指针指向的空间就不会释放了？

4.5 引用对象

讨论：在函数中能否直接将 elems 指向 rr.elems?

并删除 memcpy 语句？

```
STACK  CreateTemp_Move() {
    cout << "Enter CreateTemp_Move" << endl;
    STACK  *temp=new STACK(max);
    for (int i = 0;i < max;i++) temp->elems[i] = 1;
    temp->pos = max;
    return *temp;   }
```

修改临时对象为指针，避免临时数据的释放，

但最后调用的是以左值引用为参数的构造函数。

temp 不再是临时的(函数结束后仍存在).

4.5 引用对象

讨论：在函数中能否直接将 elems 指向 rr.elems?

并删除 memcpy 语句？

STACK CreateTemp { ..... } 保持不变

但让 临时对象 temp 的elems 指向NULL

```
// 移动构造
STACK(STACK&& rr) : max(rr.max), elems(rr.elems)
{
  cout << " STACK(STACK  && rr)" << endl;
  pos = rr.pos;
 *(char **)&(rr.elems) = NULL;
}
```

4.5 引用对象

右值引用

移动语义：

通过移动构造函数与移动赋值运算符来实现。

将内存的所有权从一个对象转移到另外一个对象，高效的移动用来替换效率低下的复制，对象的移动语义需要实现移动构造函数（move constructor）和移动赋值运算符（move assignment operator）。

4.5 引用对象

不产生临时对象，不会调用移动构造函数

```
STACK  d = a.Create_Directly();
STACK  Create_Directly() {
   cout << "Enter Directly" << endl;
   return STACK(max);  // 注意与CreateTemp的区别
}
```

```
        return STACK(max);
 mov        ecx,dword ptr [eax+4]
 push       ecx
 mov        ecx,dword ptr [ebp+8]
 call       STACK::STACK (06414E7h)
```

// 直接调用的是STACK的构造函数，不产生临时对象

4.5 引用对象

右值引用　　移动赋值

```
STACK  a(10);
STACK  d(5);
d=a.CreateTemp();
```

4.5 引用对象

右值引用　移动赋值

```
STACK& operator =(STACK && v) {
        cout << "operator = (STACK  && v)" << endl;
        if (elems != 0)     delete[] elems;
        *(int*)& max = v.max;
                // 注意max的定义是 const int max;
                // *const_cast<int *>(&max) = v.max;
        *(int**)& elems = v.elems;
        pos = v.pos;
         *(int**)& v.elems=NULL;
        return *this;  }
```

4.5 引用对象

右值引用

移动语义：

通过移动构造函数与移动赋值运算符来实现。

将内存的所有权从一个对象转移到另外一个对象，高效的移动用来替换效率低下的复制，对象的移动语义需要实现移动构造函数（move constructor）和移动赋值运算符（move assignment operator）。

4.5 引用对象

左值可以绑定到左值引用，

右值可以绑定到右值引用

任何东西都可以绑定到左值引用const。

也就是说，不能将右值绑定到左值引用。

4.5 引用对象

c3 = c1+c2;　c1,c2,c3 都是 OneArray的对象

```
OneArray& operator=( OneArray& a) {
  //在这一句中去掉const会报错，为什么呢?
        length = a.length;
        delete[]data;
        data = new int[a.length];
        for (int i = 0; i < a.length; i++) {
              data[i] = a.data[i];
        }
        return *this;
    }
```

4.5 引用对象

c3 = c1+c2;　c1,c2,c3 都是 OneArray的对象

```
OneArray& operator=( OneArray&& a) {
  //不要const，是正确的。右值引用
                length = a.length;
                delete[]data;
                data = new int[a.length];
                for (int i = 0; i < a.length; i++) {
                        data[i] = a.data[i];
                }
                return *this;
        }
```

练习

P 85　习题 4.9

```
struct A {
    char* a;
    char  b;
    char  * geta();
    char A::* p;        // 成员指针，指向一个char 类型的成员
    char * A::* pp;     // 指向一个 char * 类型的成员
    char* A::* q();     // 函数返回一个指向 char *类型的成员
    char* (A::* r)();   //  成员函数指针，
                        //  指向返回类型为char *的函数
};
```

练习


```
struct A {
    char* a;
    char  b;
    char  * geta();
    char A::* p;        //  a.p = &A::b;
    char * A::* pp;    // a.pp = &A::a;
    char* A::* q();
    char* (A::* r)(); //  a.r = &A::geta;
}a;
 char* A::* A::q()  { return &A::a;  }
 char* A:: geta()   {  return 0; }
```