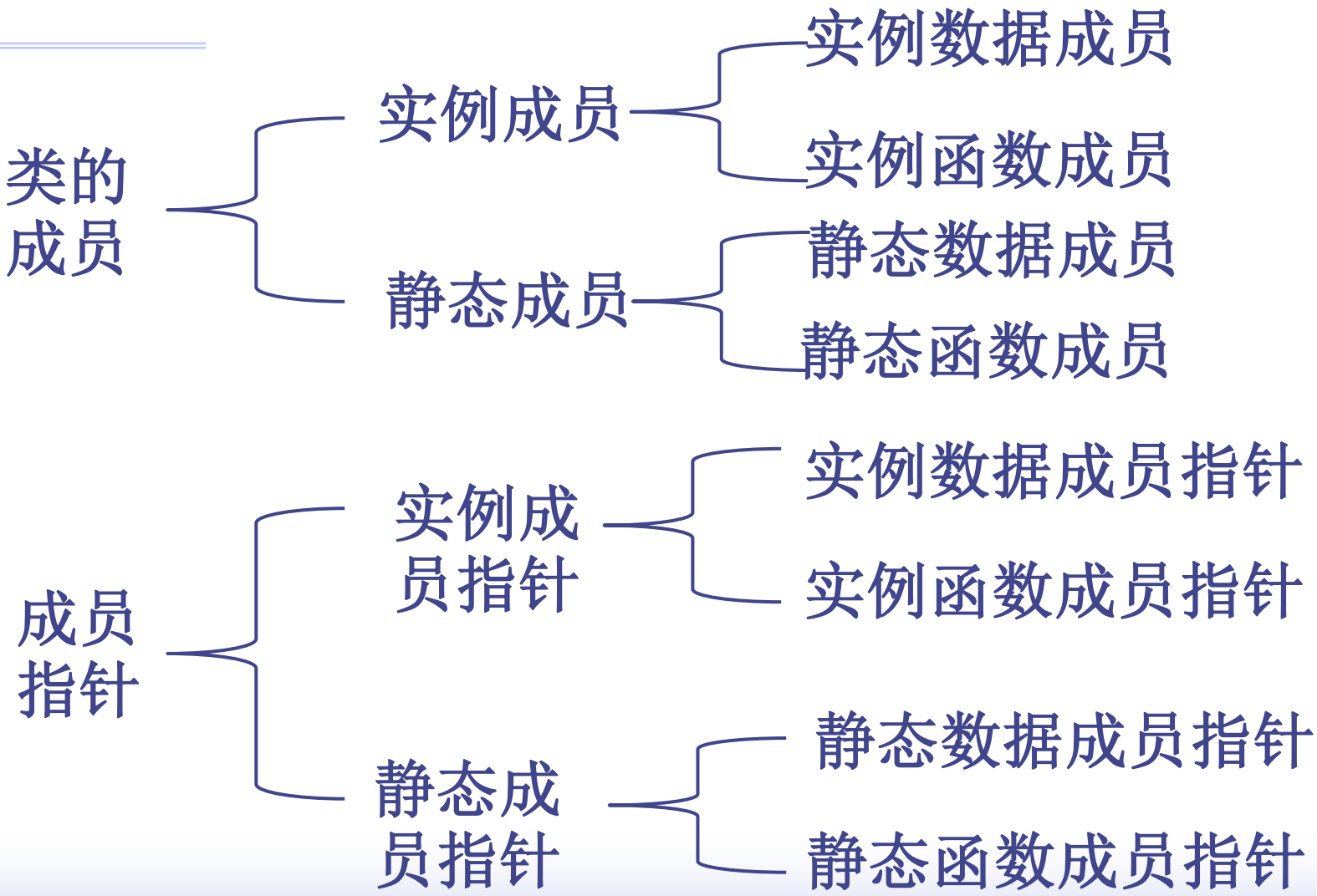
第5章 成员及成员指针

许向阳

xuxy@hust.edu.cn

内容

内容概要


类的成员
实例成员
实例数据成员
实例函数成员
静态成员
静态数据成员
静态函数成员
成员指针
实例成员指针
实例数据成员指针
实例函数成员指针
静态成员指针
静态数据成员指针
静态函数成员指针

内容概要

实例数据成员：

不同对象的数据成员各有独立的空间

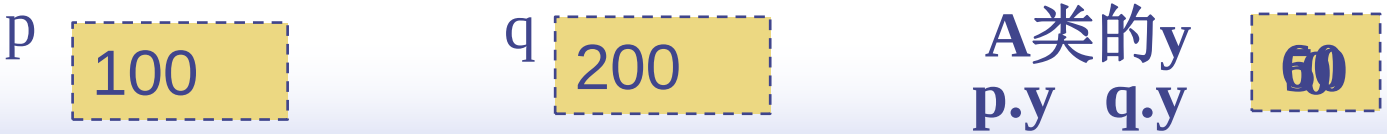
静态数据成员：

不同对象的静态数据成员 共享存储同一空间；

对象的存储空间中，不包含静态数据成员。

实例数据成员 VS 静态数据成员

```
class A {
public:
    int    x;            // 实例数据成员
    static int y;        // 静态数据成员
};                       // sizeof(A) = 4
int A::y=0;
A   p,  q;
p.x =100;          p.y=50;
q.x = 200;         q.y=60;
```



内容概要

实例函数成员：

对象的地址为第一参数，即隐含参数 **this**

静态函数成员：

无对象的地址。

实例函数成员 VS 静态函数成员

```
class A {
public:
    int   f( ) ;            // 实例函数成员
    static int g( );        // 静态函数成员
}
int A::y=0;
A   p,  q;
p.f( );
q.f( );
A::g( );  与 p.g( ) ;  与  q.g( ); 等价
```

核心差别：有无 对象的地址作为第一个参数。

5.1 实例成员指针

- 实例数据成员指针
  - 什么是实例数据成员指针？
  - 与普通的数据指针有何差别？
  - 使用数据成员指针有何优点？
  - 如何使用实例数据成员指针？
  - 实例数据成员空指针与普通数据空指针的差别
- 实例函数成员指针

5.1 实例成员指针

- 实例数据成员指针 指向 类的一个实例数据成员
- 存放数据实例成员在类中的偏移地址
- 存放的不是一个对象中实例成员的地址
- 数据成员指针 不一定是 类的成员

```
int    *p;     // 普通指针
int   类名A :: *q; //成员指针
                   // 指向 A 类中的一个 int 成员
      q = & 类名A :: A 中的int成员 ;
```

5.1 实例成员指针


```
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
```

5.1 实例成员指针


```
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 ;
```

5.1 实例成员指针


```
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

5.1 实例成员指针

普通数据指针

```
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

5.1 实例成员指针

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();
```

5.1 实例成员指针

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 *

5.1 实例成员指针

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);
```

5.1 实例成员指针

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
```

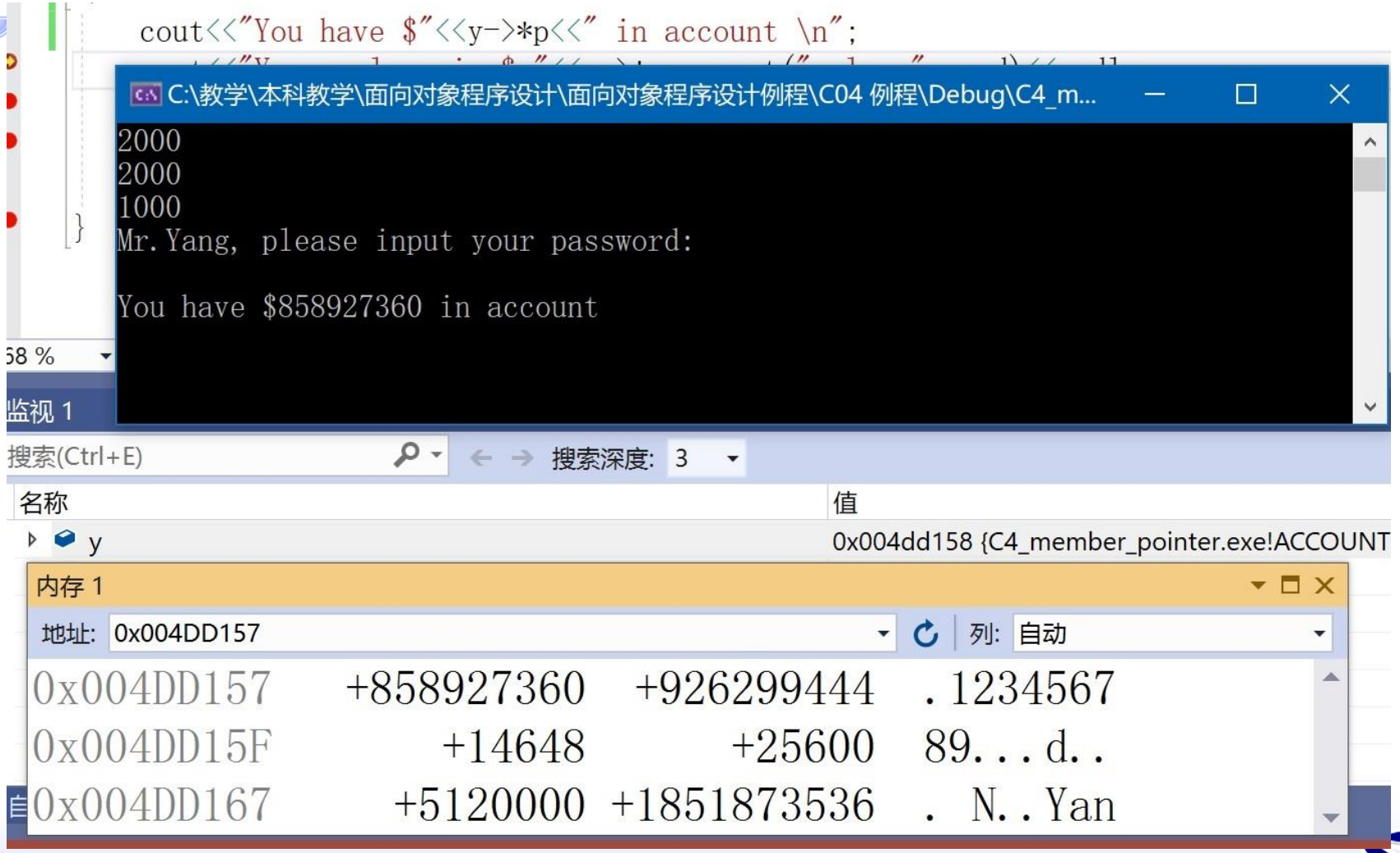
5.1 实例成员指针

数据成员空指针

```
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？

5.1 实例成员指针


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
58 %
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

5.1 实例成员指针

实例函数成员指针

- 指针函数与函数指针
- 普通的函数指针的用法与优点
- 实例函数成员指针的定义和使用

5.1 实例成员指针

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);
```

5.1 实例成员指针


```
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；
调用函数指针时，有无 * 一样

5.1 实例成员指针


```
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
```

5.1 实例成员指针

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。

5.1 实例成员指针

成员指针：指向类的成员，本身并非类的一个成员

实例成员指针

- 实例数据成员指针
- 实例函数成员指针

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。

5.1 实例成员指针

- 使用实例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
- 实例成员指针通过.*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

5.1 实例成员指针

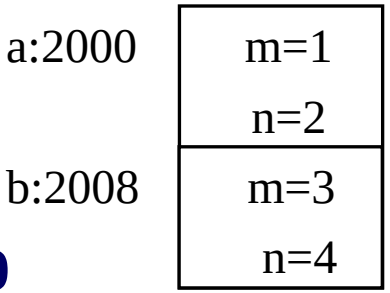
- 实例成员指针是一个偏移量，存放的不是成员地址，故不能移动：

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

- 实例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。

5.1 实例成员指针

```
struct A{   //实例成员指针是偏移量
   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
}
```



5.1 实例成员指针


```
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;          //实例数据成员指针pi指向public成员A::i
   int(A::*pf)( )=&A::f;       //实例函数成员指针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
}
```

5.2 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

5.2 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
```

5.2 const、volatile和mutable

函数的返回类型前有const

```
TUTOR : 有私有成员  char  name[20];
const char * TUTOR::getname( ) const{
    return name;
}       // 返回值的 类型 是 const char *.

TUTOR xu("xuxy",'M',2000);
char *p;
p=xu.getname( );        // 无法从 const char * 转换为 char *
const char *q;
q = xu.getname( );
*xu.getname( ) ='X';   // 不能给常量赋值
                       //  等同 *q='X';     q[0]='X';
                       //  表达式必须是可修改的左值
```

5.2 const、volatile和mutable

函数的返回类型前有const

```
TUTOR : 有私有成员  char  name[20];
const char * TUTOR::getname( ) const {return name;  }

TUTOR xu("xuxy",'M',2000);
const char *q;
q = xu.getname( );
q = “hello”;
```

讨论： 能否用 xu.getname( ) 代换q, 写出语句

xu.getname( )=“hello” ?

编译器给出的错误：

表达式必须是可修改的左值； = 左操作数必须是 左值。

5.2 const、volatile和mutable

讨论： 能否用 xu.getname( ) 代换q, 写出语句

xu.getname( )=“hello” ?

核心：函数是如何返回结果的！

```
int  f( ) {int x=20;  return x;}
        mov   dword ptr [x], 14h
        mov   eax, dword ptr [x]    // eax 存放返回结果
y =f( );
        mov    dword ptr [y], eax
```

eax 不是内存单元，无法获得其地址，特别是不允许修改eax中的值。 f( ) = 10; 就是企图 mov eax, 0ah

对于 const char * getname( ) const {… } 也是使用 eax来传递结果，同样，无法使用 xu.getname( ) =“hello”;

5.2 const、volatile和mutable

函数的返回引用 与 返回值的差别

```
int  & g( ) {int x=20;  return x;}                    int &g=x;
        mov   dword ptr [x], 14h
        lea     eax, [x]              // eax 存放 x 的地址
y =g( );                                              y=g;
        mov        eax, dword ptr [eax]
        mov        dword ptr [y],  eax
g( ) =10;                                             g=10;
       mov        dword ptr [eax], 0Ah
```

对于引用类型的返回，eax 中存放的是地址。可以将函数放在 = 的左边，将结果存放到 eax 所指向的内存单元。也即【eax】对应的单元。

5.2 const、volatile和mutable

函数的返回引用 与 返回值的差别

讨论： **const char * TUTOR::getname( ) const**

{return name; }

返回结果在 eax 中，其值代表的是单元的地址，
相当于指针。虽然不能修改 eax， 即 不能 **eax = p ;**
但能否使用 [eax] 的形式来修改指向单元的内容？
即 ***xu.getname( )='X' ;**

不行。函数前有 const 约束，相当于 const char *q;
不能有 ***q ='X'.**

5.2 const、volatile和mutable

函数的返回引用 与 返回值的差别

讨论： **char * TUTOR::getname( ) const**
{return name; }

能否直接去掉 函数返回上的 const 约束，然后：
***xu.getname( )='X' ;**

编译错误：返回值类型与函数类型不匹配。
return ： 无法从 const char [20] 转换为 char *.

getname 后面有一个 const, 即 代表
const TUTOR * const this. this->name 是 const的.

如何解决这一错误？

5.2 const、volatile和mutable

函数的返回引用 与 返回值的差别

方法1 ： 去掉 getname后的 const

```
char * TUTOR::getname( )
 { return  name;  }

*xu.getname( )='X' ;
```

方法2 ： 返回时进行强制类型转换

```
char * TUTOR::getname( ) const
 { return  (char *)name;  }

*xu.getname( )='X' ;
```

5.2 const、volatile和mutable

華中科技大學

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

const 变量是右值

5.2 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。

5.2 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

5.2 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

5.2 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
}
```

5.2 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;
```

5.2 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

5.2 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；

5.2 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

5.2 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

5.2 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
```

5.2 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。

5.2 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
}
```

5.3 静态数据成员

5.3 静态数据成员
5.4 静态函数成员
5.5 静态成员指针


```
static  int  *p;    // 静态数据成员
static  int   s;    // 静态数据成员
static  ..  f(…) ;  // 静态函数成员
int  *q =&C.s;   // 静态成员指针
int C:: *mq;     // 数据成员指针
```

```
class C { static int s; …};
```

静态成员指针就是普通指针，不同于实例数据成员指针；
指向一个静态成员。静态数据成员的空间不在对象内。

5.3 静态数据成员

定义与初始化

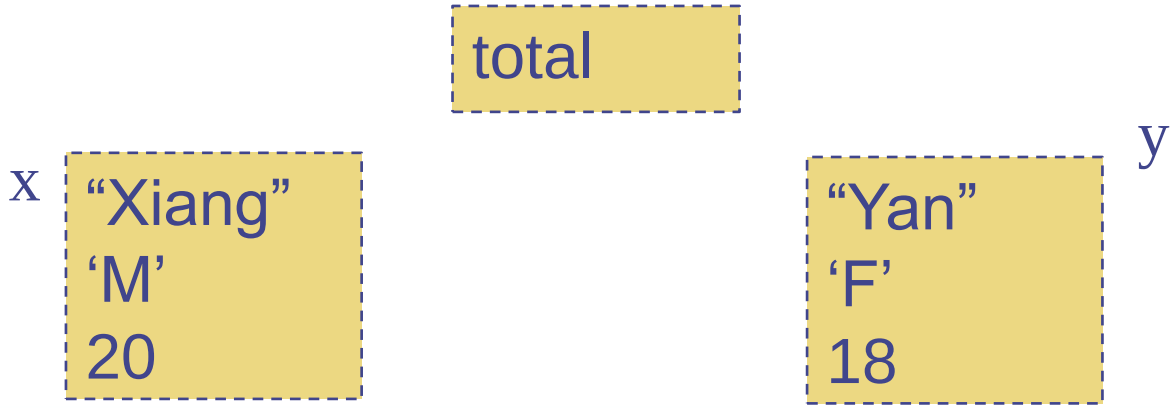
```
#include <iostream.h>
class HUMAN{
 private:
   char name[11];
    char sex;       int  age;
public:
   static int total;  //类体内声明，访问权限public
    HUMAN(char* n,char s,int a)
                    // 静态成员不能在函数体前初始化
   {     strncpy_s(name, n, 10);
          sex = s;     age = a;      total++;
    }
   ~HUMAN( ) {  total --; };
};

int  HUMAN::total=0;
```

//类体外定义并初始化，有访问权限的独立变量

5.3 静态数据成员


```
class HUMAN{
public:
   static int total; ……
};
int  HUMAN::total=0;
void main(void){
    HUMAN x((char *) "Xiang", 'M', 20);//HUMAN::total=x.total=1
    HUMAN y((char *) "Yan", 'F', 18); //HUMAN::total=x.total=y.total=2
}
```

空间分配



sizeof(HUMAN)=16，不包括静态成员total

5.3 静态数据成员

- 静态数据成员用 static 声明
- 静态数据成员的访问控制可以是公有、保护和私有
- 在类体内声明，在类体外定义并初始化
- 静态数据成员独立分配内存，不属于任何对象内存
- 所有对象共享静态数据成员内存，任何对象修改该成员的值，都会同时影响其他对象关于该成员的值。
- 用于描述类的总体信息，如对象总数、连接所有对象的链表表头等。

静态数据成员 VS 静态全局变量 VS 静态局部变量

作用域的差别？　生命周期：整个程序

5.3 静态数据成员

跳过

- 局部类不能定义静态数据成员

```
void f(void) {
    class T {          // T 是局部类
        int  c;
        static int d;  // 错误
    };
}
```

此规定合理吗？

- 联合体的数据成员必须共享存储空间，而静态数据成员各自独立分配存储单元，联合不能定义静态数据成员。

```
union UNTP{ static int c;  static long d; };        //错误
```

5.3 静态数据成员

- 静态且为常量数据成员的初始化（声明、定义）：

  static const int x=10;

  但不能在构造函数中初始化

- 静态数据成员必须在体外定义、初始化

  static int x;

  .....

  int 类名:: x = 10;

- 在构造函数中，可以给静态数据成员赋值，但不称为初始化。

常量数据成员初始化

方法1： **const int x=10;**

方法2： **const int x;**

类名（参数）：**x(10) { ..... }** // 构造函数

5.4 静态函数成员

- 在函数的最前面加 static
- 静态函数成员的访问权限及继承规则同普通函数成员没有区别，可以缺省参数、省略参数以及重载。
- 普通函数成员的第一个参数是隐含this指针，指向调用该函数的对象；
- 静态函数成员没有this指针，若无参通常只应访问类的静态数据成员和静态函数成员。

5.4 静态函数成员

- stu1.Modify(100);

```
lea         ecx,[ebp-5Ch]
call        Student::Modify (8E1195h)
```

- stu2.Modify(87);

```
lea         ecx,[ebp-9Ch]
call        Student::Modify (8E1195h)
```

对于一般的函数成员只存放一次；与对象无关，

ecx 中存放对象指针，调用的函数地址相同

5.4 静态函数成员

- 静态函数成员的调用和静态数据成员类似
- 类名::静态函数成员(...);　　推荐使用
- 对象.静态函数成员(...);
- 对象.类名::静态函数成员(...)。

5.4 静态函数成员


```
class A{
    int x; //普通成员必须在对象(同this相关)存在时才能访问
    static int  I;
    static int  f( ){ return  x+ i; }; //错：static int  f( )无this则无x
public:
    static int  m( ) { return  i;  }; //静态函数成员无this
}a;
int A::i=0;        //私有的，体外定义并初始化
void main(void){
    int i=A::f( ); //错误，f私有的
    i=A::m( )+a.m( )+a.A::m( ); //正确，访问公有的静态函数成员
    i=a.f( );      //错误，f私有的不能访问
    i=a.A::f( );   //错误，f私有的不能访问
}
```

5.4 静态函数成员

- 不能使用**static**定义构造函数、析构函数以及虚函数。

```
class A {
   private:
        int x;
   public:
        static A( ) { x=0; };          //错误
        static virtual void f( ) { }  //错误
   };
```

构造函数、析构函数以及虚函数等必须有 **this** 参数

5.4 静态函数成员

- 常函数成员要说明隐含this参数。
- 常函数成员不能定义为没有this参数的静态函数成员

```
class A {
    static int f( )const      // 错误
        { return 1; }
};
```

- 联合union不能定义静态数据成员，但可以定义静态函数成员。

```
union A {
    int x,y;
    static void f( );
};
```

5.4 静态函数成员

静态数据成员：

空间分配在对象之外；

多对象共享

类内申明、类外定义和初始化

静态函数成员

无this参数

5.5 静态成员指针

- **静态成员指针**指向类的静态数据（函数）成员。
- 变量、数据成员、普通函数和函数成员的参数和返回值都可以定义成静态成员指针类型。
- 静态成员相当于具有访问权限的变量和函数，同普通变量和函数相比，除访问权限外没有区别，因此，静态成员指针和普通指针形式上也没有区别。
- 静态成员指针存放静态成员地址，普通成员指针存放普通成员偏移；静态成员指针可以移动，普通成员指针不能移动；静态成员指针可以转换类型，普通成员指针不能转换类型。

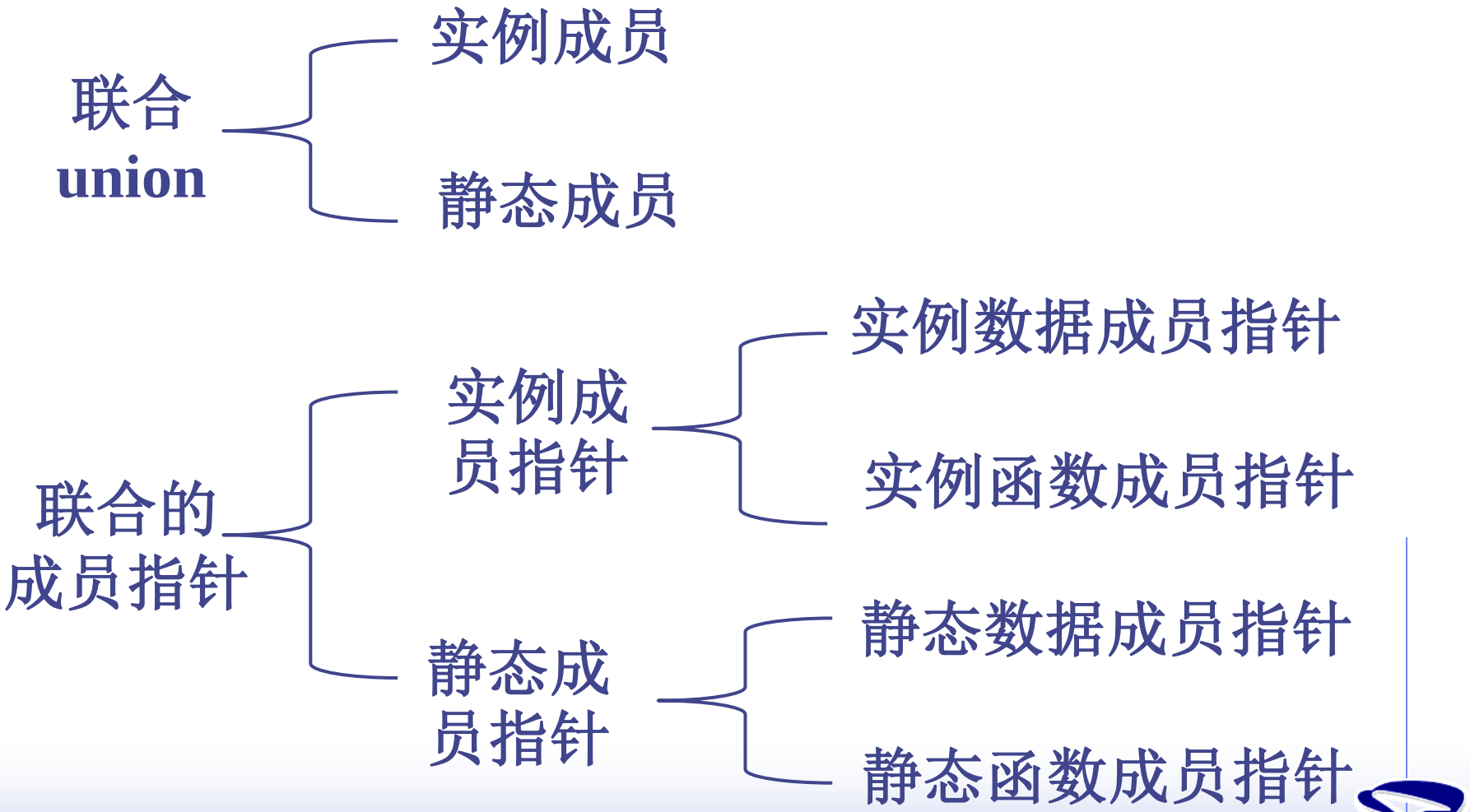
5.5 静态成员指针


```
class  P{
    char   name[20];
public:
    static int n;
    static int  getn( )  { return n; }
    P(char *n) { strcpy_s(name, n);    n++; }
    ~P( ) { n--; }
}zan((char *)"zan");
int P::n=0;
void main(void){
    int  m;
    int  *d=&P::n;          // d 为静态成员指针；等价于d=&zan.n;
                            //静态成员指针 与 普通指针 没有差别
    int (*f)( )=&P::getn;  //普通函数指针指向静态函数成员
    m=*d + (*f)( );
}
```

5.5 静态成员指针


```
struct A{
    int a, *b, A::*u, A::*A::*x, A::**y, *A::*z;
    static int c, A::*d;
};z;
int A::c=0, A::*A::d=&A::a;  //静态数据成员初始化
void main(void){     //注意优先级高低：  ".">"*">".*"
    int i, A::**m;
    z.a=5;        z.u=&A::a;  i=z.*z.u; //i=z.*&A::a=z.a=5
    z.x=&A::u; i=z.*(z.*z.x);
    //i=z.*(z.*&A::u)=z.*z.u=z.a=5
    m=&A::d;  m=&z.u;  i=z.**m; //i=z.**&z.u=z.*z.u
    z.y=&z.u;    i=z.**z.y; // i=z.**&z.u=z.*z.u=z.a=5
    z.b=&z.a;    z.z=&A::b;  i=*(z.*z.z);
    //i=*(z.*&A::b)=*z.b=*&z.a
}
```

5.6 联合的成员指针


联合
union
实例成员
静态成员
联合的
成员指针
实例成员
指针
静态成
员指针
实例数据成员指针
实例函数成员指针
静态数据成员指针
静态函数成员指针

5.6 联合的成员指针

- 联合可以定义实例和静态成员，故也可以定义实例和静态成员指针。
- 联合的实例数据成员共享内存，指向这些实例数据成员的指针存储的偏移量值实际上是相同的。
- 函数中局部类不能定义静态数据成员，
  函数中的局部联合也不能定义静态数据成员。
- 全局类中的联合或全局联合可以定义静态数据成员；静态数据成员指针一般指向全局类中的联合或全局联合的静态数据成员。

总结

实例数据成员、常数据成员、静态数据成员

实例函数成员、常函数成员、静态函数成员

实例成员指针、静态成员指针

const　　static　volatile　mutable

各是什么意义？　有何差别？　如何使用？

应用

在 QString 类中， 对于习惯传统用法的人提供静态函数的用法。

```
static  int compare(const QString &s1, const QString &s2,
                    Qt::CaseSensitivity cs = Qt::CaseSensitive)
//  比较 s1 与 s2

int compare(const QString &other, Qt::CaseSensitivity cs =
                Qt::CaseSensitive) const

// 本串  与 other 比较
```

练习

```
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; }
```