C++程序设计精要教程

华中科技大学

第5章 成员及成员指针

◆5.1 实例成员指针

- 成员指针：指向类的成员（普通和静态成员）的指针，分为**实例成员指针**和**静态成员指针**。变量、数据成员、函数参数和返回类型都可定义为成员指针类型，即普通指针能用的地方成员指针都能用
- **实例成员指针声明：**

```
int A::*p; //A类的成员指针，指向A类的int类型实例数据成员
```

- 实例成员指针使用：

```
struct A{
        int  m, n;
}a={1, 2}, b={3, 4};
p = &A::m; //实例成员指针p指向A类的实例数据成员m
int x = a.*p      //x = a.m; 利用实例成员指针访问成员 对象名.*指针名或对象指针->*指针名
```

第5章 成员及成员指针

◆5.1　实例成员指针

- 运算符.*和->*均为双目运算符，优先级均为第14级，结合性自左向右
- .*的左操作数为类的实例(对象)，右操作数为指向实例成员的指针
- ->*的左操作数为对象指针，右操作数为指向该对象实例成员的指针。
- 实例成员指针是指向实例成员的指针，可分为实例数据成员指针和实例函数成员指针。
- 实例成员指针必须直接或间接同.*或->*左边的实例(对象)结合，以便访问该对象的实例数据成员或函数成员。
- 构造函数不能被显式调用，故不能有指向构造函数的实例成员指针。
- 声明语法

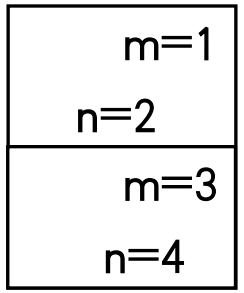
数据类型　类名::*指针名 = &类名::实例成员名

第5章 成员及成员指针

◆5.1 实例数据成员指针

- 实例数据成员指针是数据成员相对于对象首地址的偏移，不是真正的代表地址的指针。
- 实例数据成员指针不能移动：
  - 数据成员的大小及类型不一定相同，移动后指向的内存可能是某个成员的一部分，或者跨越两个(或以上)成员的内存；
  - 即使移动前后指向的成员的类型正好相同，这两个成员的访问权限也有可能不同，移动后可能出现越权访问问题。
- 实例成员指针不能转换类型：
- 否则便可以通过类型转换，间接实现实例成员指针移动

实例数据成员指针




```
struct A{  //实例数据成员指针是偏移量
   int  m, n;
}a={1, 2}, b={3, 4};  //假设int类型2个字节
void main (void)  {//以下p=0表示偏移，实现时实际<>0
   int x;
   int  A::*p=&A::m; //p是A的实例数据成员指针，指向整型成员m
   //设p=0：m相对结构体首址的偏移。编译后，偏移实际<>0
   x=a.*p;                    //x=* (a的地址+p) =* (2000+0) =1 = m
                              //a.*p指向相对对象a首地址偏移为p的成员，即m
   x=b.*p;                    //x=* (b的地址+p) =* (2004+0) =3
   p=&A::n;                   //p=2：n相对结构体首址偏移，实际<>2
   x=a.*p;                    //x=* (a的地址+p) =* (2000+2) =2
   x=b.*p;                    //x=* (b的地址+p) =* (2004+2) =4
}
```

第5章 成员及成员指针

【例5.2】本例说明实例成员指针不能移动。

```
#include <iostream.h>
struct A{
   int i;    //公有的成员i
private:
   long j;
public:
   int f( ){ cout<<"F\n";   return 1;  }
private:
   void  g( ){ cout<<"Function g\n";  }
}a;
```

第5章 成员及成员指针

```
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;              //等价于x=a.*(&A::i)=a.A::i=a.i
    x=(a.*pf)( );               //.*的优先级低，故用(a.*pf)
    pi++;                       //错误， pi不能移动，否则指向私有成员j
    pf+=1;                      //错误， pf不能移动
    x=(long) pi;                //错误，pi不能转换为长整型
    x=x+ sizeof(int)            //间接移动指针
    pi=(int A::*)x;             //错误，x不能转换为成员指针
}
```

实例成员指针不能和其它类型互相转换：否则便可以通过类型转换间接实现指针移动。

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- const只读，volatile易变，mutable机动；
- const和volatile可以定义变量、类的数据成员、函数成员及普通函数的参数和返回类型。
- mutable只能用来定义类的数据成员。
- 含const数据成员的类必须定义构造函数（如果没有类内就地初始），且数据成员必须在构造函数参数表之后，函数体之前初始化。
- 含volatile、mutable数据成员的类则不一定需要定义构造函数。

第5章 成员及成员指针

【例5.3】定义导师类，允许改名但不允许改性别。

```
#include <string.h>
#include <iostream.h>
class TUTOR{
    char          name[20];
    const         char  sex;          //性别为只读成员
    int   salary;
public:
    TUTOR(const char *name, const TUTOR *t);
    TUTOR(const char *name, char gender, int salary);
    const char *getname( ) { return name; }
    char *setname(const char *name);
};
```

第5章 成员及成员指针

```
TUTOR::TUTOR(const char *n, const TUTOR *t): sex(t->sex){
   strcpy(name,n);   salary=t->salary;
}//只读成员sex必须在构造函数体之前初始化
TUTOR::TUTOR(const char *n, char g, int s): sex(g),sarlary(s){
   strcpy(name,n);
}//非只读成员sarlary可在函数体前初始化，也可在构造函数体内再次赋值
char *TUTOR::setname(const char*n){
   return strcpy(name, n); //注意：strcpy的返回值为name
}
void main(void){
   TUTOR  wang("wang", 'F', 2000);
   TUTOR  yang("yang", &wang);
   *wang.getname( )= 'W' ;  //错误:不能改wang.getname( )返回的指针指向的字符(const char *)
   *yang.setname( "Zang" )= 'Y' ;// 可改wang.setname( )返回的指针指向的字符(char *)
}
```

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- 普通函数成员参数表后出现const或volatile，修饰隐含参数this指向的对象。出现const表示this指向的对象(其非静态数据成员)不能被函数修改，但可以修改this指向对象的非只读类型的静态数据成员。
- 构造或析构函数的this不能被说明为const或volatile的(即要构造或析构的对象应该能被修改，且状态要稳定不易变)。
- 对隐含参数的修饰还会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，否则编译程序会对函数调用发出警告。

第5章 成员及成员指针

【例5.4】参数表后出现const和volatile。

```
#include <iostream.h>
class A{
   int a;  const  int  b; //b为const成员或y引用时，只能在构造函数后的成员初始化列表初始化
public:
   int f( ){a++; //this类型为A * const  this, 指向的
         return a; //对象可修改(故其普通成员a可修改)，只读成员b不可改
   }
    int f( )const{//a++;        //this类型为const A * const  this，指向的对象
         return a;  //不可改，其普通成员a不可改。同上,b不可改
   }
   int f( )volatile { //this类型为volatile A * const this，指向的对象可修
        a++;        //改，其普通成员a可修改。同上，只读成员b不可改
        return a;
   }
```

第5章 成员及成员指针

```
    int f( )const volatile{      //this类型为const volatile A* const this,
        //a++;          //不能修改普通成员a。同上，只读成员b不可改
        return a;
    }
     A(int x) ： b(x) { a=x; }
} x(3);                          //等价于A x(3), x可修改,
const A y(6);                    // y、z不可改
const volatile A z(8);           // x、y、z由开工函数构造、收工函数析构
void main(void) {
    x.f( ); //普通对象x调用int f( ): 指向的对象可修改
    y.f( ); //只读对象y调用int f( )const:指向的对象不可修改
    z.f( ); //只读易变对象z调用int f( )const volatile
}
```

const、volatile和mutable

```
#include <string.h>
class TUTOR{
   char      name[20];
   const     char  gender;          //性别为只读成员
   int                wage;
   mutable int querytimes;
public:
   TUTOR (const char *n,  char g,  int w) :  gender (g),  wage (w) { strcpy (name,  n);  }

   //函数体不能修改当前对象，this指针类型变成const  TUTOR * const this;
   const char *getname ( ) const { return name;  }

   char *setname (const char *n)   volatile // this指针类型变成volatile TUTOR * const this;
            //由于this指针被volatile修饰，导致name类型为 voliatile char [20]
   {return strcpy ( (char *) name,  n)  ;  } //必须强制将volatile char * 转换为char *
  void query (char* &n,  char &s,  int &w)   const
   //函数体不能修改当前对象，但修改机动成员。&name[0]类型为const char *
   {  n= (char*) &name[0];  s=gender;  w=wage;
      wage ++;                        //错误，不能修改const对象的成员
     querytimes++;                    //ok，querytimes是mutable成员
   }
};
```

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- 函数成员参数表后出现volatile，常表示调用该函数成员的对象是挥发对象，这通常意味着存在并发执行的进程。

- 函数成员参数表后出现const时，不能修改调用对象的非静态数据成员，但如果该数据成员的存储类为mutable，则该数据成员就可以被修改。
- mutable说明数据成员为机动数据成员，该成员不能用const、volatile或static修饰。

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- 有址（左值）引用变量(&)只是被引用对象的别名，被引用对象自己负责构造和析构，该引用变量(逻辑上不分配内存的实体)不必构造和析构。
- 无址（右值）引用变量(&&)常用来引用常量对象或者生命周期即将结束的对象，该引用变量(逻辑上不分配缓存的实体)不必构造和析构。无址引用变量为左值，但若同时用const定义则不能出现在＝左边。
- 如果A类型的有址（左值）引用变量r引用了new生成的(一定有址的）对象x，则应使用delete &r析构x，同时释放其所占内存。
- 引用变量必须在定义的同时初始化，函数的引用参数则在调用函数时初始化。非const左值引用变量和参数必须用同类型的左值表达式初始化。

左值引用对象

```
class A{
        int i;
public:
        A(int i){ A::i= i; cout<<"A:i = "<<i<<"\n";}
        ~A(){if(i) cout<<"~A:i = "<<i<<"\n";i=0;}
};
void g(A &a){cout<<"g is running\n";}
void main(int argc, char* argv[]){
        A a(1),b(2);
        A &p = a; //p引用a，没有构造新对象
        A &q = *new A(3);
        A &r = p; //r引用a，没有构造新对象
        cout<<"call g(b)\n";
        g(b); // 实参传递给形参，A &a = b; 没有构造新对象
        cout<<"main return\n";
        delete &q; //必须delete
}
```

输出：

```
A:i=1
A:i=2
A:i=3
call g(b)
g is running
main return
~A:i=3
~A:i=2
~A:i=1
```

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- mutable仅用于说明实例数据成员为机动成员，不能用于静态数据成员的。
- 所谓机动是指在整个对象为只读状态时，其每个成员理论上都是不可写的，但若某个成员是mutable成员，则该成员在此状态是可写的。
- 例如，产品对象的信息在查询时应处于只读状态，但是其成员“查询次数”应在此状态可写，故可以定义为“机动”成员。
- 保留字mutable还可用于定义Lambda表达式的参数列表是否允许在Lambda的表达式内修改捕获的外部的参数列表的值。

第5章 成员及成员指针

◆5.2 const、volatile和mutable

```
class PRODUCT {
   char* name;                          //产品名称
   int  price;                          //产品价格
   int  quantity;                       //产品数量
   mutable int count;                   //产品查询次数
public:
   PRODUCT(const char* n, int m, int p);
   int buy(int money);
   void get(int& p, int& q)const;
   ~PRODUCT(void);
};
void PRODUCT::get(int &p, int &q)const{//const  PRODUCT *const this
   p=price;      q=quantity; //当前对象为const对象，故其成员不能被修改
   count++;                  //但count为mutable成员，可以修改
}
```

第5章 成员及成员指针

◆5.3 静态数据成员

静态成员用static声明，包括静态数据成员和静态函数成员，static声明只能出现在类内。

非const、非inline静态数据成员在类体内声明、类体外定义并初始化。

类的静态数据成员在类还没有实例化对象前就已存在，相当于Java的类变量，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。访问权限同普通成员。

逻辑上，所有对象共享静态数据成员内存，任何对象修改静态数据成员的值，都会同时影响其他对象关于该成员的值。物理上，静态数据成员相当于独立分配内存的变量，不属于任何对象内存的一部分。

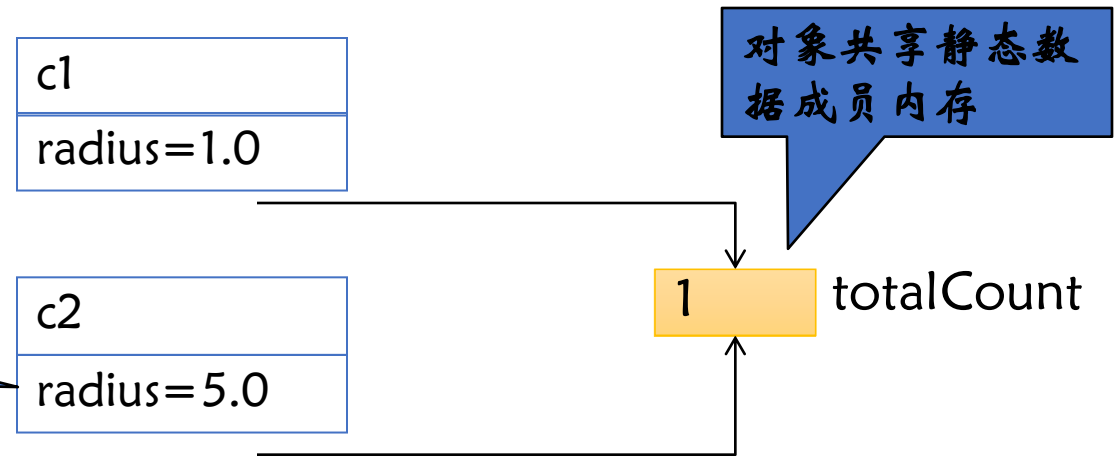
静态数据成员相当于有类名限定、带访问权限的全局变量

```
class Circle {
private:
    double radius;
    static int totalCount;
public:
    static int getTotalCount();
public:
    Circle():radius(1.0){Circle::totalCount++;}
    Circle(double r):radius(r){Circle::totalCount++;}
};
int Circle::totalCount = 0;

int Circle::getTotalCount(){
   return totalCount;
}

   Circle c1;
   Circle c2(5.0);
```

非const，非inline静态数据成员成员在类体内声明

非const静态数据成员成员在类体外定义并初始化

静态函数成员内只能访问静态成员，不能访问实例(普通)数据成员（radius）

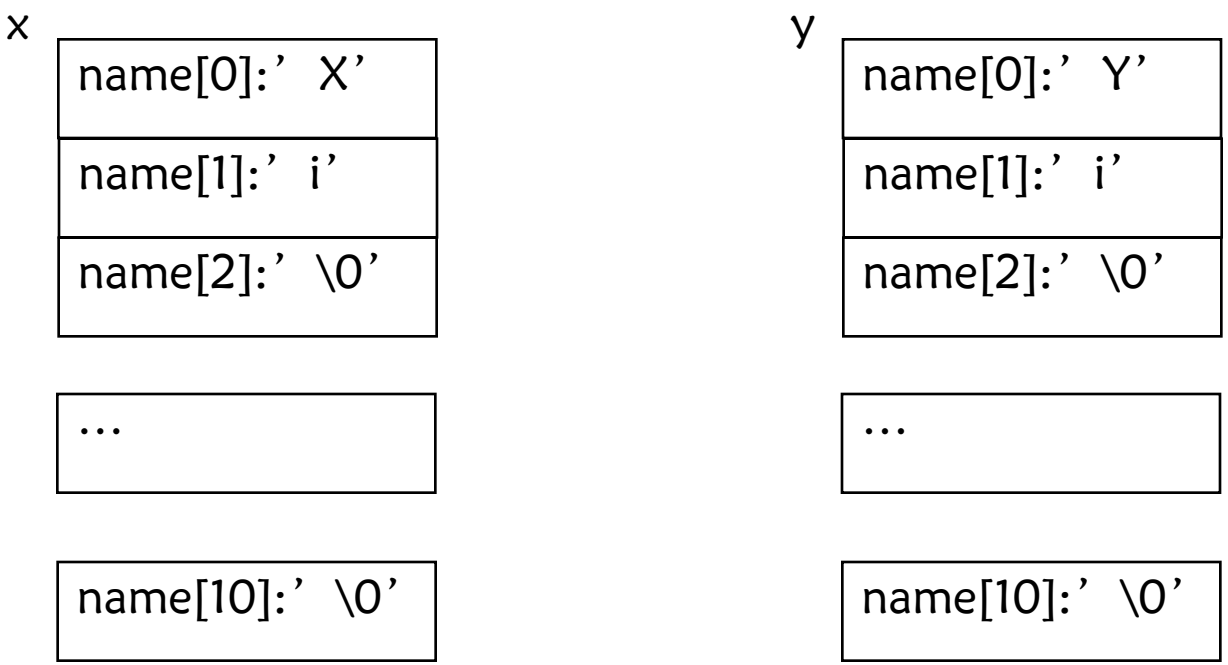


对象共享静态数据成员内存

实例数据成员属于对象内存布局的一部分，随着对象的存在而存在

```
#include <iostream.h>
#include <string.h>
class H {    //sizeof (H)=sizeof (char[11])，对象内存布局不包括静态成员total
   char name[11];
public:
   static int total;   //类体内声明，访问权限public
   H (char *n)  {
      strncpy (name, n, 10) ;
      H::total++;      //这里,H::可以省略,但不提倡
   }
   ~H ( )  { H::total --; };
};
int H::total=0;       //类体外定义并初始化，有访问权限的独立变量，
                      //注意这里不能加static，加了static使H::total局限于本文件
void main (void)  {  //sizeof (H)=sizeof x=sizeof y
           H  x ("Xi") ; //H::total=x.total=1
           H  y ("Yi") ;  //H::total=x.total=y.total=2
           cout<<H::total<<x.H::total<<x.total;//静态数据成员的三种访问方式
}
```

静态数据成员



静态数据成员相当于独立分配内存的变量，不属于任何对象内存的一部分

H::total = 2

此时

x.H::total = x.total = H::total = 2

y.H::total = y.total = H::total = 2

静态数据成员的初始化-非const、非inline成员

```
class A {
public:
        //非常量静态成员必须在类里声明，类外定义和初始化，静态成员的static声明只能出现在类里
        static int i = 0; //错误
};
```

error C2864: A::i: 带有类内初始化表达式的静态 数据成员 必须具有不可变的常量整型类型，或必须被指定为“内联”

静态数据成员的初始化-非const、非inline成员

```
class A {
public:
        //非常量静态成员必须在类里声明，类外定义和初始化，静态成员的static声明只能出现在类里
        static int i ; //类里声明
};

int A::i = 10; //类体外初始化，注意不能加static
```

静态数据成员的初始化-const成员

```
class B {
public:
        //const静态整型常量可以在类里初始化
        const static int i = 0;                    //正确
        const static int j{ 1 };                   //正确

        //错误。error C2864: B::d: 带有类内初始化表达式的静态数据成员必须具有不可变的常量整型类型，或必须被指定为“内联”
        //const static double d = 3.14 ;

        // constexpr类型的字面值类型可以在类体里初始化
        static constexpr double dd = 3.14;    //static constexpr double
};
```

静态数据成员的初始化-inline成员（17 标准）

```
class C {
public:
        //内联的静态数据成员可以在类里初始化
        inline static int i = 0;
        inline static double d = { 3.14 };

        //inline和const可以一起使用
        inline static const int j = 1;
};
```

第5章 成员及成员指针

◆5.3 静态数据成员

例5.9函数局部类不能定义静态数据成员

```
int  x=3;
union  S{ //定义全局类T
   const  static int b=0;      //全局类中可用const定义同时初始化静态整型成员,必须用常量
   inline  static int c = x;  //全局类可用inline定义同时初始化静态成员，可用任意表达式
   inline  const static int d = x;  //可用任意表达式
};
void f(void){
   class  T{                  //定义函数中的局部类T
      int  c;
     //static int d;          //错误：函数中的局部类不能定义静态数据成员
   };
   T a;                       //局部自动变量a
   static T s;                //局部静态变量s
}
void main( ){ f( ); f( ); }  //第一个函数调用f( )返回后a.d⇔s.d⇔T::d产生静态成员声明周期矛盾
```

第5章 成员及成员指针

◆5.4 静态函数成员

- 静态函数成员通常在类里以static说明或定义，它没有this参数。
- 有this的构造和析构函数、虚函数及纯虚函数都不能定义为静态函数成员。
- 静态函数成员一般用来访问类的总体信息，例如对象总个数。
- 静态函数成员可以重载、内联、定义默认值参数。
- 静态函数成员同实例成员的继承、访问规则没有区别。
- 静态函数成员的参数表后不能出现const、volatile、const volatile等修饰符。
- 静态函数成员的返回类型可以同时使用inline、const、volatile等修饰。

第5章 成员及成员指针

◆5.4 静态函数成员

```
class A{
   double  i;
   const static int  j=3;
public:
   static A& inc(A &);        //说明静态函数成员
   static A& dec(A &a){       //在内体内定义静态函数成员：自动inline
       a.i=a.i-A::j;
       return a;
    }
};
A& A::inc(A&a){               //不能定义static  A& A::inc(A&a)
   a.i+=A::j;                 //静态函数可访问静态数据成员，但不能访问实例成员this->i，但可以访问a.i（另外一个已构造好的对象）
   return a;
}
```

第5章 成员及成员指针

◆5.5 静态成员指针

- 静态成员指针是指向类的静态成员的指针，包括静态数据成员指针和静态函数成员指针。
- 静态数据成员的存储单元为该类所有的对象共享，因此，通过该指针修改成员的值时会影响到所有对象该成员的值。
- 静态数据成员和静态函数成员除了具有访问权限外，同普通变量/普通函数没有本质区别；静态成员指针则和普通指针没有任何区别。
- 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。

静态成员指针

静态成员指针（不管指向静态数据成员还是指向静态函数成员）就是普通指针，因此如果申明指向类的静态成员的指针，就用普通指针。

第5章 成员及成员指针

◆5.5 静态成员指针

【例5.15】定义群众类，使每个群众共享人数信息。

```
#include <iostream>
using namespace std;
class  CROWD{
   int    age;
   char   name[20];
public:
   static int number;
   static int getn( ) { return number;}
 CROWD(char *n, int a){
       strcpy(name,n);
       age=a;   number++;
   }
   ~CROWD( ) { number--; }
};
int CROWD::number=0;
void main(void){
   int *d=&CROWD::number; //普通指针指向静态数据成员
   int (*f)( )=&CROWD::getn; //普通函数指针指向静态函数成员
 //类CROWD无对象时访问静态成员
   cout<< "\nCrowd number=" <<*d; //输出0

   CROWD zan("zan", 20);
   //d=&zan.number; 等价于如下
   //d=&CROWD::number;
   cout<< "\nCrowd number=" <<*d;      //输出1
   CROWD tan("tan", 21);
   cout<< "\nCrowd number=" <<(*f)( );   //输出2
}
```

静态函数成员指针与普通函数成员指针的区别

- 对于一个全局函数，我们可以声明一个指向该函数的指针（普通函数指针），如

```
int sum( int x, int y) { return x + y;}
int (*pf)(int, int) = &sum; //&可省略
int s = pf(2,3); //通过普通函数指针调用函数，或int s = (*pf)(2,3)
```

- 对于类的静态函数，静态函数成员指针就是普通函数指针，如：

```
class A { public: static int sum(int x, int y);};
int A::sum(int x, int y) { return x + y;}
int  (*pf_A)(int,int) = &A::sum   //普通函数指针，&可省略
int s = pf_A(2,3); // 或 (*pf_A)(2,3);
```

静态函数成员指针与普通函数成员指针的区别

- 但指向类的普通函数成员的指针该怎么声明呢？如果用普通函数指针来声明会怎样？

```
class A { public:  int sum(int x, int y); };
int A::sum(int x, int y) { return x + y;}
int  (*pf_A)(int,int) = &A::sum  //这样可以吗？错误
```

- 因为类的普通函数成员除了函数返回类型及参数列表这两个重要特性外，还有第三个重要特性：所属的类类型。类的普通成员函数必通过一个对象来调用（this指针就是指向这个对象）。
- 而普通函数指针无法匹配类的普通函数成员第三个特征：类类型
- 所以类的普通函数成员必须用成员指针来指向。

```
int (A::*pf_A)(int,int) = &A::sum;
```

第5章 成员及成员指针

◆5.5 静态成员指针

- 静态成员指针与普通成员指针有很大区别。静态成员指针存放成员地址，普通成员指针存放成员偏移；静态成员指针可以移动，普通成员指针不能移动；静态成员指针可以强制转换类型，普通成员指针不能强制转换类型。

```
struct A{
    int a, *b;
    int A::*u;   int A::*A::*x;
    int A::**y; int *A::*z;
    static int c, A::*d;
}z;
int A::c=0;
int A::*A::d=&A::a;
void main(void){
    int i, A::**m;
    z.a=5;    z.u=&A::a;   i=z.*z.u = z.a;
    z.x=&A::u;      i=z.*(z.*z.x)=z.*z.u = z.a;
    m=&A::d;
    m=&z.u;         i=z.**m = z.*z.u = z.a;
    z.y=&z.u;       i=z.**z.y= z.*z.u = z.a;
    z.b=&z.a;
    z.z=&A::b;      i=*(z.*z.z) = *(z.b) = *(&z.a)=z.a;
}
```