华中科技大学计算机学院
李瑞轩

第五章静态成员与友元

本章内容：

- 5.1 静态数据成员
  5.2 静态函数成员
  5.3 静态成员指针
  5.4 成员友元
  5.5 普通友元

5.1 静态数据成员

- 静态成员包括静态数据成员和静态函数成员，其有关访问权限的规定和普通成员一样。
- 静态数据成员用于描述类的总体信息，必须在类的体外定义并初始化，union类不允许定义静态数据成员。
- 类的总体信息包括类的对象总数、连接所有对象的链表表头等。在开发基于窗口平台的应用软件时，可以通过存放窗口链表表头的静态数据成员维护窗口。

【例5.1】定义插入和删除节点分别由构造函数和析构函数自动完成的链表。

```
class  LIST{
    int   value;
    LIST *next;
    static LIST  *head;//声明静态数据成员
public:
    LIST(int value);
    ~LIST( );
};
LIST  *LIST::head=0;//定义并初始化静态数据成员
LIST::LIST (int v)
{
    value=v;
    next=head;
    head=this;
}
```

```
LIST::~LIST( )
{
    LIST *p= head;
    if(head==this) head=this->next;
    else {
        while(p->next!=this) p=p->next;
        p->next=this->next;
    }
}
```

5.1 静态数据成员

```
void main(void)
{
    LIST a(1);  //生成链首为节点a的链表
    LIST b(2);  //生成链首为节点b链尾为节点a的链表
    LIST c(3);   //生成链首为节点c链尾为节点a的链表
    b.~LIST( ); //从链表删除节点b
}
```

5.1 静态数据成员

- 静态数据成员脱离具体对象独立存在，其存储单元不是任何对象存储空间的一部分。所以，在计算对象或类的空间大小时不能包括静态数据成员。
- 静态数据成员独立分配一个存储单元，该存储单元为所有对象所共享，而不是像普通数据成员一样，为某个对象的一部分。
- 因此，对静态数据成员的访问不能通过this指针，而必须采用如下所示的三种形式。

5.1 静态数据成员

- ① 类名::静态数据成员名，如LIST::head；
- ②对象名.类名::静态数据成员名，如a.LIST::head；
- ③对象名. 静态数据成员名 如a.head。

5.1 静态数据成员

- 静态数据成员为所有对象所共享，因此，任何对象对静态数据成员的修改都会影响其它对象对该静态数据成员的访问值。
- 类中静态数据成员的说明为引用性说明,必须进行定义性说明,并且只能在类的体外进行唯一的一次说明. 注意在体外定义静态成员时,不需要static关键字

【例5.2】定义描述个人信息的类，使每个人都共享人口数量这一信息。

```
#include <iostream.h>
#include <string.h>
class HUMAN{
    char name[11];
    char sex;
    int  age;
public:
    static int total;
    HUMAN(char *n, char s, int a);
    ~HUMAN( );
};
int  HUMAN::total=0;
HUMAN::HUMAN(char *n, char s, int a)
{
    strncpy(name, n, 10);
    sex=s;
    age=a;
    HUMAN::total++;
}
HUMAN::~HUMAN( )
{
    HUMAN::total – –;
}
```

【例5.2】定义描述个人信息的类，使每个人都共享人口数量这一信息。

```
void main(void)
{
    cout<<"HUMAN::total="<<HUMAN::total<<"\n";
    cout<<"sizeof(int)="<<sizeof(int)<<"\n";
    HUMAN x("Xi", 'M', 20);
    cout<<"sizeof(x)="<<sizeof(x)<<"\n";
   cout<<"sizeof(HUMAN)="<<sizeof(HUMAN)<<"\n";
    cout<<"HUMAN::total="<<HUMAN::total;
    cout<<" x.total="<<x.total<<"\n";
    HUMAN y("Yi", 'F', 18);//同时改变x.total和y.total
    cout<<"HUMAN::total="<<HUMAN::total;
    cout<<" x.total="<<x.total;
    cout<<" y.total="<<y.total<<"\n";
}
```

- 输出：
- HUMAN::total=0
- sizeof(int)=4
- sizeof(x)=16
- sizeof(HUMAN)=16
- HUMAN::total=1 x.total=1
- HUMAN::total=2 x.total=2 y.total=2

5.1 静态数据成员

- 静态数据成员描述类的总体信息，由于全局类作用于所有程序文件，故全局类的静态数据成员也必须作用于所有程序文件。
- 联合union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，因此，静态数据成员不能成为联合的成员。

【例5.3】本例说明了全局类静态数据成员的作用域不能局限于程序文件。

```
class  P{
    int a;
    static in
    static i
    static c
public:
    void inc
    P(int x) { p+=x; };
};
int P::p=0;
static int P::q=0;
```

```
const int P::r=0;
void P::inc( )const  {
        a++;
        p++;
        r++;
}
union  UNTP{
        static   int  c;
        static   long d;
    };
void main(void) { }
```

//错误,不能修改只读对象的普通数据成员

//正确，可以修改只读对象的普通静态数据成员

//错误,不能修改只读静态数据成员

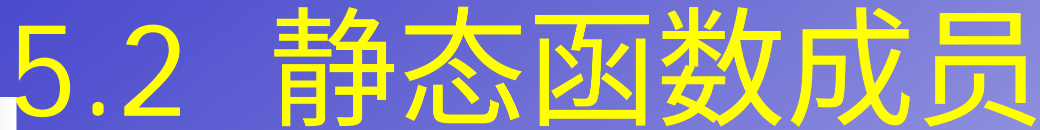
const说明调用inc的对象为只读对象

//错误,不能使用static

//错误，static声明p::q局限于程序文件

5.2 静态函数成员

- 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，同样，静态函数成员也可以缺省参数、省略参数以及进行重载。不同的是，普通函数成员的第一个参数为隐含this参数，而静态函数成员则没有隐含this参数。
- 调用静态函数成员同访问静态数据成员一样



5.2 静态函数成员

- 私有的和受保护的静态函数成员不能被普通函数main访问，除非main被定义为类A的友元函数。

【例5.4】本例说明了静态函数成员的用法。

```
class A{
    static int  i, f( );
protected:
    static int  g( );
public:
    static int  m( );
};
int A::i=0;
int A::f( ) { return  A::i; }
int A::g( ) { return A::f( ); }
```

```
void main(void)
{
    int  i=A::f( );
    i=A::g( );
    i=A::m( );
    A  a;
    i=a.f( );
    i=a.g( );
    i=a.m( );
    i=a.A::f( );
    i=a.A::g( );
    i=a.A::m( );
```

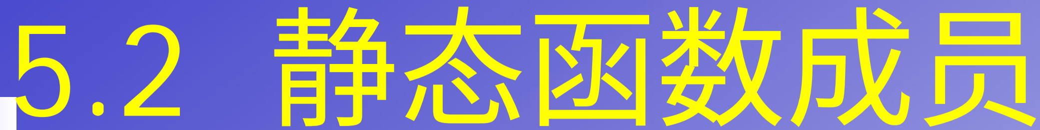
错误，main不能访问私有和保护函数成员

错误，main不能访问私有和保护函数成员

错误，main不能访问私有和保护函数成员

5.2 静态函数成员

- 构造函数、析构函数以及虚函数等都有this参数，此外，若函数成员的参数表后出现了const或volatile，则该函数成员的参数表必包含隐含的this参数，因此，这些函数不能定义为（没有this参数的）静态函数成员。
- 联合union虽然不能定义静态数据成员，却可以定义静态函数成员。



- 同全局类的静态数据成员一样，全局类的静态函数成员也必须作用于整个程序

5.3 静态成员指针

- 静态数据成员的存储单元为所有对象共享，对于共享该静态数据成员的所有对象来说，指向该静态成员的指针都应指向同样的地址。事实上，静态数据成员除了具有访问权限外，同普通变量没有本质上的区别，因此，静态成员指针类型和普通指针类型没有区别。
- 变量、数据成员、函数和函数成员的参数和返回值都可以定义为静态成员指针类型。

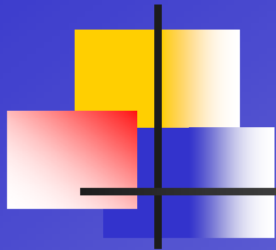
【例5.8】定义群众类，使每个群众共享人数信息。

```
#include <string.h>
#include <iostream.h>
class  CROWD{
    int    age;
    char   name[20];
public:
    static int number;
    static int getnumber( )
{ 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;
int (*f)( )=&CROWD::getnumber;
   cout<<"\nCrowd
number="<<*d;
   CROWD zan("zan", 20);
   cout<<"\nCrowd
number="<<*d;
   CROWD tan("tan", 21);
   cout<<"\nCrowd
   number="<<(*f)( );
   CROWD wan("wan", 21);
   cout<<"\nCrowd
   number="<<(*f)( );
}
```

5.3 静态成员指针

- 静态成员指针就是普通指针，较普通成员指针有很大差别。在计算具有指针变量的表达式时，要注意各类指针的优先级和结合性。



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

```
z.u=&A::a;
i=z.*z.u;
z.x=&A::u;
i=z.*(z.*z.x);
m=&z.u;
i=z.**m;
z.y=&z.u;
i=z.**z.y;
z.b=&z.a;
z.z=&A::b;
i=*(z.*z.z);
}
```

5.4 成员友元

- 友元函数（简称友元）不是定义该友元的类的函数成员，但是，它能像类的函数成员一样访问类的所有成员。
- 友元有普通友元和成员友元两种类型。
- 普通友元是指将普通函数定义为某个类的友元.
- 成员友元是指将一个类的函数成员定义为另一个类的友元。

5.4 成员友元

- 类型封装为访问类的成员提供了公共接口函数，但是，通过公共接口函数访问类的成员效率较低。如果将访问类的成员的函数声明为类的友元，就可以大大提高这些函数访问该类成员的效率。

```
class INTSET{
int *elems, card, maxcard;
public:
    INTSET(int maxcard);
    ~INTSET( ) {
        delete elems; };
    int getcard( ){
     return card; };
    int getelem(int i);
};
INTSET::INTSET(int max)
{
    elems=new int[maxcard=max];
    card=0;
}
int INTSET::getelem(int i) { return elems[i]; };
class REALSET{
    float  *elems;
    int  card, maxcard;
public:
   REALSET(INTSET &s);
    ~REALSET( ) { delete elems; }
};
REALSET::REALSET(INTSET &s){
elems=new float[maxcard=card=s.getcard( )];
    for(int i=0; i<card; i++)
        elems[i]=s.getelem(i);
}
```

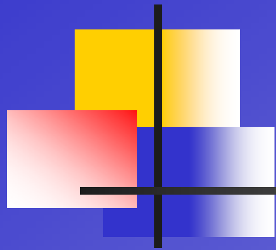
- 在用整型集合类INTSET的对象构造实型集合类REALSET的对象时，需要反复调用INTSET::getelem(int i) 公共接口函数访问INTSET的数据成员，这种函数调用的访问效率不高。
- 有两种办法可以提高访问效率：
- ①将INTSET::getelem(int i)定义为inline函数，②将构造函数REALSET(INTSET &)声明为类INTSET的成员友元。

5.4 成员友元

- 友元用friend声明。友元不是定义该友元的类的函数成员，故不受该类的访问权限的限制，因此，可随意在该类的private、protected或public下声明友元。
- 一个类的构造函数和析构函数也可以定义为另一个类的成员友元，由于构造函数和析构函数没有返回类型，故定义为另一个类的友元时可以随意指定它们的返回类型。C++将这一规定放宽到定义任何成员友元。
- 在定义成员友元时，可以同时定义成员友元的函数体，由于函数体是在类体中定义的，因此，该成员函数自动成为inline函数，且该成员函数的作用域局限于当前程序文件。

【例5.11】将REALSET的构造函数定义为INTSET的成员友元。

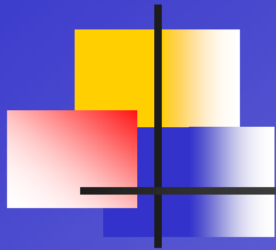
```
class INTSETclass REALSET{
    float *elems;
    int card, maxcard;
public:
    REALSET(INTSET &s);
    ~REALSET( ) { delete elems; }
};
class INTSET{
    int *elems, card, maxcard;
    friend double REALSET::REALSET(INTSET &)    {
elems=new float[maxcard=s.maxcard];
card=s.card;
for(int i=0; i<card; i++)
        elems[i]=s.elems[i];
}
public:
    INTSET(int maxcard);
    ~INTSET( ) {
        delete elems; };
    int getcard( ) {
        return card; };
    int getelem(int i) { return elems[i]; };
};
```



```
INTSET::INTSET(int max)
{
    elems=new int[maxcard=max];
    card=0;
}
void main(void)
{
    INTSET iset(20);
    REALSET rset(iset);
}
```

5.4 成员友元

- 在类A的函数成员f被声明为类B的成员友元后，还是由类A的对象而不是类B的对象调用f。若类A的静态函数成员被声明为类B的友元，则这种友元称为静态成员友元，静态成员友元的调用可以脱离具体对象。
- 如果一个类A被声明为类B的友元，则类A的所有成员函数都是类B的成员友元。



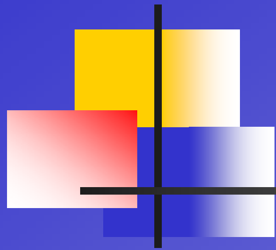
```
#include <iostream.h>
class B;
class A{
    int  i;
public:
    int set(B &);
    int get( ) { return i; };
    A(int x) { i=x; };
};
class B{
    int i;
public:

B(int x) { i=x; };
    friend A;        };
int A::set(B&b)
{
    return i=b.i;
}
void main(void)
{
    A a(1);
    B b(2);
    a.set(b);
cout<<"a.i="<<a.get( );
}
```

- 友元关系不能传递,即若类A是类B的友元,类B是类C的友元,此时,类A并不是类C的友元;
- 友元关系也不能互换,即类A是类B的友元,类B并不一定是类A的友元;

- 包括主函数main在内，任何普通函数都可以定义为类的普通友元。普通友元不是类的函数成员，故普通友元可在类的任何访问权限下定义。一个普通函数可以定义为多个类的普通友元。
- 像类自身的函数成员一样，普通友元可以访问类的任何数据成员和函数成员。

```
#include <iostream.h>
class INTSET;
class REALSET{
    float *elems;
    int card, maxcard;
public:
    REALSET(int maxcard);
    ~REALSET( ) {
        delete elems; };
 friend void Inttoreal(INTSET
       &, REALSET &);
};
REALSET::REALSET(int max)
{
 elems=new float[maxcard=max];
    card=0;
}
class INTSET{
    int *elems, card, maxcard;
    friend void main(void);
    friend void Inttoreal(INTSET &, REALSET &);
public:
    INTSET(int maxcard);
   ~INTSET( ) {
      delete elems; };
};
```



```
INTSET::INTSET(int max)
{   elems=new int[maxcard=max];
    card=0;
}
void Inttoreal(INTSET &s, REALSET &r)
{   int i, j;    j=r.card=s.card;
    for(i=0; i<j; i++)
        r.elems[i]=s.elems[i];
}
void main(void)
{
    INTSET  iset(20);
    REALSET rset(iset.maxcard); Inttoreal(iset,rset);}
```

5.5 普通友元

- 在重载函数中，未声明为友元的函数只能访问类的公有成员，只有声明为友元的函数才能访问类的所有成员。此外，同普通函数和函数成员一样，友元函数的参数也可以缺省和省略。

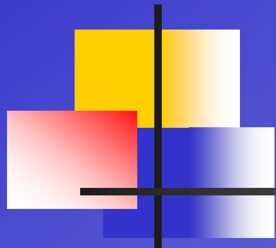
【例5.14】定义类SCORE，用于记录学生的姓名和成绩，并报告学生成绩是否优异。

```
#include <string.h>
#include <iostream.h>
class SCORE{
    int score;
public:
    char *name;
    friend void report(SCORE &);
    SCORE(char *name, int score);
    ~SCORE( ) { delete name; }
};
SCORE::SCORE(char *n, int s)
{
 strcpy(name=new
   char[strlen(n)+1],n);
  score=s;
}
```

```
void report(SCORE &s)
{    if(s.score>90)
         cout<<s.name<<" is
excellent\n";
}
void report(SCORE &s, int
excellent,int good=80)
{ if(s.score>excellent)
       cout<<s.name<<" is
excellent\n";
      if(s.score>good)
cout<<s.name<<" is
good\n";
}
void main(void)
{    SCORE z("Zang", 92);
     report(z);   }
```

5.5 普通友元

- static用于定义静态函数成员，virtual用于定义虚函数成员，因友元不是类的函数成员，故static、virtual不能和friend同时使用。



```
int m( ), n( );
class A{
    int s;
protected:
    static  int  h( );
    static  int  i( );
    virtual int  j( );
    virtual int  k( );
};
class B{
    int b;
protected:
    friend static int A::h( );
    friend int A::i( );

    friend virtual int A::j( );
    friend int A::k( );
    static friend int m( );
    virtual friend int n( );
};
```

//错误，不得使用virtual

//错误，不得使用static

//错误，不得使用static 和virtual

5.5 普通友元

- 任何函数的原型声明及其函数体的定义都可以分开，但一个函数的函数体只能定义一次。在定义普通友元时，也可以同时定义该函数的函数体。
- 在类体中定义函数体的普通友元缺省为内联函数，内联函数的存储类缺省为static，即内联函数的作用域仅局限于当前程序文件。由于全局main函数的存储类缺省为extern，因此，不能在类体中定义全局main的函数体。全局main可以定义为类的普通友元，且必须在类的体外定义main的函数体。

```
#include <iostream.h>
class A{
    int i;
public:
    friend int get(A &a) { return a.i; };
    A(int x) { i=x; };
};
void main(void)
{
    A a(1);
    cout<<"a.i="<<get(a);
}
```

【例5.17】本例展示了全局main定义为普通友元的方法。

程序文件1：

```
#include <iostream.h>
class A{
    int i;
    friend void main(void);
public:
    A(int x) { i=x; }
};
void main(void)
{   A a(5);
    cout<<"a.i="<<a.i;
   }
```

程序文件2：

```
#include <iostream.h>
class B{
    int i;
    friend void main(void)
   {
    cout<<"This static main\n";
    }
public:
    B(int x) { i=x; }
};
```