The C++ Programming Language

类的设计

陈辰
chenc@fudan.edu.cn
复旦大学软件学院

课程内容

- 自引用指针this ；
- 对象数组与对象指针；
- 向函数传递对象；
- 静态成员；
- 友元；
- 对象成员；
- 常类型；
- 多文件组成；

3.1自引用指针this （隐式）

例 3.1 this指针的引例

```
#include<iostream.h>
class A{
    public:
        A(int x1){ x=x1; }
        void disp(){ cout<<"x= "<<x<<endl;}
     private:
        int x;
};
main()
{   A a(1),b(2);
    cout<<" a: ";
    a.disp();
    cout<<" b: ";
     b.disp();
    return 0;
}
```

程序运行的结果：

a： x=1

b： x=2

3.1 自引用指针this

- 每一个类的成员函数都有一个隐藏定义的常量指针，我们把它称为this指针。
- this指针的类型就是成员函数所属的类的类型。
- 每当调用成员函数时，它被初始化为被调函数所在类的对象的地址。也就是自动地将对象的指针传给它。不同的对象调用同一个成员函数时，编译器将根据成员函数的this指针所指向的对象来确定应该引用哪一个对象的数据成员。
- 在通常情况下，this指针在系统中是隐含地存在的。也可以将其显式地表示出来。
- cout<<“x=”<<this->x<<endl;

Box类的volume计算体积函数为

```
float Box::volume() { return height * width * length; }
等价于
float Box::volume()
{ return this->height * this->width * this->length; }
```

例3.2 显示this指针的值。（显式）

```
#include<iostream.h>
class A{
  public:
     A(int x1){ x=x1; }
     void disp(){cout<<"\nthis="<<this<<"when x="<<this->x;}
  private:
      int x;
  };
main()
{   A a(1),b(2),c(3);
    a.disp();
    b.disp();
    c .disp();
     return 0;
}
```

```
this=0x0065FDF4    when x=1
this=0x0065FDF0    when x=2
this=0x0065FDEC    when x=3
```

【例3.3】分析下列程序的运行结果,说明程序中this和*this的用法

```
#include<iostream.h>
class Sample{
private:
     int x,y;
public:
     Sample(int i=0,int j=0)
        {  x=i;   y=j;   }
     void copy(Sample& xy);
     void print()
         { cout<<x<<","<<y<<endl;   }
};
void Sample::copy(Sample& xy)
{  if (this==&xy)   return;
    *this=xy;  }
void main()
{  Sample p1,p2(5,6);
    p1.copy(p2);
    p1.print();
}
```

运行结果：

5,6

例.一个使用this的类

```
class Circle
{
private:
double radius;
public:
Circle(double radius)  // 参数与数据成员同名时
{
this->radius = radius;  // 去掉 this 如何理解?
}                       // P272.例9.5中形式参数为何取 nam ?
double get_area()
{
return 3.14 * radius * radius;
}
};
int main()
{
Circle c = Circle(1);
double a = c.get_area();
cout << a << endl;
return 0;
}
```

- **说明:**

➢ this指针是一个const指针，不能在程序中修改它或给它赋值。

➢ this指针是一个局部数据，它的作用域仅在一个对象的内部。

（1）隐式存在：this指针是隐式使用的，它是作为参数被传递给成员函数的。

（2）自动实现：是编译系统自动实现的，程序设计者不必人为地在形参中增加this指针，也不必将对象a的地址传给this指针。

（3）*this：表示对象当前对象本身

例如，计算Box体积的函数可以改写为

```
return((*this).height * (*this).width * (*this).length);
```

3.2 对象数组与对象指针

对象数组

所谓对象数组是指每一数组元素都是对象的数组，也就是说，若一个类有若干个对象，则把这一系列的对象用一个数组来存放。对象数组的元素是对象，不仅具有数据成员，而且还有函数成员

定义一个一维数组的格式如下:

类名 数组名[下标表达式];

例如:

exam ob[4]; //定义了类exam的对象数组ob.

与基本数据类型的数组一样，在使用对象数组时也只能访问单个数组元素，也就是一个对象，通过这个对象，也可以访问到它的公有成员，一般形式是：

数组名[下标].成员名

例3.4 对象数组的应用

```
#include<iostream.h>
class exam{
   public:
      void set_x(int n){ x=n; }
       int get_x(){ return x; }
       private:
        int x;
    };
 main()
 {  exam ob[4];
     int i;
    for (i=0;i<4;i++)
        ob[i].set_x(i);
    for (i=0;i<4;i++)
         cout<<ob[i].get_x()<<' ';
    cout<<endl;
    return 0;
 }
```

0 1 2 3

如果需要建立某个类的对象数组，在设计类的构造函数时就要充分考虑到数组元素初始化的需要;

- 当各个元素的初值要求为相同的值时，应该在类中定义出不带参数的构造函数或带缺省参数值的构造函数;
- 当各元素对象的初值要求为不同的值时需要定义带形参(无缺省值)的构造函数。
- 定义对象数组时，可通过初始化表进行赋值。

定义对象数组时，可通过初始化表进行赋值

【例3.5】通过初始化表给对象数组赋值

```
#include<iostream.h>
class exam{
public:
    exam()
     { x=123;}
    exam(int n)
     { x=n;}
    int get_x()
     { return x; }
private:
 int x;
};
```

```
main()
 {  exam ob1[4]={11,22,33,44};
     exam ob2[4]={55,66};
      exam ob3[4]={exam(11),exam(22),exam(33),exam(44)};
     exam ob4[4]={exam(55),exam(66)};
     ob4[2]=exam(77);
     ob4[3]=exam(88);
     int i;
     for (i=0;i<4;i++)
         cout<<ob1[i].get_x()<<' ';
     cout<<endl;
     for (i=0;i<4;i++)
         cout<<ob2[i].get_x()<<' ';
     cout<<endl;
     for (i=0;i<4;i++)
        cout<<ob3[i].get_x()<<' ';
      cout<<endl;
      for (i=0;i<4;i++)
         cout<<ob4[i].get_x()<<' ';
     cout<<endl;
      return 0;
}
```

```
11 22 33  44
55 66 123 123
11 22 33 44
55 66 77 88
```

【例3.6】本例说明当构造函数具有一个以上的参数时，如何对二维数组对象进行初始化

```
#include <iostream.h>
class example{
public:
    example(int n, int m)
    {  x = n;
       y = m;
    }
    ~example()
      { cout<<"Destructor called.\n"; }
    int get_x()
      { return x;}
    int get_y()
      { return y;}
private:
    int x, y;
};
```

```
main()
{   example op[3][2]={
    example(1,2),example(3,4),
    example(5,6),example(7,8),
    example(9,10),example(11,12)    };
    int i;
    for (i=0;i<3;i++)
      { cout<<op[i][0].get_x()<<' ';
        cout<<op[i][0].get_y()<<"\n";
        cout<<op[i][1].get_x()<<' ';
        cout<<op[i][1].get_y()<<"\n";
      }
    cout<<"\n";
    return 0;
}
```

```
1  2
3  4
5  6
7  8
9  10
11 l2
Destructor called.
Destructor called.
Destructor called.
Destructor called.
Destructor called.
Destructor called.
```

对象指针

每一个对象在初始化后都会在内存中占有一定的空间。因此，即可以通过对象名访问一个对象，也可以通过对象地址来访问一个对象。对象指针就是用于存放对象地址的变量。声明对象指针的一般语法形式为:

类名* 对象指针名;

1.用指针访问单个对象成员

定义指针变量：Date *p,date1;

初始化：指向一个已创建的对象　p=&date1;

访问：用“->”操作符,只能访问该对象的公有成员。

p->setdate(2007,10,18);

例3.7对象指针的使用

```
#include<iostream.h>
class exe{
public:
    void set_a(int a){ x=a; }
    void show_a(){ cout<<x<<endl; }
private:
     int x;
    };
main()
{   exe ob,*p;     // 声明类exe的对象ob和类exe的对象指针p
    ob.set_a(2);
    ob.show_a();       // 利用对象名访问对象的成员
    p=&ob;             // 将对象ob的地址赋给对象指针p
    p->show_a();       // 利用对象指针访问对象的成员
    return 0;
}
```

```
2
2
```

2. 用对象指针访问对象数组

对象指针不仅能访问单个对象,也能访问对象数组.

例如:　　exe *p;　　//声明对象指针p

　　exe ob[2];　　//声明对象数组ob[2]

　　p=ob;　　//将对象数组的首地址赋给对象指针

例如将例4.7的main()改写为:

```
main()
{  exe ob[2],*p;
   ob[0].set_a(10);
    ob[1].set_a(20);
    p=ob;
    p->show_a();
    p++;
    p->show_a();
    return 0;
}
```

结果: 10 20

一般而言,当指针加1或减1时,它总是指向其基本类型中相邻的一个元素,对象指针也是如此. 本例中指针对象p加1时,指向下一个数组对象元素.

【例3.8】本例说明如何通过对象指针来访问对象数组，使程序以相反的顺序显示对象数组的内容

```
#include <iostream.h>
class example
{  public:
        example(int n, int m)
        { x=n;
          y=m;
        }
       int get_x()
         { return x;}
       int get_y()
         { return y;}
   private:
       int x,y;
};
```

```
main()
{    example op[4]={
       example(1,2),
       example(3,4),
       example(5,6),
       example(7,8)
     };
     int i;
     example *p;
     p=&op[3];                    // 取出最后一个数组元素的地址
     for (i=0;i<4;i++)
       {  cout<<p->get_x()<<' ';
          cout<<p->get_y()<<"\n";
          p--;                    // 指向前一个数组元素
        }
     cout<<"\n";
     return 0;
}
```

```
7  8
5  6
3  4
1  2
```

指向类的成员的指针

类的成员自身也是一些变量、函数或者对象等。因此，也可以直接将它们的地址存放到一个指针变量中，这样就可以使指针直接指向对象的成员，进而可以通过指针访问对象的成员。

可在类外定义指向类的成员的指针来控制对象的成员。

注意：

- 指向成员的指针只能访问公有数据成员和成员函数。
- 使用要先声明，再赋值，然后访问。

1.指向数据成员的指针

声明：类型说明符 类名:: *数据成员指针名；

赋值：数据成员指针名＝&类名:: 数据成员名

使用：对象名. *数据成员指针名

对象指针名-> *数据成员指针名

【例3.9】访问对象的公有数据成员的几种方式

```
#include<iostream.h>
class A{
public:
    A(int i)    {  z=i;  }
    int z;
};
void main()
  { A ob(5);
    A *pc1;          // 声明一个对象指针pc1
    pc1=&ob;                    //  给对象指针pc1赋值
    int A::*pc2;     // 声明一个数据成员指针pc2  // ①
    pc2=&A::z;     // 给数据成员指针pc2赋值    // ②
    cout<<ob.*pc2<<endl;        // 用成员指针pc2访问数据成员z
    cout<<pc1->*pc2<<endl;    // 用成员指针pc2访问数据成员z
    cout<<ob.z<<endl;           // 使用对象名访问数据成员z
  }
```

2．指向成员函数的指针

声明：类型说明符　(类名:: *指针名)(参数表);

赋值：成员函数指针名 = 类名::成员函数名;

使用：(对象名.*成员函数指针名)(参数表)；

（对象指针名 -> *成员函数指针名）(参数表);

【例3.10】访问对象的公有成员函数的几种方式

```
#include<iostream.h>
class Coord {
public:
      Coord(int a=0,int b=0)  { x=a; y=b; }
     int getx()  { return x; }
     int gety()  { return y; }
private:
    int x,y;  };
void main()
{  Coord op(5,6);
   Coord *pc1=&op;
   int (Coord::*pc_getx)();
   pc_getx=Coord::getx;
   cout<<pc1->getx()<<endl;
   cout<<op.getx()<<endl;
   cout<<(op.*pc_getx)()<<endl;
   cout<<(pc1->*pc_getx)()<<endl;
}
```

3.3 向函数传递对象

使用对象作为函数参数

对象可以作为参数传递给函数，其方法与传递其他类型的数据相同。在向函数传递对象时，是通过传值调用传递给函数的。因此，函数中对对象的任何修改均不影响调用该函数的对象本身。

例3.11　使用对象作为函数参数。

```
#include<iostream.h>
class aClass{
 public:
        aClass(int n) { i=n; }
        void set(int n){ i=n; }
        int get( ){ return i; }
private:
        int i; };
void sqr(aClass ob)
{ ob.set(ob.get()*ob.get());
   cout<<"copy of obj has i value of ";
   cout<<ob.get()<<"\n"; }
main()
{ aClass  obj(10);
   sqr(obj);
   cout<<"But, obj.i is unchanged in main:";
   cout<<obj.get( );    return 0;  }
```

copy of obj bas i value of 100
But，obj.i is unchanged in main：10

使用对象指针作为函数参数

对象指针可以作为函数的参数，使用对象指针作为函数参数可以实现传址调用，即可在被调用函数中改变调用函数的参数对象的值，实现函数之间的信息传递。同时使用对象指针实参仅将对象的地址值传给形参，而不进行副本的拷贝，这样可以提高运行效率，减少时空开销。

当函数的形参是对象指针时，调用函数的对应实参应该是某个对象的地址值。

例3.12 使用对象指针作为函数参数。

```
#include<iostream.h>
class aClass {
  public:
     aClass(int n) { i=n; }
     void set(int n){ i=n; }
      int get(){ return i;}
  private:
     int i; };
  void sqr(aClass  *ob)
    { ob->set(ob->get() * ob->get());
       cout<<"Copy of obj has i value of ";
       cout<<ob->get()<<"\n"; }
main()
 {  aClass obj(10);
    sqr(&obj);
   cout<<"Now, obj.i in main() has been changed :";
  cout<<obj.get() <<"\n";  return 0;
}
```

使用对象引用作为函数参数

使用对象引用作为函数参数不但具有用对象指针作函数参数的优点，而且用对象引用作函数参数将更简单、更直接。

例3.13 使用对象引用作为函数参数。

```
#include<iostream.h>
class aClass {
public:
   aClass(int n)   { i=n; }
   void set(int n)  { i=n; }
   int get()  { return i;}
 private:
   int i; };
void sqr(aClass& ob)
{   ob.set(ob.get() * ob.get());
     cout<<"Copy of obj has i value of ";
     cout<<ob.get()<<"\n"; }
main()
 { aClass obj(10);
    sqr(obj);
    cout<<"Now, obj.i in main() has been changed :";
    cout<<obj.get() <<"\n";
    return 0;   }
```

3.4. 静态成员

引入目的：实现一个类的不同对象之间数据和函数共享

谁共享：本类所有对象。

- 静态数据成员
  - 用关键字static声明
  - 该类的所有对象维护该成员的同一个拷贝
  - 必须在类外定义和初始化，用(::)来指明所属的类。
- 静态成员函数
  - 类外代码可以使用类名和作用域操作符来调用公有静态成员函数。
  - 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。访问非静态数据成员，必须通过参数传递方式得到对象名，通过对象名访问。

静态数据成员

- 为何需要静态数据成员？

例 静态数据成员的引例

```
#include<iostream.h>
#include<string.h>
class Student {
    public:
        Student(char *name1,char *stu_no1,float score1);
        ~Student();
        void show();                          // 输出姓名、学号和成绩
        void show_count_sum_ave();   // 输出学生人数
    private:
        char *name;                          // 学生姓名
        char *stu_no;                      // 学生学号
        float score;                       // 学生成绩
         int count;
          float sum;
          float ave;   };
```

```
Student::Student(char *name1,char *stu_no1,float score1 )
{  name=new char[strlen(name1)+1];
    strcpy(name,name1);
    stu_no=new char[strlen(stu_no1)+1];
    strcpy(stu_no,stu_no1);
    score=score1;
    ++count;                    // 累加学生人数
    sum=sum+score;              // 累加总成绩
    ave=sum/count;              // 计算平均成绩
}
Student::~Student()
{   delete []name;
     delete []stu_no;
     --count;
     sum=sum-score;
}
```

```
void Student::show()
{   cout<<"\n name: "<<name;
     cout<<"\n stu_no: "<<stu_no;
     cout<<"\n score: "<<score; }
void Student::show_count_sum_ave()
{       cout<<"\n count: "<<count;
        cout<<"\n sum: "<<sum;
        cout<<"\n ave: "<<ave;   }
void main()
{   Student stu1("Liming","990201",90);
     stu1.show();
     stu1.show_count_sum_ave();
      Student stu2("Zhanghao","990202",85);
      stu2.show();
     stu2.show_count_sum_ave();    }
```

■如何定义静态数据成员？

在一个类中,若将一个数据成员说明为static,这种成员称为静态数据成员。

■静态数据成员特点？

与一般的数据成员不同,无论建立多少个类的对象,都只有一个静态数据的拷贝。从而实现了同一个类的不同对象之间的数据共享。它不因对象的建立而产生，也不因对象的析构而删除。它是类定义的一部分，所以使用静态数据成员不会破坏类的隐蔽性。

在类的声明中只能声明静态数据成员的存在。由于类的声明是抽象的，静态数据成员的初始化需要在类的外部进行，通过类名对它进行访问。

静态数据成员初始化的格式如下：

<数据类型><类名>::<静态数据成员名>=<值>;

- 初始化时不加该成员的访问权限控制符private，public等。
- 初始化时使用作用域运算符来标明它所属的类，因此，静态数据成员是类的成员，而不是对象的成员。

引用静态数据成员时，采用如下格式:

- <类名> :: 〈静态成员名>
- 如果静态数据成员的访问权限允许的话（即public的成员），可在程序中，按上述格式来引用静态数据成员。

例3.15静态数据成员的使用引例。

```
#include<iostream.h>
#include<string.h>
class Student {
    public:
        Student(char *name1,char *stu_no1,float score1);
        ~Student();
        void show();          // 输出姓名、学号和成绩
        void show_count_sum_ave();  // 输出学生人数
private:
        char *name;           // 学生姓名
        char *stu_no;         // 学生学号
        float score;          // 学生成绩
        static int count;     // 静态数据成员,统计学生人数
        static float sum;     // 静态数据成员,统计总成绩
        static float ave;     // 静态数据成员,统计平均成绩
};
```

```
Student::Student(char *name1,char *stu_no1,float score1 )
{  name=new char[strlen(name1)+1];
   strcpy(name,name1);
    stu_no=new char[strlen(stu_no1)+1];
    strcpy(stu_no,stu_no1);
    score=score1;
    ++count;                    // 累加学生人数
    sum=sum+score;              // 累加总成绩
    ave=sum/count;              // 计算平均成绩
}
Student::~Student()
{   delete []name;
    delete []stu_no;
    --count;
    sum=sum-score;
}
```

```
void Student::show()
{   cout<<"\n name: "<<name;
    cout<<"\n stu_no: "<<stu_no;
    cout<<"\n score: "<<score; }
void Student::show_count_sum_ave()
{
    cout<<"\n count: "<<count;   // 输出静态数据成员count
    cout<<"\n sum: "<<sum;       // 输出静态数据成员sum
    cout<<"\n ave: "<<ave;       // 输出静态数据成员ave
}
int Student::count=0;                // 静态数员count初始化
float Student::sum=0.0;              // 静态数员sum初始化
float Student::ave=0.0;              // 静态数员ave初始化
```

```
void main()
{
    Student stu1("Liming","990201",90);
    stu1.show();
    stu1.show_count_sum_ave();
    Student stu2("Zhanghao","990202",85);
    stu2.show();
    stu2.show_count_sum_ave();
}
```

如何使用静态数据成员?

(1) 静态数据成员的定义与一般数据成员相似，但前面要加上static关键词。

static 数据类型 数据成员名;

(2) 静态数据成员的初始化与一般数据成员不同。

初始化格式: 数据类型 类名::静态数据成员=值;

初始化位置: 定义对象之前，一般在类定义后，main()前进行。

(3) 访问方式(只能访问公有静态数据成员)

可用类名访问: 类名::静态数据成员

也可用对象访问: 对象名.静态数据成员

对象指针->静态数据成员

【例3.16】公有静态数据成员的访问

```
#include<iostream.h>
class myclass
{ public:
    static int i;
    int geti()
     { return i;}
};
int myclass∷i=0; // 初始化,不必在前面加static
main()
{ myclass∷i=200;
  myclass ob1,ob2;
  cout<<"ob1.i="<<ob1.geti()<<endl;
  cout<<"ob2.i="<<ob2.geti()<<endl;
  ob1.i=300;  // 对象进行访问
  cout<<"ob1.i="<<ob1.geti()<<endl;
  cout<<"ob2.i="<<ob2.geti()<<endl;
  return 0;
}
```

- (4)私有静态数据成员不能被类外部函数访问，也不能用对象进行访问。
- (5)支持静态数据成员的一个主要原因是可以不必使用全局变量。静态数据成员的主要用途是定义类的各个对象所公用的数据。

【例3.17】静态数据成员和一般数据成员的不同

```
#include<iostream.h>
class Student{
public:
    Student()    // 构造函数
    {  ++count;              // 每创建一个学生对象，学生数加1
       StudentNo=count;   // 给当前学生的学号赋值
    }
    void print()             // 成员函数，显示学生的学号和当前学生数
    {  cout<<"Student"<<StudentNo<<" ";
       cout<<"count="<<count<<endl;
    }
 private:
     static int count;   // 静态数据成员count，统计学生的总数
     int StudentNo;    // 普通数据成员，用于表示每个学生的学号
  };
int Student::count=0;
```

```
main()
   {   Student Student1;
       Student1.print();
       cout<<"----------\n";
       Student Student2;
       Student1.print();
       Student2.print();
       cout<<"----------\n";
       Student Student3;
       Student1.print();
       Student2.print();
       Student3.print();
       cout<<"----------\n";
       Student Student4;
       Student1.print();
       Student2.print();
       Student3.print();
       Student4.print();
       return 0;}
```

执行结果为：

Student1 count=1

Student1 count=2
Student2 count=2

Student1 count=3
Student2 count=3
Student3 count=3

Student1 count=4
Student2 count=4
Student3 count=4
Student4 count=4

静态成员函数

- 可以通过定义和使用**静态成员函数**来访问静态数据成员。
- 所谓静态成员函数就是**使用static关键字声明函数成员**。同静态数据成员一样，静态成员函数也属整个类，由同一个类的所有对象共同维护，为这些对象所共享。
- 静态成员函数作为成员函数，它的访问属性可以受到类的严格控制。对公有静态成员函数，可以通过类名或对象名来调用；而一般的非静态公有成员函数只能通过对象名来调用。
- 静态成员函数可以直接访问该类的静态数据成员和函数成员；而访问非静态数据成员，必须通过参数传递方式得到对象名，然后通过对象名来访问。

静态成员函数

定义静态成员函数的格式如下:

static 返回类型 静态成员函数名（参数表）;

在对象建立之前处理静态数据成员或全局变量

与静态数据成员类似，使用公有静态成员函数的一般格式有如下几种:

类名::静态成员函数名(实参表)

对象. 静态成员函数名(实参表)

对象指针->静态成员函数名(实参表)

例3.18静态成员函数来访问静态数据成员

```
#include<iostream.h>
#include<string.h>
class Student {
    private:
        char *name;
          char *stu_no;
          float score;
          static int count;
         static float sum;
    public:
        Student(char *name1,char *stu_no1,float score1);
           ~Student();
        void show();
        static void show_count_sum();
};
```

```
Student::Student(char *name1,char *stu_no1,float score1 )
{      name=new char[strlen(name1)+1];
       strcpy(name,name1);
       stu_no=new char[strlen(stu_no1)+1];
      strcpy(stu_no,stu_no1);
      score=score1;
      ++count;                    // 累加学生人数
      sum=sum+score;              // 累加总成绩
}
Student::~Student()
{
      delete []name;
      delete []stu_no;
      --count;
      sum=sum-score;    }
```

```
void Student::show()
{
        cout<<"\n name: "<<name;
        cout<<"\n stu_no: "<<stu_no;
        cout<<"\n score: "<<score;
}
void Student::show_count_sum()      // 静态成员函数
{
        cout<<"\n count: "<<count;    // 输出静态数据成员
       cout<<"\n sum: "<<sum;        // 输出静态数据成员
}
int Student::count=0;               // 静态数据成员初始化
float Student::sum=0.0;             // 静态数据成员初始化
```

```
void main()
{  Student stu1("Liming","990201",90);
   stu1.show();
   Student::show_count_sum();  // 使用类名访问静态成员函数
   Student stu2("Zhanghao","990202",85);
   stu2.show();
   stu2.show_count_sum();  // 使用对象stu2访问静态成员函数
    }
```

使用静态成员函数时须注意：

(1)静态成员函数可以定义成内嵌的，也可以在类外定义，在类外定义时不能用static前缀。

(2)静态成员函数主要用来访问全局变量或同一个类中的静态数据成员。特别是，当它与静态数据成员一起使用时，达到了对同一个类中对象之间共享数据进行维护的目的。

(3)私有静态成员函数不能被类外部函数和对象访问。

(4)使用静态成员函数的一个原因是，可以用它在建立任何对象之前处理静态数据成员。这是普通成员函数不能实现的。

(5)静态成员函数中没有指针this，所以静态成员函数不访问类中的非静态数据成员，若确实需要则只能通过对象名（作为参数）访问。

【例3.19】静态成员函数访问非静态数据成员

```
#include<iostream.h>
class small_cat {
  public:
    small_cat(double w)
    { weight=w;
      total_weight+=w;
      total_number++;  }
     static  void display(small_cat& w)//访问非静态成员
       { cout<<"The small_cat weights  "<<w.weight<<" kg\n";}
    static void total_disp() //访问静态成员
       { cout<<total_number<<" small_cat total_weight ";
            cout<<total_weight<<"kg "<<endl;    }
  private:
    double weight;
    static double total_weight;
    static double total_number;
};
double small_cat::total_weight=0;
double small_cat::total_number=0;
```

```
int main()
{   small_cat w1(0.9)，w2(0.8)，w3(0.7);
    small_cat::display(w1);
    small_cat::display(w2);
    small_cat::display(w3);
    small_cat::total_disp();
    return 0;
}
```

通过普通指针访问静态成员

例3.20 通过指针访问类的静态数据成员。

```
#include<iostream.h>
class myclass {
   public:
      myclass()                          // 构造函数,每定义一个对象,
        { ++i; }                                  静态数据成员i加1
      static int i;                       // 声明静态数据成员i
         };
 int myclass::i=0;                       // 静态数据成员i初始化,
                                               不必在前面加static
 main()
 {  int *count=&myclass::i;
    myclass ob1,ob2,ob3,ob4;
    cout<<“ myclass::i= ”<<*count<<endl;
    return 0;
}
```

【例3.21】通过指针访问类的静态成员函数

```
#include<iostream.h>
class myclass {
public:
    myclass()
      {   ++i;  }
    static int geti()
      {   return i;  }
private:
    static int i;
  };
int myclass::i=0;
main()
{  int (*get)()=myclass::geti;
   myclass ob1,ob2,ob3;
   cout<<"myclass∶∶i="<<(*get)()<<endl;
   return 0;
}
```

例A 通过指针访问类的静态数据成员

```
#include <iostream>
using namespace std;
class Point {  //Point类定义
public:                          //外部接口
    Point(int x = 0, int y = 0) : x(x), y(y) {
          count++;}
    Point(const Point &p) : x(p.x), y(p.y) {
          count++;}
    ~Point() {  count--; }
    int getX() const { return x; }
    int getY() const { return y; }
    static int count;
private:          //私有数据成员
    int x, y;
};
int Point::count = 0;
```

```
int main() {   //主函数实现
    //定义一个int型指针，指向类的静态成员
    int *ptr = &Point::count;
    Point a(4, 5);        //定义对象a
    cout << "Point A: " << a.getX() << ", " << a.getY();
    cout << " Object count = " << *ptr << endl;
    Point b(a); //定义对象b
    cout << "Point B: " << b.getX() << ", " << b.getY();
    cout << " Object count = " << *ptr << endl;
    return 0;
}
```

例B　通过指针访问类的静态函数成员

```
#include <iostream>
using namespace std;
class Point {       //Point类定义
public:             //外部接口
    Point(int x = 0, int y = 0) : x(x), y(y) {count++; }
    Point(const Point &p) : x(p.x), y(p.y) {count++;}
    ~Point() {count--;}
    int getX() const {return x;}
    int getY() const {return y;}
    static void showCount(){
    cout << "Object count = " << count << endl;}
private: //私有数据成员
    int x, y;
    static int count;
};
int Point::count = 0;
```

```
int main() {   //主函数实现
    //定义一个指向函数的指针，指向类的静态成员函数
    void (*funcPtr)() = Point::showCount;
    Point a(4, 5);        //定义对象A
    cout << "Point A: " << a.getX() << ", " << a.getY();
    funcPtr();  //输出对象个数，直接通过指针访问静态函数成员
    Point b(a); //定义对象B
    cout << "Point B: " << b.getX() << ", " << b.getY();
    funcPtr();  //输出对象个数，直接通过指针访问静态函数成员
    return 0;
}
```

3.5 友元

在一个类中可以有公用的(public)成员和私有的(private)成员。在类外可以访问公用成员，只有本类中的函数可以访问本类的私有成员。现在，我们来补充介绍一个例外——友元(friend)。

友元可以访问与其有好友关系的类中的私有成员。友元包括友元函数和友元类。

友元既可以是不属于任何类的一般函数,也可以是另一个类的成员函数,还可以是整个的类。

友元函数

友元函数不是当前类的成员函数，而是独立于当前类的外部函数，但它可以访问该类的所有对象的成员，包括私有、保护和公有成员。

友元函数的声明：

位置：当前类体中.

格式：函数名前加friend

友元函数的定义：

类体外：同一般函数（函数名前不能加　类名::）

类体内：函数名前加friend

友元函数没有this指针，它是通过入口参数（该类对象）来访问对象的成员的。

Friend functions友元函数

```
// friend functions
#include <iostream>
using namespace std;
class Rectangle {
        int width, height;
public:
        Rectangle() {}
        Rectangle (int x, int y) : width(x), height(y) {}
        int area() {return width * height;}
        friend Rectangle duplicate (const Rectangle&);
    };
```

Friend functions友元函数

```
Rectangle duplicate (const Rectangle& param) {
        Rectangle res;
        res.width = param.width*2;
        res.height = param.height*2;
return res;
}
int main () {
        Rectangle foo;
        Rectangle bar (2,3);
        foo = duplicate (bar);
        cout << foo.area() << '\n';
        return 0;
}
```

例. 将普通函数声明为友元函数

```
#include <iostream>
using namespace std;
class Time
{  public:
Time(int,int,int);
friend void display(Time &);   //声明display函数为Time类的友元函数
private:                       //以下数据是私有数据成员
int hour;
int minute;
int sec;
};
Time ::Time(int h,int m,int s)      //构造函数，给hour,minute,sec赋初值
{  hour=h;
```

```
minute=m;
sec=s;
}
void display(Time& t)
{  cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;
}
int main( )
{  Time t1(10,13,56);
display(t1);
return 0;
}
```

程序输出结果如下:

10:13:56

例3.22 友元函数的使用。

```
#include<iostream.h>
#include<string.h>
class girl {
public:
    girl(char *n,int d)
      {   name=new char[strlen(n)+1];
          strcpy(name,n);  age=d; }
    friend void disp(girl &);    //声明友元函数
    ~girl(){ delete []name; }
 private:
     char *name;  int age;
 };
 void disp(girl &x)              //定义友元函数
  {  cout<<"Girl\'s name is "<<x.name<<",age:"<<x.age<<"\n"; }
 void main()
 {   girl e("Chen Xingwei",18);
    disp(e);                    // 调用友元函数
 }
```

说明

- (1)友元函数虽然可以访问该类的私有成员，但它毕竟不是成员函数，因此，在类的外部定义友元函数时，不能在函数名前加上“类名::”。
- (2)友元函数一般带有一个该类的入口参数。因为友元函数不是类的成员函数，它没有this指针，所以它不能直接引用对象成员的名字，也不能通过this指针引用对象的成员，它必须通过作为入口参数传递进来的对象名或对象指针来引用该对象的成员。

引入友元机制的原因

- (1)友元机制是对类的封装机制的补充，利用此机制，一个类可以赋予某些函数访问它的私有成员的特权。
- (2)友元提供了不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。

友元函数可以是多个类的:

```
#include<iostream.h>
#include<string.h>
class boy;                    //向前引用
class girl{
  public:
      void init(char N[],int A);
      friend void prdata(const girl plg ,const boy plb);
  private:
      char name[25];
      int age;
};
  void girl :: init(char N[],int A)
   {  strcpy(name，N);
       age=A;      }
```

```
class boy {
public:
    void init(char N[],int A);
     friend void prdata(const girl plg,const boy plb);
private:
     char name[25];
     int age;
};
void boy :: init(char N[],int A)
{   strcpy(name,N);
     age=A;          }
void prdata(const girl plg,const boy plb)
{   cout<<"name:  "<<plg.name<<"\n";
    cout<<"age:   "<<plg.age<<"\n";
    cout<<"name:  "<<plb.name<<"\n";
    cout<<"age:   "<<plb.age<<"\n";
}
```

```
main()
{  girl G1,G2,G3;
    boy B1,B2,B3;
    G1.init("Zhang Hao",12);
    G2.init("Li Ying",13);
    G3.init("Wan Hong",12);
    B1.init("Chen Lin",11);
    B2.init("Wang Hua",13);
    B3.init("Bai Xiu",12);
    prdata(G1,B1);                    //调用友元函数prdata()
    prdata(G2,B2);          //调用友元函数prdata()
    prdata(G3,B3);                    //调用友元函数prdata()
    return 0;
}
```

友元成员函数不仅可以是一般函数(非成员函数)，而且可以是另一个类中的成员函数:

例:友元成员函数的简单应用。

在本例中还将用到类的提前引用声明，请注意。

```
#include <iostream>
using namespace std;
class Date;                    //对Date类的提前引用声明
class Time                     //定义Time类
{  public:
Time(int,int,int);
void display(Date &);    //display是成员函数，形参是Date类对象的引用
private:
 int hour;
```

```
    int minute;
    int sec;
    };
class Date                          //声明Date类
{  public:
Date(int,int,int);
friend void Time::display(Date &);
private:
    int month;
    int day;
    int year;
    };
Time::Time(int h,int m,int s)    //类Time的构造函数
{   hour=h;
    minute=m;
    sec=s;
    }
```

```
void Time :: display(Date &d)
{    cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl;
     cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
Date :: Date(int m,in  d,int y)          //类Date的构造函数
{   month=m;
    day=d;
    year=y;
}
int main( )
{   Time t1(10,13,56);              //定义Time类对象t1
    Date d1(12,25,2004);            //定义Date类对象d1
t1.display(d1);              //调用t1中的display函数，实参是Date类对象d1
    return 0;
}
```

友元成员

一个类的成员函数也可以作为另一个类的友元,这种成员函数不仅可以访问自己所在类对象中的所有成员,还可以访问friend声明语句所在类对象中的所有成员。

这样能使两个类相互合作、协调工作,完成某一任务。

一个类的成员函数作为另一个类的友元函数时，必须先定义这个类。

例3.24 一个类的成员函数作为另一个类的友元。

```
#include<iostream.h>
#include<string.h>
class girl;
class boy {
  public:
      boy(char *n,int d)
        {   name=new char[strlen(n)+1];
            strcpy(name,n); age=d; }
      void disp(girl &);        // 声明disp()为类boy的成员函数
      ~boy(){ delete name;  }
  private:
      char *name;
      int age;
};
```

```
class girl {
    public:
        girl(char *n,int d)
        {   name=new char[strlen(n)+1];
            strcpy(name,n); age=d; }
        friend void boy::disp(girl &);
        ~girl(){ delete name; }
    private:
        char *name;
        int age; };
    void boy::disp(girl &x)
    {  cout<<"Boy\'s name is "<<name<<",age:"<<age<<"\n";
        cout<<"Girl\'s name is "<<x.name<<",age:"<<x.age<<"\n";
     }
```

```
void main()
 {  boy b("Chen Hao",25);
    girl e("Zhang Wei ",18);
    b.disp(e);
  }
```

程序运行结果如下:

```
  Boy′s name is Chen Hao ,age: 25
  Girl′s name is: Zhang Wei ,age: 18
```

友元类

不仅可以将一个函数声明为一个类的“朋友”，而且可以将一个类(例如B类)声明为另一个类(例如A类)的“朋友”。这时B类就是A类的友元类。友元类B中的所有函数都是A类的友元函数，可以访问A类中的所有成员。

友元类

一个类也可以作为另一个类的友元。

例如：

```
class  Y {
    //…
};
class X {
     //…
    friend  Y; // 声明类Y为类X的友元类
     //…
};
```

类Y的所有成员函数都是类X的友元函数

友元类

(1) 友元的关系是单向的而不是双向的。

(2) 友元的关系不能传递。

在实际工作中，除非确有必要，一般并不把整个类声明为友元类，而只将确实有需要的成员函数声明为友元函数，这样更安全一些。

关于友元利弊的分析： 面向对象程序设计的一个基本原则是封装性和信息隐蔽，而友元却可以访问其他类中的私有成员，不能不说这是对封装原则的一个小的破坏。但是它能有助于数据共享，能提高程序的效率，在使用友元时，要注意到它的副作用，不要过多地使用友元，只有在使用它能使程序精炼，并能大大提高程序的效率时才用友元。

Friend classes

```
// friend class
#include <iostream>
using namespace std;

class Square;
class Rectangle {
    int width, height;
public:
    int area () {return (width * height);}
    void convert (Square a);
};
class Square {
    friend class Rectangle;
private:
    int side;
public:
    Square (int a) : side(a) {}
};
```

Friend classes

```
void Rectangle::convert (Square a) {
    width = a.side;
    height = a.side;
}

int main () {
    Rectangle rect;
    Square sqr (4);
    rect.convert(sqr);
    cout << rect.area();

    return 0;
}
```

【例3.25】一个类作为另一个类的友元

```
#include<iostream.h>
#include<string.h>
class girl;
class boy {
public:
    boy(char *n,int d)
     {   name=new char[strlen(n)+1];
          strcpy(name,n);
           age=d;       }
     void disp(girl&);
     ~boy()
      {    delete name; }
private:
     char *name;
      int age;
};
```

```
class girl
{
    public:
        girl(char *n,int d)
        {  name=new char[strlen(n)+1];
            strcpy(name,n);
            age=d;
         }
         ~girl()
         {   delete name; }
      private:
        char *name;
        int age;
        friend boy;
};
```

```
void boy::disp(girl &x)
  { cout<<"Boy\'s name is "<<name <<",age:"<<age<<"\n";
     cout<<"Girl\'s name is "<<x.name <<",age:"<<x.age<<"\n";
   }
void main()
  { boy b("Chen Hao",25);
    girl e("Zhang Wei ",18);
     b.disp(e);
    }
```

注意：

友元关系是单向的，不具有交换性

即若类X是类Y的友元，但类Y不一定是类X的友元。

友元关系不具有传递性

即若类X是类Y的友元，类Y是类Z的友元，但类X不一定是类Z的友元。

3.6 对象成员

如果一个类的对象是另一个类的数据成员，则称这样的数据成员为对象成员。例如：

```
class A
 {
       //...
 };
class B
 {
      A a;      //类A的对象a为类B的对象成员
      public:
        //…
 };
```

使用对象成员着重要注意的问题是对象成员的初始化问题,即类B的构造函数如何定义?

例如有以下的类:

```
class  X{
    类名1    对象成员名1;
    类名2    对象成员名2;
       …
    类名n    对象成员名n;
};
```

一般来说,类X的构造函数的定义形式为;

```
X::X(形参表0):对象成员名1(形参表1),…,
                对象成员名i(形参表i) ,…,对象成员名n(形参表n)
{
   // …构造函数体
}
```

【例3.26】对象成员的初始化

```
#include<iostream.h>
class A{
public:
    A(int x1，float y1)
    { x=x1; y=y1; }
    void show()
    { cout<<"\n x="<<x<<" y="<<y; }
private:
    int x;
    float y;
};
```

```
class B{
public:
  B(int x1,float y1,int z1):a(x1,y1)
   { z=z1; }
  void show()
   { a.show();
     cout<<" z="<<z; }
private:
    A a;
    int z;
};
main()
{ B b(11,22,33);
  b.show();
  return 0;
}
```

具有对象成员的类构造函数的另一种定义

```
B(A a1，int z1):a(a1)
 { z=z1; }

main()
 { A a(11,22);
   B b(a,33);
   b.show();
   return 0;
}
```

【例3.27】对象成员的应用

```
#include<iostream.h>
#include<string.h>
class Score{
public:
        Score(float c,float e,float m);
        Score();
        void show();
        void modify(float c,float e,float m);
    private:
         float computer;
        float english;
        float mathematics;
 };
Score :: Score(float c,float e,float m)
{       computer = c;
        english = e;
        mathematics = m;
}
```

```
Score :: Score()
{       computer=english=mathematics=0;  }

void Score :: modify(float c,float e,float m)
{
        computer = c;
        english = e;
        mathematics = m;
}
void Score :: show()
{
        cout<<"\n Score computer: "<<computer;
        cout<<"\n Score english: "<<english;
        cout<<"\n Score mathematics: "<<mathematics;
}
```

```
class Student{
private:
    char *name;   // 学生姓名
    char *stu_no;  // 学生学号
     Score score1;              // 学生成绩(对象成员，是类Score的对象)
public:
    Student(char *name1,char *stu_no1,float s1,float s2,float s3);
    ~Student();    // 析构函数
    void modify(char * name1,char *stu_no1,float s1,float s2,float s3);
    void show();   // 数据输出
};
Student :: Student(char *name1,char *stu_no1,float s1,float s2,float
    s3) :score1(s1,s2,s3)
{    name=new char[strlen(name1)+1];
    strcpy(name,name1);
    stu_no=new char[strlen(stu_no1)+1];
    strcpy(stu_no,stu_no1);
}
```

```
Student :: ~Student()
{       delete []name;
        delete []stu_no;  }
void Student :: modify(char *name1,char *stu_no1,float s1,float s2,float s3)
{       delete []name;
        name=new char[strlen(name1)+1];
        strcpy(name,name1);
        delete []stu_no;
        stu_no=new char[strlen(stu_no1)+1];
        strcpy(stu_no,stu_no1);
         score1.modify(s1,s2,s3);
}
void Student :: show()
{       cout<<"\n name: "<<name;
        cout<<"\n stu_no: "<<stu_no;
        score1.show();
}
```

```
void main()
{
	Student stu1("Liming","990201",90,80,70);
	stu1.show();
	cout<<endl;
	stu1.modify("Zhanghao","990202",95,85,75);
	stu1.show();
}
```

```
name:Liming
stu_no:990201
Score computer: 90
Score english: 80
Score mathematics: 70
name:Zhanghao
stu_no:990202
Score computer: 95
Score english: 85
Score mathematics: 75
```

3.7共用数据的保护（const的使用）

1) const 对象的一般形式

类型名 const 对象名[(构造实参表列)];

const 类型名 对象名[(构造实参表列)];

注意：常对象必须要有初值（因为以后不能改值了）。

2) 限定作用:

定义为 const 的对象的所有数据成员的值都不能被修改。凡出现调用非const的成员函数，将出现编译错误。但构造函数除外。

```
Time const t1(12,34,46);  // t1是常对象
t1.set_Time();  // 编译错误PE，非const成员函数
t2.show_Time();  // 错误，非const成员函数
```

3) mutable成员

对数据成员声明为mutable（易变的）时，即使是const对象，仍然可以修改该数据成员值。

常对象成员

1）常数据成员

用 const 声明的常数据成员，其值是不能改变的。只能通过构造函数的参数初始化表对常数据成员进行初始化。

```
class Time
    {
const int hour;              // 声明hour为常数据成员
Time(int h):hour(h){}   // 通过参数初始化表对常数据成员hour初始化
    };
```

```
class Time
    {
const int hour;              // 声明hour为常数据成员
 Time(int h)
    {
hour = h;  // 错误
    }
};
```

2）常成员函数

成员函数声明中包含const时为常成员函数。此时，该函数只能引用本类中的数据成员，而不能修改它们（注意：可以改mutable成员），即成员数据不能作为语句的左值。

```
void show_Time( ) const;  // 注意const的位置在函数名和括号之后
void Time::show_Time const
{
cout << hour << minute << sec << endl;
}
```

注意：const是函数类型的一部分，在声明函数和定义函数时都要有const关键字，在调用时不必加const。

例3.28 常引用作函数形参

```
#include<iostream.h>
int add(const int& i,const int& j);
void main()
{  int a=20;
   int b=30;
   cout<<a<<"+"<<b<<"="<<add(a,b)<<endl;
}
int add(const int& i,const& j)
{
   return i+j;
}
```

例.const对象及mutable成员

```
class Time
{
public:
   Time()
   {
      hour = 0;
      minute = 0;
   }
   void set_Hour(int h)  const  { hour = h;}
   void set_Minute(int m) { minute = m; }
   void show_Time() const
   { cout<<hour <<":"<<minute << endl; }
private:
   mutable int hour;
   int minute;
};
```

```
int main()
{
   Time t1;
    t1.set_Hour(8);
    t1.set_Minute(8);
    t1.show_Time();

    Time const t2;
    t2.set_Hour(8);
    t2.set_Minute(8);
    t2.show_Time();
    return 0;
}
```

3）常成员的使用

a) 如果在一个类中，有些数据成员的值允许改变，另一些数据成员的值不允许改变，则可以将一部分数据成员声明为const，以保证其值不被改变，可以用非const的成员函数引用这些数据成员的值，并修改非const数据成员的值。

b) 如果要求所有的数据成员的值都不允许改变，则可以将所有的数据成员声明为const，或将对象声明为const(常对象)，然后用const成员函数引用数据成员，这样起到“双保险”的作用，切实保证了数据成员不被修改。

c) 如果已定义了一个常对象，则只能调用其中的const成员函数，而不能调用非const成员函数(不论这些函数是否会修改对象中的数据)。如果需要访问对象中的数据成员，可将常对象中所有成员函数都声明为const成员函数，但应确保在函数中不修改对象中的数据成员。

指向*常对象*的指针变量

1）指向常对象的指针变量的一般形式

const 类型 *指针变量名

```
const char *pc;  // pc 指向的 char 是const型的
*pa = 'a';  // 错误：pa 指向的目标不能改变
pa++;  // 正确，pa 本身的值可以改变
```

```
Time t1;
const Time *pt;
*pt = t1;  // 错误：pt指向的目标不能改变
```

2）关于指向常对象的指针变量的说明

a) 指向常对象（变量）的指针变量，不能通过它来改变所指向目标对象的值，但指针变量的值是可以改变的。

b) 如果被声明为常对象（变量），只能用指向常对象（变量）的指针变量指向它，而不能非const型指针变量去指向它。

```
const int a = 10, b = 20;
const int *pa = &a;  // 正确
int *pb = &a;  // 错误：非 const 型指针
```

c) 指向常对象（变量）的指针变量除了可以指向常对象（变量）外，还可以指向未被声明为const的对象（变量）。此时不能通过此指针变量改变该变量的值。

```
pa = &b;  // 正确：也可指向非 const 型的变量
*pa = 30;  // 错误：指向 const 型的指针不能改目标值
```

d) 指向常对象（变量）的指针变量可以指向const和非const型的对象（变量），而指向非const型变量的指针变量只能指向非const的对象（变量）。

```
const int *pa = &b;  // 正确
int *pb = &a;  // 错误
```

e) 如果函数的形参是指向非const型变量的指针，实参只能用指向非const变量的指针，而不能用指向const变量的指针，这样，在执行函数的过程中可以改变形参指针变量所指向的变量(也就是实参指针所指向的变量)的值。

f) 如果函数形参是指向const型变量的指针，允许实参是指向const变量的指针，或指向非const变量的指针。

void f(Time *pt);	**void g(const Time *pt)**
Time *p1;	**Time *p1;**
const Time*p2;	**Const Time *p2;**
f(p1); // 正确	**f(p1); // 正确**
f(p2); // 错误	**f(p2); // 正确**

g) 基本规则：希望在调用函数时对象的值不被修改，就应当把形参定义为指向常对象的指针变量，同时用对象的地址作实参(对象可以是const或非const型)。如果要求该对象不仅在调用函数过程中不被改变，而且要求它在程序执行过程中都不改变，则应把它定义为const型

对象的常引用

在C++中，经常用常指针和常引用作函数参数。这样既能保证数据安全，使数据不能被随意修改，在调用函数时又不必建立实参的拷贝，可以提高程序运行效率。

例 对象的常引用。

```
class Time
{
public:
    Time(int,int,int);
    int hour;
    int minute;
    int sec;
};
Time::Time(int h,int m,int s)
{
    hour = h;
    minute = m;
    sec = s;
}
```

```
void fun1(Time &t)  // t 是对象引用
{ t.hour = 18; }

void fun2(const Time &t)  // t 为常引用
{ t.hour = 18;  // 错误 }

int main( )
{
    void fun1(&);
    void fun2(const Time &);
    Time t1(10,13,56);
    fun(t1);
    cout<<t1.hour<<endl;
    return 0;
}
```

例3.29 非常对象和常对象的比较。

```
#include<iostream.h>
    class Sample {
public:
    int m;
    Sample(int i,int j){  m=i; n=j;}
    void setvalue(int i){  n=i; }
    void disply(){ cout<<"m="<<m<<endl;
    cout<<"n="<<n<<endl;
}
private:
    int n;
};
void main()
{   Sample a(10,20); //若为:const Sample a(10,20);
    a.setvalue(40);//不能修改常对象的数据成员
    a.m=30; //不能修改常对象的数据成员
    a.disply();//常对象不能调用普通成员函数
}
```

常对象成员

1. 常数据成员

使用const说明的数据成员称为常数据成员。

如果在一个类中说明了常数据成员,那么构造函数就只能通过初始化列表对该数据成员进行初始化,而任何其他函数都不能对该成员赋值。

例3.30 常数据成员举例。

```
#include <iostream.h>
class Date {
   public:
         Date(int y,int m,int d);
          void showDate();
          const int &r;
   private:
       const int year ;  const int month; const int day;  };
Date::Date(int y,int m,int d) :year(y),month(m),day(d),r(year){ }
inline void Date::showDate()
{    cout<<"r="<<r<<endl;
     cout<<year<<"."<<month<<"."<<day<<endl;}
void main()
{    Date date1(1998,4,28);    date1.showDate(); }
```

2.常成员函数

在类中使用关键字const说明的函数为常成员函数,常成员函数的说明格式如下:

类型说明符 函数名(参数表)const;

const是函数类型的一个组成部分,因此在函数的实现部分也要带关键字const。

如果将一个对象说明常对象，则通过该对象只能调用它的常成员函数，而不能调用普通成员函数。而且常成员函数也不能更新对象的数据成员。

例3.31 常成员函数的使用。

```
#include <iostream.h>
class Date {
    public:
       Date(int y,int m,int d);
       void showDate();
       void showDate() const;
    private: int year; int month; int day;  };
Date::Date(int y,int m,int d):year(y),month(m),day(d){ }
void Date::showDate()
{   cout<<"ShowDate1:"<<endl;
    cout<<year<<"."<<month<<"."<<day<<endl;}
void Date::showDate() const
{   cout<<"ShowDate2:"<<endl;
    cout<<year<<"."<<month<<"."<<day<<endl; }
void main()
{   Date date1(1998,4,28); date1.showDate();
   const Date date2(2002,11,14); date2.showDate(); }
}
```

const型数据的小结

Time const t = Time(1,2,3); **const Time t = Time(1,2,3);** **const int a = 10;** **int const a = 10;**	**t是常对象，其成员值在任何情况下都不能被改变。** **a 是常变量，其值不能被改变**
void Time::fun() const;	**fun 是Time类的常成员函数，可以调用该函数，但不能修改本类中的数据成员（非mutable）**
Time * const pt; **int * const pa;**	**pt 是指向Time对象的常指针，pa 是指向整数的常指针。指针值不能改变。**
const Time *pt; **const int *pa;**	**pt是指向Time类常对象的指针，pa是指向常整数的指针，不能通过指针来改变指向的对象（值）**

3.8 C++程序的多文件组成

- 一个源程序可以划分为多个源文件：
  - 类声明文件（.h文件）
  - 类实现文件（.cpp文件）
  - 类的使用文件（main()所在的.cpp文件）
- 利用工程来组合各个文件。

3.8 C++程序的多文件组成

【例3.32】一个源程序按照结构划分为3个文件

```
// 文件1 student.h （类的声明部分）
#include<iostream.h>
#include<string.h>
class Student {
private:
    char *name;              // 学生姓名
    char *stu_no;            // 学生学号
    float score;             // 学生成绩
public:                      // 类的外部接口
    Student(char *name1,char *stu_no1,float score1); // 构造函数
     ~Student();             // 析构函数
    void modify(float score1);      // 数据修改
    void show();             // 数据输出
};
```

```
// 文件2 student.cpp (类的实现部分)
#include "student.h"        // 包含类的声明文件
Student :: Student(char *name1,char *stu_no1,float score1)
{     name=new char[strlen(name1)+1];
      strcpy(name,name1);
      stu_no=new char[strlen(stu_no1)+1];
     strcpy(stu_no,stu_no1);
     score=score1;    }
Student :: ~Student()
{    delete []name;
     delete []stu_no;     }
void Student :: modify(float score1)
{     score=score1;     }
void Student :: show()
{     cout<<"\n name: "<<name;
      cout<<"\n stu_no: "<<stu_no;
      cout<<"\n score: "<<score;     }
```

// 文件3 studentmain.cpp （类的使用部分）

```
#include "student.h"        // 包含类的声明文件
void main()
{
        Student stu1("Liming","990201",90);
        stu1.show();
        stu1.modify(88);
        stu1.show();
}
```

3.9 程序举例

【例3.33】利用类表示一个堆栈(stack)，并为此堆栈建立push()、pop()及显示堆栈内容的showstack()等函数

```
//文件1 stack.h
#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>
const int SIZE=10;
class stack
{    int stck[SIZE];          // 数组,用于存放栈中数据
     int tos;                          // 栈顶位置（数组下标）
public:
     stack();
     void push(int ch);       // 将数据ch压入栈
     int pop();                        // 将栈顶数据弹出栈
     void ShowStack();
};
```

```
// 文件2 stack.cpp
#include <iostream.h>
#include "stack.h"
stack :: stack()                              // 构造函数,初始化栈
{   tos= 0;  }
void stack :: push(int ch)
{
    if(tos==SIZE)
    {
        cout<<"Stack is full";
        return;
    }
    stck[tos]=ch;
    tos++;
    cout<<"You have pushed a data into the stack!\n";
}
```

```
int stack :: pop()
{
    if (tos==0)
    {   cout<<"Stack is empty";
        return 0;      }
    tos--;
    return stck[tos];
}
void stack :: ShowStack()
{   cout<<"\n The content of stack: \n" ;
    if (tos==0)
    {   cout<<"\nThe stack has no data!\n";
        return;    }
    for (int i=tos-1; i>=0;i--)
    cout<<stck[i]<<"  ";
    cout<<"\n\n";
}
```

```
//文件3 stackmain.cpp
#include <iostream.h>
#include "stack.h"
main()
{
    cout<<endl;
    stack ss;
    int x;
    char ch;
    cout<<" <I> ------ Push data to stack\n";
    cout<<" <O> ------ Pop data from stack\n";
    cout<<" <S> ------ Show the content of stack\n";
    cout<<" <Q> ------ Quit... \n";
```

```
 while (1)
   {   cout<<"Please select an item: ";
       cin>>ch;
       ch=toupper(ch);
       switch(ch)
       {    case 'I':
            cout<<"\n Enter the value that "<<"you want to push: ";
            cin >>x;
            ss.push(x);              break;
          case 'O':
            x=ss.pop();
            cout<<"\n Pop "<<x<<" from stack.\n";              break;
          case 'S':
            ss.ShowStack();              break;
          case 'Q':
            return 0;
          default:
          cout<<"\n You have inputted a wrong item! Please try again!\n";
            continue;     }
   }
}
```

嵌套类

- 在一个类中定义的类称为嵌套类，定义嵌套类的类称为外围类。
- 定义嵌套类的目的在于隐藏类名，减少全局的标识符，从而限制用户能否使用该类建立对象。这样可以提高类的抽象能力，并且强调了两个类（外围类和嵌套类）之间的主从关系。
- 对嵌套类的若干说明:

1.从作用域的角度上看，嵌套类被隐藏在外围类之中，该类名只能在外围类中使用。如果在外围类的作用域内使用该类名时，需要加名字限定。

2.从访问权限的角度来看，嵌套类名与它的外围类的对象成员名具有相同的访问权限规则。不能访问嵌套类的对象中的私有成员函数，也不能对外围类的私有部分中的嵌套类建立对象。

3.嵌套类中的成员函数可以在它的类体外定义。

4.嵌套类中说明的成员不是外围类中对象的成员，反之亦然。嵌套类的成员函数对外围类的成员没有访问权，反之亦然。因此，在分析嵌套类与外围类的成员访问关系时，往往把嵌套类看作非嵌套类来处理。

5.在嵌套类中说明的友元对外围类的成员没有访问权。

6.如果嵌套类比较复杂，可以只在外围类中对嵌套类进行说明，关于嵌套类的详细的内容可在外围类体外的文件域中进行定义。

谢谢大家！