第 4 章　对象成员和友元

类实现了不同类型数据的聚集，也就是说，类的数据成员可以是不同的数据类型，那么由用户自定义的数据类型在类机制中如何使用呢？另外，类实现了数据的封装和保护，类的外部函数如果要访问类内的数据成员又要如何实现呢？本章主要针对以上问题讲解 C++面向对象中的两个重要内容：对象成员和友元。

- 对象成员
- 静态数据成员和静态成员函数的概念和用法
- 友元的概念和用法

4.1　对象成员

对象成员是指在类的定义中数据成员可以为其它类的对象，即类对象作为另一个类的数据成员。

如果在类定义中包含有对象成员，则在创建类对象时先调用对象成员的构造函数，再调用类本身的构造函数。析构函数和构造函数的调用顺序正好相反。

【例 4.1】对象成员。

```
#include<iostream>
using namespace std;
class StudentID{
  public:
    StudentID(int id=0)          // 带缺省参数的构造函数
     {
       value=id;
       cout <<"Assigning student id " <<value <<endl;
     }
   ~StudentID()
    {
       cout <<"Destructing id " <<value <<endl;
    }
private:
  int value;
};
```

```
class Student{
public:
  Student(char* pName="no name",int ssID=0):id(ssID)
  {
    cout <<"Constructing student " <<pName <<endl;
    strncpy(name,pName,sizeof(name));
    name[sizeof(name)-1]='\n';
  }
  ~Student()
  {cout<<"Deconstructing student  "<<name<<endl;}
protected:
  char name[20];
  StudentID id;                // 对象成员
};
void main()
{
  Student s("li ming",20090101);
  Student t("wang lan");
}
```

程序运行结果如下：

```
Assigning student id 20090101
Constructing student li ming
Assigning student id 0
Constructing student wang lan
Deconstructing student  wang lan
Destructing id 0
Deconstructing student  li ming
Destructing id 20090101
```

从程序执行的结果可以看出，如果一个程序中有对象成员，程序执行时，首先调用类本身的构造函数，在执行本身构造函数的函数体之前，调用成员对象的构造函数，然后再执行类本身构造函数的函数体。

4.2　对象数组与对象指针

4.2.1　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，我们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，而且具有成员函数。

1. 一维对象数组的定义

类名 数组名[下标表达式];

例如：

```
Student obj[10];
```

定义了类 Student 的对象数组 obj。

2. 一维对象数组的访问

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某个数组元素，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，一般格式是：

数组名[下标].成员名

【例 4.2】对象数组。

```
#include<iostream>
using namespace std;
    class Studentid {
    public:
        void set(int n)
            { x=n; }
        int get()
            { return x; }
        private:
        int x;
    };
void  main()
{   Studentid ob[4];
    int i;
    for (i=0;i<4;i++) //利用循环对 ob[i]赋值
        ob[i].set(i);
    for (i=0;i<4;i++) //循环打印 ob[i]
        cout<<"20090"<<ob[i].get()<<' '<<endl;
  }
```


程序的运行结果为：

```
200900
200901
200902
200903
```

3. 一维对象数组的初始化

建立某个类的对象数组时，在设计类的构造函数时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，应该在类中定义不带参数的构造函数或带缺省参数的构造函数；当各元素对象的初值要求为不同的值时要求定义带形参的构造函数。定义对象数组时，可通过初始化表进行赋值。

【例 4.3】通过初始化表给对象数组赋值。

```
#include<iostream>
using namespace std;
class Studentid
{
  public:
     Studentid ()//类的初始化
     {x=0;}
     Studentid (int n)//初始赋值
```

```
    {x=n;}
    int get()
    {return x;}
  private:
    int x;
};
void main()
{
  Studentid ob1[4]={1,2,3,4};
  Studentid ob2[4]={5,6};
  Studentid ob3[4]={ Studentid (1), Studentid (2), Studentid (3),Studentid(4)};
  Studentid ob4[4]={ Studentid (5), Studentid (6)};
  ob4[2]= Studentid (7);
  ob4[3]= Studentid (8);
  int i;
  for(i=0;i<4;i++)
  cout<<"20090"<<ob1[i].get()<<' ';
  cout<<endl;
  for(i=0;i<4;i++)
  cout<<"20090"<<ob2[i].get()<<' ';
  cout<<endl;
  for(i=0;i<4;i++)
  cout<<"20090"<<ob3[i].get()<<' ';
  cout<<endl;
  for(i=0;i<4;i++)
  cout<<"20090"<<ob4[i].get()<<' ';
  cout<<endl;
}
```


程序的运行结果如下：

```
200901 200902 200903 200904
200905 200906 200900 200900
200901 200902 200903 200904
200905 200906 200907 200908
```

说明：本例在执行语句“Studentid ob1[4]={1,2,3,4};”时，分别初始化 ob1[0]，ob1[1]，ob1[2]和 ob1[3]。如果没有指定初始值，就调用不带参数的构造函数。

二维对象数组初始化的方法可结合一维对象数组，参考二维数组的初始化方法来进行。

4.2.2　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，既可以通过一个对象名访问对象，也可以通过对象地址来访问一个对象，对象指针就是一个用来存放对象地址的变量。当指针加 1 或减 1 时，它总是指向其基本类型中的一个元素，对象指针也是如此。指针对象加 1 时，指向下一个数组对象元素。

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

对象指针和其它数据类型的指针相同。使用对象指针时，首先要把它指向一个已创建的对象，然后才能访问对象的公有成员。

在第 3 章中我们已经知道，用点运算符（.）来访问对象的成员，当用指向对象的指针访问对象成员时，就要用“->”操作符。

【例 4.4】对象指针的使用。

```
#include<iostream>
using namespace std;
class A{
      public:
      void set(int a){ x=a; }
      void show(){ cout<<x<<endl; }
      private:
     int x;
    };
void main()
{ A ob,*p;           // 声明类 A 的对象 ob 和类 A 的对象指针 p
   ob.set(15);
   ob.show();        // 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();        // 利用对象指针访问对象的成员
};
```


程序的运行结果如下：

```
15
15
```

此例声明了一个类 A，ob 是类 A 的一个对象，p 是类 A 的对象指针，对象 ob 的地址是用地址操作符（&）获得并赋给对象指针 p 的。

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。

A　ob[2]表示声明了对象数组 ob[2]，而 p=ob 表示将数组对象的首地址赋给对象指针 p。

将上面的例子的 main()函数改写为：

```
void  main()
{
    A ob[2],*p;
    ob[0].set(10);
    ob[1].set(20);
    p=ob;
    p->show();
    p++;
    p->show();
}
```

程序的运行结果如下：

```
10
20
```

4.2.3　指向类的成员的指针

C++提供一种特殊的指针类型，它指向类的成员，而不是指向该类的一个对象中该成员的一个实例，这种指针称为指向类成员的指针。通过指向成员的指针只能访问公有的数据成员和成员函数。

1. 指向数据成员的指针

指向数据成员的指针格式如下：

类型说明符　类名::*数据成员指针名;

声明指向数据成员的指针后，需要对其进行赋值，也就是要确定指向类的哪一个成员。指向对象的成员的指针使用前也要先声明，再赋值，然后访问。

对数据成员赋值的一般格式如下：

数据成员指针名=&类名::数据成员名;

由于类是通过对象而实例化的，只有在定义了对象时才能为具体的对象分配内存空间，这时只要将对象在内存中的起始地址与成员指针中的存放的相对偏移结合起来就可以访问到对象的数据成员了。用数据成员指针访问数据成员可以通过以下两种格式实现：

对象名.*数据成员指针名

或

对象指针名->*数据成员指针名

【例 4.5】指向数据成员的指针。

```
#include<iostream>
using namespace std;
class DataPointer{
  public:
    DataPointer(int x)
      {z=x;}
    int z;
};
void main()
{
  DataPointer ob(10);   //声明一个对象指针 pc1
  DataPointer *pc1;
  pc1=&ob;  //给对象指针 pc1 赋值
  int DataPointer::*pc2;
  pc2=&DataPointer::z;//指针指向 DataPointer 域中的 z 值
     //以上两语句可以合写成 int DataPointer::*pc2=&DataPointer::z;
  cout<<ob.*pc2<<endl;
  cout<<pc1->*pc2<<endl;
  cout<<ob.z<<endl;
}
```

程序运行结果如下：

```
10
10
10
```

2. 指向成员函数的指针

指向成员函数的指针格式如下：

类型说明符　(类名::*指针名)(参数表);

对成员函数指针赋值的一般格式为：

成员函数指针名=&类名::成员函数名;

成员函数指针在声明后要对其赋值，也就是要确定指向类的哪一个成员函数。对于一个普通函数而言，函数名就表示它的起始地址，将起始地址赋给指针，就可以通过指针调用函数。虽然类的成员函数并不在每个对象中复制一份拷贝，但是语法规定必须要通过对象来调用成员函数，因此上述赋值之后，也还不能用指针直接调用成员函数，而是要首先声明类的对象，然后通过下面两种形式利用成员函数指针调用成员函数：

(对象名.*成员函数指针名)(参数表)

或

(对象指针名—>*成员函数指针名)(参数表)

【例 4.6】指向成员函数的指针。

```
#include<iostream>
using namespace std;
class FunctionPointer{
public:
  FunctionPointer(int m){x=m;}//初始化
  void print()
  {cout<<"x="<<x;   }
   private:
           int x;
   };
void main()
{
   FunctionPointer a(1);
   void (FunctionPointer:: *p)()=&FunctionPointer::print;//给成员函数指针赋值
   (a.*p)();//调用成员函数
       /*或者如此调用
      FunctionPointer *q;
       q=&a;
      (q->*p)();*/
}
```

程序运行结果：

```
x=1
```

3. 由类外指向类内的指针

语法声明格式：

类型符　类名::*指针名=类中数据成员地址描述;

【例 4.7】类外指向类内的指针变量。

```
#include<iostream>
using namespace std;
```

```
class  A
{
   public:
     int  i,*p;
     A(){i=10;p=&i;}
};
   int  A::*p=&A::i; // p 是类外指针数据
void  main()
{
   A  aa,bb; // 按缺省构造函数的安排，aa、bb 中的 i 初值都是 10
    (bb.*p)++;   // 括号不可缺，否则编译器将理解为非法操作
   --*aa.p;
   cout<<"AA:"<<aa.*p<<"  BB:"<<bb.*p<<"\n";
   cout<<"AA:"<<*aa.p<<"  BB:"<<*bb.p<<endl;
}
```

程序的运行结果为：

```
AA:9  BB:11
AA:9  BB:11
```

例 4.7 中的指针变量 p 出现在两处，一处是类 A 的构造函数中（p=&i），另一处是全局数据区中。在本例中它们彼此并不冲突，各行其事。要特别注意由类外指向类内的指针数据在使用时的写法：

类所定义的对象名．指针名…

这种在类外定义的指向类内数据成员的指针在使用上由于受类的数据封装特性的限制，其所指成员只能处于 public 区中。

4. 类外指向成员函数的指针

语法声明格式：

类型符(类名::*指针名)(参数类型表)=&类名::函数名;

与指向数据成员的指针相同，这里的成员函数也只能在 public 区中声明。

【例 4.8】类外指向成员函数的指针变量。

```
#include<iostream>
using namespace std;
class  A
{
    int  i;
   public:
    int  set(int  k)
   {i=++k;return  i;}
};
void  main()
{
   int  (A::*f)(int)=&A::set;
   A  aa;
   cout<<(aa.*f)(10)<<endl;     // 括号不能省略
}
```

程序的运行结果为：

```
11
```

4.3 向函数传递对象

在第 2 章讲到函数的时候，我们知道函数中参数的传递分为值传递和引用传递，对象作为一种自定义的数据类型，也同样可以作为参数在函数中传递。

对象也可以像其它类型的数据一样作为参数传递给函数，不同的是，它是通过传值调用给函数的。在函数中对对象的任何修改不会影响调用函数的对象本身。

【例 4.9】使用对象作为函数参数。

```
#include<iostream>
using namespace std;
class Object{
   public:
       Object(int n) { x=n; }
       void set(int n){ x=n; }
int get( ){ return x; }
private:
       int x;
};
void Multiplication (Object a)//值的更改说明不影响对象本身
{   a.set(a.get()*a.get());
       cout<<"调用后 a 的值为：";
       cout<<a.get()<<"\n"; }
void main()
{   Object  a(10);
   Multiplication (a);
   cout<<"主函数中 a 的值为:";
   cout<<a.get( );
}
```

程序运行结果如下：

```
调用后 a 的值为： 100
主函数中 a 的值为：10
```

和其他类型的变量一样，也可以将对象的地址传递给函数。这时函数对对象的修改将影响到调用函数对象的本身。

1. 使用对象指针作为函数参数

使用对象指针作为函数参数可以实现传递地址调用，可实现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。当函数的形参是对象的指针时，调用函数的对应实参应该是某个对象的地址值。

【例 4.10】使用对象指针作为函数参数。

```
#include<iostream>
using namespace std;
class Object{
   public:
       Object(int n) { x=n; }
```

```
        void set(int n){ x=n; }
        int get( ){ return x; }
private:
        int x;
};
    void Multiplication (Object *a) //把对象地址当形参传过来
    {  a->set(a->get()*a->get());    //这里必须用->而不能用.，因为用. 作对象内
                                     //成员的访问
   cout<<"调用后 a 的值为：";
   cout<<a->get()<<"\n"; }
   void main()
   {  Object  a(10);
      Multiplication (&a);
      cout<<"调用后主函数中对象 a 中的值为 :";
      cout<<a.get() <<"\n";
 }
```

程序的运行结果如下：

```
调用后 a 的值为：100
调用后主函数中对象 a 中的值为：100
```

不难看出，调用函数前 a.x 的值是 10，调用后的值变为 100，可见在函数中对对象的修改，影响了调用该函数的对象本身。

2. 使用对象引用作为函数参数

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递地址调用，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传递给形参，并不进行拷贝，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的指针时，调用函数的对应实参应该是某个对象的地址值。下面对例 4.10 稍作修改，说明对象指针作为函数参数这个问题。

【例 4.11】使用对象引用作为函数参数。

```
#include<iostream>
using namespace std;
class Object{
   public:
       Object(int n) { x=n; }
       void set(int n){ x=n; }
       int get( ){ return x; }
private:
       int x;
};
void Multiplication (Object &a)//这里只是把指针换成了引用
{    a.set(a.get()*a.get());//这里必须用.而不能用->，这里是对类内成员的直接访问
       cout<<"调用后 a 的值为：";
       cout<<a.get()<<"\n"; }
void main()
{   Object  a(10);
```

```
    Multiplication (a);
    cout<<"调用后主函数中对象 a 中的值为 :";
    cout<<a.get() <<"\n";
}
```

程序的运行结果如下：

```
调用后 a 的值为：100
调用后主函数中对象 a 中的值为：100
```

由程序结果可以看出，使用对象指针作为函数参数与使用对象引用作为函数参数，其输出结果完全相同。

4.4 静态成员

通常在解决实现一个类的不同对象之间的数据和函数共享问题时，可以定义全局变量，但全局变量的安全性得不到保证，因此在实际工作中常采用定义静态数据成员的方法来实现。

4.4.1 静态数据成员

1. 静态数据成员的定义

定义格式如下：

static 数据类型 数据成员名;

例如：


```
class Student
{
     char stu_no[8];      //学号
     float score;         //学生成绩
     static float sum;    //sum 为静态数据成员
    public:
    ……
};
```

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。

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。

2. 静态成员的初始化

数据类型 类名::静态数据成员名=值

例如：float Student::sum=0;

注意：

（1）不能用参数初始化表对静态数据成员初始化。

例：Student(char *no,float sc):sum（0）{ } //错误

（2）静态数据成员初始化只能在类体外进行。

3. 静态数据成员的引用方式

对象名.静态数据成员名，如：stu1.sum

对象指针->静态数据成员名，如：p->sum

类名::静态数据成员名，如：Student::sum

注意：如果静态数据成员被定义为私有的，则不能在类外直接引用，而必须通过公用的成员函数引用。

使用说明：

（1）静态数据成员在所有对象之外单独开辟空间，只占一份空间。

（2）编译时被分配空间的，到程序结束时才释放空间。

（3）只要在类中定义了静态数据成员，即使不定义对象，也为静态数据成员分配空间，它可以被引用。

（4）一个类中可以有一个或多个静态数据成员，所有对象共享这些静态数据成员，都可以引用它。

（5）静态数据成员的值对所有对象都是一样的。如果改变它的值，则在各对象中这个数据成员的值都同时改变了。

【例 4.12】静态数据成员的使用引例。

```
#include <iostream>
using namespace std;
class Myclass
{
public:
    Myclass(int a, int b, int c);
    void GetNumber();
    void GetSum();
private:
    int A, B, C;
    static int Sum; //定义静态成员 Sum
};
int Myclass::Sum = 0; //静态成员的初始化，必须在类外完成
Myclass::Myclass(int a, int b, int c)
{
    A = a;
    B = b;
    C = c;
    Sum+=A+B+C; //累加到 Sum 中，看出静态成员数据的作用
}
void Myclass::GetNumber()//得到各成员的值
{
    cout<<"Number="<<A<<" "<<B<<" "<<C<<" "<<endl;
}
void Myclass::GetSum()//得到其和
{cout<<"Sum="<<Sum<<endl;}
void main()
{
    Myclass M(1, 2, 3);//初始化对象 M
    M.GetNumber();
    M.GetSum();
    Myclass N(4, 5, 6);//初始化对象 N
```

```
    N.GetNumber();
    N.GetSum();
}
```

程序运行结果：

```
Number=1 2 3
Sum=6
Number=4 5 6
Sum=21
```

从结果中可以看出，对象 M 得到和为 6，保存在 Sum 中，而对象 N 调用时，Sum 中的值仍为 6，结果变为 21，说明了数据的共享。

4.4.2 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);

与静态数据成员类似，调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 4.13】利用静态成员函数来访问静态数据成员。

```
#include <iostream>
using namespace std;
class M
{
  public:
    M(int a)
    {  B+=a;}
      static void f(M m);
  private:
    static int B;
};
  int M::B=0;
  void M::f(M m)
{
  cout<<"B="<<B<<endl;
}
void main()
{
    M P(5);//定义对象 P
    M *q=&P; //定义对象指针，指向 P
    M::f(P);//用类名调用
    q->f(P);//用对象指针调用
    M Q(10);//定义对象 Q
    Q.f(Q);//用对象调用
}
```

程序运行结果：

```
B=5
B=5
B=15
```

通过此例，可以看出静态成员和静态函数的调用方式以及其作用。

静态成员函数的优点：即使不存在类的对象，它们也存在，并且可以调用。

静态成员函数与非静态成员函数的区别：

（1）静态成员函数属于类，非静态成员函数属于对象。

（2）调用非静态成员函数时，系统会把调用对象的 this 指针传递给成员函数。

（3）静态成员函数没有 this 指针。无法对一个对象中的非静态成员进行默认访问。

（4）在静态成员函数的实现中不能直接引用类中声明的非静态成员，但可以通过对象来访问。

4.4.3　通过普通指针访问静态成员

可以通过指向类的静态成员的普通指针来访问类的静态成员。

【例 4.14】通过指针访问类的静态数据成员。

```
#include<iostream>
using namespace std;
    class myclass
{
    public:
      myclass()  // 构造函数,每定义一个对象,静态数据成员 i 加 1
          { ++i; }
      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
int main()
{
   int *count=&myclass::i;     // 声明一个 int 型指针,指向类的静态成员
   myclass ob1,ob2,ob3,ob4;
     cout<<"myclass::i="<<*count<<endl;//通过指针直接访问静态数据成员
     return 0; }
```

程序的结果如下：

```
myclass::i=4
```

4.5　友元

友元提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。对于一个类，可以利用关键字 friend 将一般函数、其他类的成员函数或者其他类声明为该类的友元，使得这个类中本来隐藏的信息（包括私有成员和保护成员）可以被友元访问。如果友元是一般成员函数或是类的成员函数，称为友元函数；如果友元是一个类，则称为友元类，友元类的所有成员函数都成为友元函数。

4.5.1 友元函数

友元函数是独立于当前类的外部函数，不属于当前类。但它可以访问该类的所有对象的成员，包括私有成员、保护成员和公有成员。声明友元函数时，只需要在函数名前加上 friend 关键字，它可以定义在类内的任何部分，也可以定义在类外。

【例 4.15】友元函数的使用。

```
#include<iostream>
using namespace std;
class Book{
public:
    Book(char *s,int n)
    {
       name=new char[strlen(s)+1];//为 name 分配空间
       strcpy(name,s);//为 name 赋值
       ID=n;//为 ID 赋值
    }
    friend  void print(Book &);//声明友元函数
    ~Book(){delete  [] name;} //析构函数，释放掉 name 的空间
private:
        char *name;
        int ID;
};
void  print(Book &book) //定义友元函数
{
    cout<<"书目的名称是："<<book.name<<"   ID 是："<<book.ID<<endl;
}
void main()
{
    Book book("c++友元函数的使用",1);
    print(book);
}
```

程序的运行结果如下：

```
书目的名称是：c++友元函数的使用    ID 是：1
```

4.5.2 友元成员

其他类的成员函数也可以声明为一个类的友元函数，这个成员函数称为友元成员。友元成员不仅可以访问自己所在类对象中的私有成员和公有成员，还可以访问 friend 声明语句所在类对象中的私有成员和公有成员，这样能使两个类相互合作、协调工作，完成某一任务。

友元成员的使用和一般友元函数的使用基本相同，只是在使用该友元成员时通常需要进行前向引用声明，并且要通过相应的类和对象名进行访问。

【例 4.16】一个类的成员函数作为另一个类的友元。

```
#include<iostream>
using namespace std;
```

```
class Book;
class BookInformation{
public:
   BookInformation(char *s,int n)
{
    publisher=new char[strlen(s)+1];//为 publisher 分配空间
    strcpy(publisher,s);//为 publisher 赋值
       Data=n;//为 Data 赋值
}
  void  print(Book &);//声明 print()为类 BookInformation 的成员函数
    ~BookInformation(){delete  [] publisher;} //析构函数，释放掉 publisher 的空间
private:
    char *publisher;
    int Data;
};
class Book{
   public:
    Book(char *s,int n)
     { name=new char[strlen(s)+1];//为 name 分配空间
       strcpy(name,s);//为 name 赋值
      ID=n;//为 ID 赋值
      }
   friend void BookInformation::print(Book &);  //声明类 BookInformation 的
                                                //成员函数
   ~Book(){delete  [] name;} //析构函数，释放掉 name 的空间
private:
   char *name;
   int ID;
};
void  BookInformation::print(Book &book)     //定义友元函数
{
    //访问 Book 类对象的成员
   cout<<"书目的名称是："<<book.name<<"   ID 是："<<book.ID<<endl;
    //访问 BookInformation 类对象的成员
   cout<<"书的出版社："<<publisher<<"  出版时间："<<Data<<endl;
}
void main()
{
   BookInformation info("中国水利水电出版社",2009);
   Book book("c++友元函数的使用",1);
   info.print(book);
}
```

程序运行结果如下：

```
书目的名称是：c++友元函数的使用    ID 是：1
书的出版社：中国水利水电出版社   出版时间：2009
```

4.5.3 友元类

一个类作为另一个类的友元时，该类称为友元类。友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包括私有成员和保护成员）。

友元类可以在另一个类的公有部分或私有部分进行说明，说明方法如下：

friend <类名>; //友元类类名

使用友元类时注意：

（1）友元关系不能被继承。

（2）友元关系是单向的，不具有交换性。若类 X 是类 Y 的友元，类 Y 不一定是类 X 的友元，要看在类中是否有相应的声明。

（3）友元关系不具有传递性。若类 X 是类 Y 的友元，类 Y 是 Z 的友元，类 X 不一定是类 Z 的友元。

友元类的应用请参考 4.7 节实例 2。

4.6 常类型

虽然数据隐藏保证了数据的安全性，但各种形式的数据共享却不同程度地破坏了数据的安全性。因此，对于既需要共享，又需要防止改变的数据应该定义为常类型进行保护，以保证它在整个程序运行期间是不可改变的。这些常量需要使用 const 修饰符进行定义。const 关键字不仅可以修饰类对象本身，也可以修饰类对象的成员函数和数据成员，分别称为常对象、常成员函数和常数据成员。

4.6.1 常引用

常引用的说明形式如下：

const 类型说明符& 引用名

常引用引用的对象不允许更改。

【例 4.17】常引用作函数形参。

```
#include<iostream>
using namespace std;
int count(const int& i,const int& j)
{
    return (i+j);
}
void main()
{
    int a=10;
    int b=20;
    cout<<a<<"+"<<b<<"="<<count(a,b)<<endl;
}
```

程序的运行结果如下：

```
10+20=30
```

4.6.2　常对象

常对象在定义时必须进行初始化，而且不能被更新。常对象的说明形式如下：

类名 const 对象名[(参数表)];

或者

const 类名 对象名[(参数表)];

【例 4.18】非常对象和常对象的比较。

```
#include <iostream>
using namespace std;
class ConstObj{
public:
   int m;
   ConstObj(int i,int j)
   {
      m=i; n=j;
    }
   void set(int i){n=i;}
   void print()
   {  cout<<"m="<<m<<endl;
       cout<<"n="<<n<<endl;
   }
private:
    int n;
};
void main()
{
    ConstObj a(0,0);
    a.set(30);
    a.m=40;
    a.print();
}
```

在这个例子当中，对象 a 是一个普通的对象，而不是常对象，分析程序的结果为：

```
m=40
n=30
```

如果将程序中的对象 a 定义为常对象，将主函数修改为：

```
void main()
{
    const ConstObj a(0,0);
    a.set(30);
    a.m=40;
    a.print();
}
```

那么，程序编译时会出现如下的 3 个错误，第一个和第二个错误出现在 a.set(30)和 a.m=40 语句，C++不允许直接或间接地更改常对象的数据成员；第三个错误出现在 a.print()语句，C++不允许常对象调用普通的成员函数。

4.6.3 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 4.19】常数据成员举例。

```
#include <iostream>
using namespace std;
class ConstData{
public:
    ConstData(int y,int m,int d);
    void ShowData();
private:
   const int year;
   const int month;
   const int day;
};
//通过构造函数列表进行类的初始化
ConstData::ConstData(int y,int m,int d):year(y),month(m),day(d){  }
void ConstData::ShowData()
{
   cout<<year<<"."<<month<<"."<<day<<endl;
}
void main()
{
   ConstData data(2009,1,1);
   data.ShowData();
}
```

程序运行结果如下：

```
2009.1.1
```

注意：类成员初始化的时候，不能够在函数中进行简单的赋值操作，const 成员只能通过初始化列表进行赋值。

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。

【例 4.20】常成员函数的使用。

```
#include <iostream>
using namespace std;
class ConstData{
public:
    ConstData(int y,int m,int d);
    void ShowData();
```

```
    void ShowData() const;
private:
   const int year;
   const int month;
   const int day;
};
//通过构造函数列表进行类的初始化
ConstData::ConstData(int y,int m,int d):year(y),month(m),day(d){ }
void ConstData::ShowData()
{
   cout<<"ShowData1"<<endl;
   cout<<year<<"."<<month<<"."<<day<<endl;
}
void ConstData::ShowData() const
{
   cout<<"ShowData2"<<endl;
   cout<<year<<"."<<month<<"."<<day<<endl;
}
void main()
{
   ConstData date1(2008,1,1); date1.ShowData();
   const  ConstData  date2(2009,1,1); date2.ShowData();
}
```


程序运行结果如下：

```
ShowData1
2008.1.1
ShowData2
2009.1.1
```

在使用常成员函数时要注意：

（1）const 是函数类型的一个组成部分，因此在函数实现部分也要带有 const 关键字。

（2）常成员函数不更新对象的数据成员，也不能调用该类中没有用 const 修饰的成员函数。

（3）常对象只能调用它的常成员函数，而不能调用其他成员函数。成员函数与对象之间的操作关系如表 4-1 所示。

表 4-1　成员函数与对象之间的操作关系

成员函数 / 对象	常成员函数	一般成员函数
常对象	√	×
一般对象	√	√

（4）const 关键字可以用于参与重载函数的区分。例如：

```
void Print();
void Print() const;
```

这两个函数可以用于重载。重载的原则是：常对象调用常成员函数，一般对象调用一般成员函数。

4.7 程序举例

【实例 1】编写一个程序，输入 N 个学生数据，包括学号、姓名、成绩，要求输出这些学生数据并计算平均分。

分析：设计一个学生类 CStudent，除了包括 ID（学号）、name（姓名）和 score（成绩）数据成员外，有三个静态变量 number、sum 和 average，分别存放学生人数、学生的总成绩和学生的平均成绩，普通成员函数 print()输出姓名、编号和成绩等信息，静态成员函数 Aver()用于计算平均分，友元函数 show()输出学生的总人数和全部成绩，在 main()函数中定义了一个对象数组用于存储输入的学生数据。

```
#include <iostream>
using namespace std;
class CStudent{
private:
   char* name; //学生姓名
   char* ID; //学号
   float score; //学生成绩
   static int number;//学生人数
   static float sum;//学生的总成绩
   static float average; //学生的平均成绩
public:
    CStudent(char* s,char * n,float a);
    static void Aver();//求平均成绩
    void print(); //输出姓名、编号和成绩等信息
    friend void show(CStudent &);//友元函数输出学生的数量和成绩
    ~CStudent()
   {
    delete  [] name;
    delete [] ID;
    number--;
    sum-=score;
   }
};
CStudent::CStudent(char* s,char * n,float a)
{
   name=new char[strlen(s)+1];
   strcpy(name,s);
   ID=new char[strlen(n)+1];
   strcpy(ID,n);
   score=a;
   number++; //学生人数增加
   sum+=score; //累加总成绩
```

```
}
void CStudent::print()
{
   cout<<"学生姓名："<<name<<endl;
   cout<<"学生编号："<<ID<<endl;
   cout<<"学生成绩："<<score<<endl;
}
void show(CStudent &student)
{
   cout<<"学生的总人数:"<<CStudent::number<<endl;//非静态成员函数访问静态成员
   cout<<"学生的总成绩"<<CStudent::sum<<endl;
}
void CStudent::Aver()
{
   average=sum/number;
   cout<<"学生的平均成绩："<<average<<endl;
}
//静态数据成员初始化
int CStudent::number=0;
float CStudent::sum=0;
float CStudent::average=0;
void main()
{
   CStudent student[4]={CStudent("张三","0809001",80),CStudent("李四",
   "0809002",90),CStudent("王五","0809003",90),CStudent("马六","0809004",80)};
    for (int i=0;i<4;i++)
    {
       student[i].print();
    }
    show(student[4]);
    student[4].Aver();
}
```


程序的运行结果：

```
学生姓名：张三
学生编号：0809001
学生成绩：80
学生姓名：李四
学生编号：0809002
学生成绩：90
学生姓名：王五
学生编号：0809003
学生成绩：90
学生姓名：马六
学生编号：0809004
学生成绩：80
学生的总人数:4
```

学生的总成绩 340
学生的平均成绩：85

【实例 2】参考本书第 1 章 1.4 中的实例 1，利用本章讲到的友元类和友元函数，写一个能输出一年 12 个月的年历程序。

分析：在第 1 章 1.4 节的实例 1 中，日期使用结构类型，在此可以修改为包含一个友元类的类类型：

```
class Date
{
  public:
    int month;
    int day;
    int year;
  private:
    friend TdateType;          //定义 TdateType 类为 Date 类的友元类
};
```

对于 TdateType 类，作如下修改：

```
class TdateType
{
   public:
        TdateType();                  //不带参数的构造函数定义
        TdateType(Date &b);           //有参数的构造函数定义
        void Modify(int m = 10,int d = 1,int y = 2009);  //修改日期
        void GetIn(int y);
        int Weekday();                //判断是星期几成员函数定义
        void Print();                 //打印日期
   friend void show();                //声明 show()函数为类的友元函数
   private:
     Date a;                          //对象成员
     int IsLeapYear();                //成员函数,判断是否闰年
     int MonthEnd(int m);             //计算某月的天数
};
```

类的成员函数的定义：

```
TdateType::TdateType()
{
    a.year=1999;
    a.month=1;
    a.day=1;
}
TdateType::TdateType(Date &b)
{
    a.month = b.month;
    a.day = b.day;
    a.year = b.year;
}
void TdateType::Modify(int m,int d,int y)
```


```
{
    a.month = m;
    a.day = d;
    a.year = y;
}
void TdateType::GetIn(int y)
{
    a.year=y;
}
int TdateType::IsLeapYear()
{
    return ( (a.year % 4==0 && a.year % 100!=0 )||(a.year % 400==0) );
}
int TdateType::MonthEnd(int m)
{
    switch(m)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12: return 31;
        case 4:
        case 6:
        case 9:
        case 11:return 30;
         case 2:
           if (IsLeapYear())
               return 29;
           else
               return 28;
    }
    return 0;
}
int TdateType::Weekday()
{
    long n;
    n=((a.year)-1)*365;                  //直至去年的天数(不考虑闰年)
    n+=((a.year)-1)/4;                   //以下 3 条语句考虑闰年数
    n-= ((a.year)-1)/100;
    n+=((a.year)-1)/400;
    for ( int i=1;i<a.month;i++)         //本年直至上月的天数
       n+=MonthEnd(i);
    n +=a.day;                           //本月的天数
```

```
    n %=7;                                  //折算成星期几,若 0,则为星期日
    return n;
}
void TdateType::Print()
{
   for(int i=0;i<=6;i++)cout<<weekdays[i]<<"  ";
   cout<<endl;                              //输出星期
   char arr=32;int j=0;
   for(j=0;j<Weekday()*5;j++)cout<<arr;  //输出每月 1 号所在行之前的空格
   int array[31];                           //定义一个月的日期
   for(int k=1;k<=MonthEnd(a.month);k++)
   {
      int m=k+Weekday()-1;
      if(m%7==0&&m>6)  cout<<endl;    //日期超出周六后输出换行符
      array[k-1]=k;                         //向数组内输入日期
      cout<<array[k-1]<<"   ";          //输出日期之间的空格
      if(k<10)cout<<" ";                    //输出占一个字符的日期多余的空格
   }
   cout<<endl;
}
```

友元函数 show()的定义：

```
void show()
{
  TdateType d;
  for((d.a).month=1;(d.a).month<=12;(d.a).month++)
  {
    d.Modify((d.a).month,(d.a).day,(d.a).year);
    cout<<(d.a).month<<"月份："<<endl;
    d.Print();
  }
}
```

程序的主函数是：

```
int main()
{
  int year;
  TdateType Getdata;
  cout<<"请输入年份:"<<endl;
  cin>>year;
  Getdata.Modify();
  Getdata.GetIn(year);
  cout<<year<<"年的年历为:"<<endl;
  show();
  return 0;
}
```

注意，调试上述程序时，应该在定义类 Date 之前先声明 class TdateType 类，即增加如下语句：

```
class TdateType;
```

程序的执行结果是：

```
请输入年份：
2010
2010 年的年历为：
1 月份：
Sun  Mon  Tue  Wed  Thu  Fri  Sat
                        1    2
3    4    5    6    7    8    9
10   11   12   13   14   15   16
17   18   19   20   21   22   23
24   25   26   27   28   29   30
31
2 月份：
Sun  Mon  Tue  Wed  Thu  Fri  Sat
     1    2    3    4    5    6
7    8    9    10   11   12   13
14   15   16   17   18   19   20
21   22   23   24   25   26   27
28
3 月份：
Sun  Mon  Tue  Wed  Thu  Fri  Sat
     1    2    3    4    5    6
7    8    9    10   11   12   13
14   15   16   17   18   19   20
21   22   23   24   25   26   27
28   29   30   31
… …
```


（1）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，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，而且还有成员函数。

（2）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，既可以通过一个对象名访问对象，也可以通过对象地址来访问一个对象，对象指针就是一个用来存放对象地址的变量。

（3）函数中参数的传递有值传递和引用传递，那么对象作为一种自定义的数据类型，也同样可以作为参数在函数中传递。

（4）友元提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。对于一个类，可以利用关键字 friend 将一般函数、其他类的成员函数或者其他类声明为该类的友元，使得这个类中本来隐藏的信息（包括私有成员和保护成员）可以被友元访问。如果友元是一般成员函数或是类的成员函数，称为友元函数；如果友元是一个类，则称为

友元类，友元类的所有成员函数都成为友元函数。

（5）const 关键字不仅可以修饰类对象本身，也可以修饰类对象的成员函数和数据成员，分别称为常对象、常成员函数和常数据成员。

（6）静态成员包括静态数据成员和静态函数成员，静态成员只有一个拷贝，一个类的所有对象共享这个静态成员。

一、填空题

1．静态成员属于_____，非静态成员属于对象。

2．this 指针总是指向_____，当调用一个成员函数时，系统就自动把 this 指针传给该函数。

3．_____成员函数中不能直接引用类中说明的非静态成员。

4．设 A 为 test 类的对象且赋有初值，则语句 test B(A); 表示_____。

5．利用“对象名.成员变量”形式访问的对象成员仅限于被声明为___（1）___的成员；若要访问其他成员变量，需要通过___（2）___函数或___（3）___函数。

二、选择题

1．友元的作用是（　）。

A．实现多态性

C．加强了类的封装性

B．提高了代码的可重用性和可维护性

D．提高代码的运行效率

2．下列各类函数中，（　）不是类的成员函数。

A．析构函数　　B．构造函数

C．拷贝初始化构造函数　　D．友元函数

3．为了使类中的某个成员不能被类的对象通过成员操作符访问，不能把该成员的访问权限定义为（　）。

A．public　　B．protected

C．private　　D．static

4．关于静态成员的描述中，（　）是错误的。

A．静态成员可分为静态数据成员和静态成员函数

B．静态数据成员定义后必须在类体内进行初始化

C．静态数据成员初始化不使用其构造函数

D．静态数据成员函数中不能直接引用非静态成员

5．关于友元的描述中，（　）是错误的。

A．友元函数是成员函数，它被说明在类体内

B．友元函数可直接访问类中的私有成员

C．友元函数破坏封装性，使用时尽量少用

D．友元类中的所有成员函数都是友元函数

三、程序设计题

1．写出下列程序的输出结果。

```
#include<iostream>
using namespace std;
class Count
{  public:
    Count() { count++;}
    static int getn() {return count;}
    ~Count() { count--; }
private:
    static int count;
};
int Count::count=100;
void main()
{ Count c1,c2,c3,c4;
  cout<<Count::getn()<<endl;
}
```

_____

2．定义一个抽象类 shape 用以计算面积，从中派生出计算长方形、梯形、圆形面积的派生类。程序中通过基类由指针来调用派生类中的虚函数，计算不同形状的面积。