第3篇

基于对象的程序设计

第8章 类和对象

第9章 关于类和对象的进一步讨论

第10章 运算符重载

第9章 关于类和对象的进一步讨论

9.1 构造函数

9.1.1 对象的初始化

类的数据成员是不能在声明类时初始化的。

如果一个类中所有的成员都是公用的，则可以在定义对象时对数据成员进行初始化。如

```
class Time
{
public:                          //声明为公用成员
   hour; minute; sec;
};
Time t1={14,56,30};              //将t1初始化为14:56:30
```

如果数据成员是私有的，或者类中有private或protected的成员，就不能用这种方法初始化。

Constructor

1. Definition:

A special member function, it is primarily used to allocate memory for object and initialize the object as a particular state.

2. Characteristic:

- The name of constructor must be same as the class name, or else it will be regarded as a common member function.
- Recalled by the complier system when the object is defined.

Constructor

- It can have any type of parameters, but can't have returning type. So when defining and declaring a constructor, you can't point out its type, even the "void" type.
- It may be inline functions, overloading functions, functions without format parameters.
- In practice, you usually need define a constructor for each class. If you haven't define a constructor, the complier will automatically create a constructor without a parameters.

9.1.2 构造函数的作用

构造函数(constructor)是一种特殊的成员函数，不需要也不能被用户来调用它，而是在建立对象时自动执行。

构造函数的名字必须与类名同名。

它不具有任何类型，不返回任何值。

构造函数的功能是初始化。

如果用户自己没有定义构造函数，则C++系统会自动生成一个空的构造函数。

Example of constructor

```
#include <iostream>
using namespace std;
class Time
{
public:
   Time( )                    //the definition of constructor
   {
      hour=0;
      minute=0;
      sec=0;
   }
   void set_time( );
   void show_time( );
private:
   int hour;                   // private data members
   int minute;
   int sec;
};
```

```
void Time::set_time()      //定义成员函数，向数据成员赋值
{
    cin>>hour;
    cin>>minute;
    cin>>sec;
}
void Time::show_time()      //定义成员函数，输出数据成员的值
{   cout<<hour<<":"<<minute<<":"<<sec<<endl;}
int main()                  //The invoke of constructor
{
    Time t1;                //隐含调用构造函数
    t1.show_time( );        //显示t1的数据成员的值
    t1.set_time( );         //对t1的数据成员赋值
    t1.show_time( );        //显示t1的数据成员的值
    return 0;
}
```

9.1.3 Constructor with parameters

```
//求长方柱的体积
#include <iostream>
using namespace std;
class Box
{
public:
    Box(int, int, int);        //the declaration of constructor
    int volume( );
private:
    int height;
    int width;
    int length;
};
//define constructor outside of the class
Box::Box(int h,int w,int len)   //the realization of constructor
{
    height=h;
    width=w;
    length=len;
}
```

```
//Box::Box(int h,int w,int len):height(h)， width(w)，
    length(len){ }//用参数初始化表实现初始化，9.1.4
int Box::volume()            //定义计算体积的函数
{
    return(height*width*length);
}

int main( )
{
    Box box1(12,25,30);          //隐含调用构造函数，将初始值作为实参
    cout<<"The volume of box1 is "<<box1.volume( )<<endl;
    Box box2(15,30,21);          //隐含调用构造函数，将初始值作为实参
    cout<<"The volume of box2 is "<<box2.volume( )<<endl;
    return 0;
}
```

9.1.5 Overloading of constructors

- Similar to a common function, the constructors are also can be overloaded.
- The constructors which have similar function ,but different type and number of parameters, can be assigned the same function name.
- The overload constructors can be respectively recalled according to their parameters' type and number by the complier.

例9.3 构造函数的重载

```
#include <iostream>
using namespace std;
class Box
{
public:
    Box( );                                  //声明一个无参的构造函数
    Box(int h,int w,int len):height(h),width(w),length(len){ }
    //声明一个有参的构造函数，用参数的初始化表对数据成员初始化
    int volume( );
private:
    int height;
    int width;
    int length;
};
Box::Box()                                   //定义一个无参的构造函数
{
    height=10;
    width=10;
    length=10;
}
```

```
int Box::volume( )
{
   return(height*width*length);
}

int main( )
{
   Box box1;                          //建立对象box1,不指定实参
   cout<<"The volume of box1 is "<<box1.volume( )<<endl;
   Box box2(15,30,25);                //建立对象box2,指定3个实参
   cout<<"The volume of box2 is "<<box2.volume( )<<endl;
   return 0;
}
```

说明：

(1) 无参的构造函数称为默认构造函数(default constructor)。一个类只能有一个默认构造函数。

(2) 如果在建立对象时选用无参构造函数，应注意定义对象的语法。

Box box1;　　// Box box1()则是错误的;

(3) 在一个类中可以包含多个构造函数，但建立对象时只执行其中一个构造函数。

9.1.6 Constructor with default parameters

- As to the constructor with default parameters, you should transfer parameters to constructors when defining the object, or the constructor can't be executed. But in reality, there are several constructors whose parameters are usually invariable

9.1.6 使用默认参数的构造函数

```
#include <iostream>
using namespace std;
class Box
{
public:
   Box(int =10,int =10,int =10);   //形参有默认值，形参名可以省略
   int volume( );
private:
   int height;
   int width;
   int length;
};
Box::Box(int h,int w,int len)       //在定义函数时可以不指定默认参数
{
   height=h;
   width=w;
   length=len;
}//Box::Box(int h,int w,int len):height(h),width(w),length(len){ }
```

```
int Box::volume( )
{   return(height*width*length);}
int main( )
{
   Box box1;                  //没有给实参
   cout<<"The volume of box1 is "<<box1.volume( )<<endl;
   Box box2(15);              //只给定一个实参
   cout<<"The volume of box2 is "<<box2.volume( )<<endl;
   Box box3(15,30);           //只给定2个实参
   cout<<"The volume of box3 is "<<box3.volume( )<<endl;
   Box box4(15,30,20);        //给定3个实参
   cout<<"The volume of box4 is "<<box4.volume( )<<endl;
   return 0;
}
```

Note:

When overloading the none-parameters' constructors and constructors with default parameters, the program will bring about duality. This situation must be avoided in practice.

函数重载与带默认参数的函数

```
#include <iostream>
using namespace std;
int main( )
{
    int max(int a=5, int b=6, int c=7);//形参有默认值，形参名可以省略
    //int max(int , int );
    float max(float a=10.1,float b=20.3,float c=30.3);//形参有默认值，形参名可以省略
    //float max(float a=10.1,float b=20.3);
    float a,b,c;
    //double a,b,c;
    int d;
    a=10.1;
    b=15.2;
    c=30.2;
    cout<<"max(a,b,c)="<<max(d)<<endl;   //输出3个数中的最大者
    cout<<"max(a,b)="<<max(a,b)<<endl;   //输出2个数中的最大者
    return 0;
}
```

函数重载与带默认参数的函数

```
int max(int a,int b,int c)
{
    if(b>a) a=b;
    if(c>a) a=c;
    return a;
}
int max(int a,int b)
{
    if(b>a) a=b;
    //if(c>a) a=c;
    return a;
}
```

```
float max(float a,float b,float c)
{
    if(b>a) a=b;
    if(c>a) a=c;
    return a;
}
float max(float a,float b)
{
    if(b>a) a=b;
    //if(c>a) a=c;
    return a;
}
```

Destructor

- **1. Definition:**
- **Destructor is a special member functions, and it perform opposite to the constructor, including the following tasks:**
  - **Complete some cleaning work before the object being deleted.**
  - **Automatically recall the destructor by complier system when the survival period ended, and then free up all resource belonging to this object, such as memory space.**
- **If destructor hasn't been declared in the program, the complier will automatically create a default destructor.**

Destructor

- **2. characteristics:**
- **The name of destructor is the same as constructor except with a "~" before it;**
- **Destructor has no parameters and no returning value, and can't be overload. So there is only a destructor within a class.**

9.2 析构函数

析构函数(destructor) 作用不是删除对象，而是在撤销对象占用的内存之前完成一些清理工作，使这部分内存可以被程序分配给新对象使用。

一个类只能有一个析构函数

析构函数不返回任何值，没有函数类型，也没有函数参数。不能被重载。

它的名字是类名的前面加一个“～”符号。如

~Time( )

如果用户没有定义析构函数，C++编译系统会自动生成一个空的析构函数。

当对象的生命期结束时，会自动执行析构函数。

①自动局部对象在对象释放前自动执行析构函数。

②static局部对象在main函数结束或调用exit函数结束程序时，调用析构函数。

③全局对象在程序的流程离开其作用域时(如main函数结束或调用exit函数) 时，调用该全局对象的析构函数。

④用new建立的动态对象，当用delete运算符释放该对象时，调用该对象的析构函数。

例9.5 包含构造函数和析构函数的程序

```
#include<string>
#include<iostream>
using namespace std;
class Student                                          //声明Student类
{
public:
   Student(int n,string nam,char s )                   //定义构造函数
   {
       num=n;
       name=nam;
       sex=s;
       cout<<"Constructor called."<<endl;          //输出提示信息
   }
   ~Student()                                          //定义析构函数
    {   cout<<"Destructor called."<<"  " <<num<<endl;   } //输出提示信息
```

```
    void display()                    //定义成员函数
    {
        cout<<"num: "<<num<<endl;
        cout<<"name: "<<name<<endl;
        cout<<"sex: "<<sex<<endl<<endl;
    }
private:
    int num;
    string name;
    char sex;
};
```

```
int main( )
{
   Student stud1(10010,"Wang_li",'f');        //建立对象stud1
   stud1.display( );                          //输出学生1的数据
   Student stud2(10011,"Zhang_fun",'m');   //定义对象stud2
   stud2.display( );                          //输出学生2的数据
   return 0;
}
```

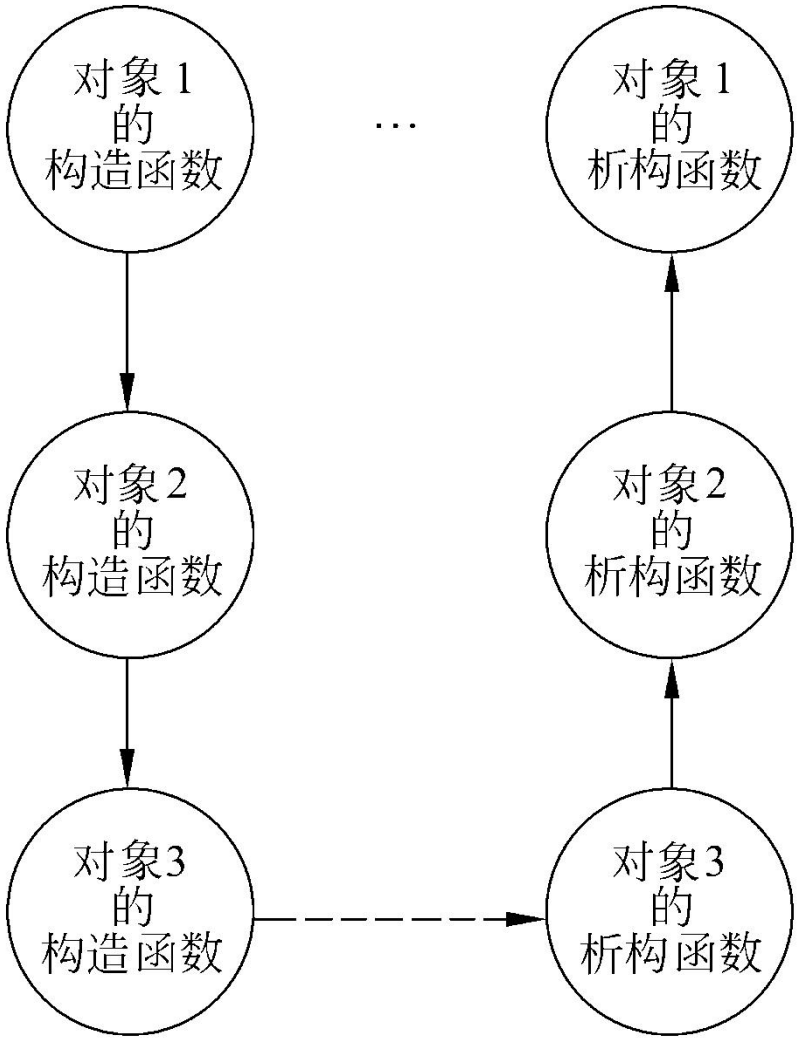
9.3 调用构造函数和析构函数的顺序

先进后出

(1) 全局对象的构造函数在文件中的所有函数(包括main函数)执行之前调用。

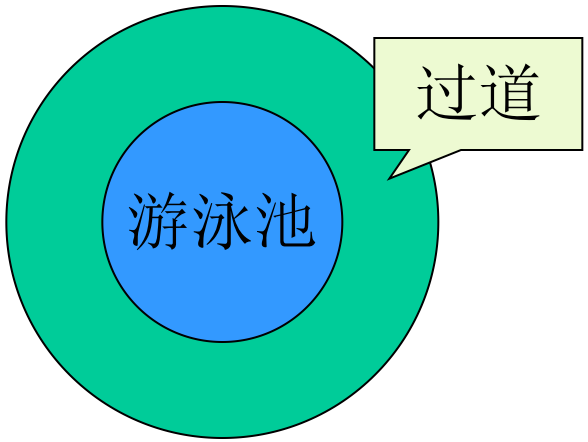
(2) 局部自动对象，则在建立对象时调用其构造函数。

(3) 静态(static)局部对象，则只在程序第一次调用此函数建立对象时调用构造函数。



Example of class application

一圆型游泳池如图所示，现在需在其周围建一圆型过道，并在其四周围上栅栏。栅栏价格为35元/米，过道造价为20元/平方米。过道宽度为3米，游泳池半径由键盘输入。要求编程计算并输出过道和栅栏的造价。



```
#include <iostream.h>

const float PI = 3.14159;
const float FencePrice = 35;
const float ConcretePrice = 20;

//声明类Circle 及其数据和方法
class Circle
{
private:
   float   radius;

public:
   Circle(float r);  //构造函数

   float Circumference()const; //圆周长
   float Area()const;  //圆面积
};
```

```
// 类的实现
// 构造函数初始化数据成员radius
Circle::Circle(float r)
{radius=r}

// 计算圆的周长
float Circle::Circumference()const
{
   return 2 * PI * radius;
}

// 计算圆的面积
float Circle::Area()const
{
   return PI * radius * radius;
}
```

```
int main ()
{
    float radius;
    float FenceCost, ConcreteCost;

    // 提示用户输入半径
    cout<<"Enter the radius of the pool: ";
    cin>>radius;

    // 声明 Circle 对象
    Circle Pool(radius);
    Circle PoolRim(radius + 3);
```

```
// 计算栅栏造价并输出
FenceCost = PoolRim.Circumference()* FencePrice;
cout << "Fencing Cost is ￥" << FenceCost << endl;

// 计算过道造价并输出
ConcreteCost = (PoolRim.Area()- Pool.Area
（）)*ConcretePrice;
cout << "Concrete Cost is ￥" << ConcreteCost << endl;
return 0;
}
```

9.4 对象数组

数组的每一个元素都是同类的对象。

Student stud[50];

如果构造函数只有一个参数，在定义数组时可以直接在等号后面的花括号内提供实参。如

Student stud[3]={60,70,78};

有多个参数的构造函数在定义对象数组时的初始化方法：

```
Student:: Student(int=1001,int=18,int=60);//形参名可以省略
Student stud[3]={15,60,70};//歧义
Student stud[3]={15,60,70, 15,60,70, 15,60,70};
//error: too many initializers
```

原因：编译系统只为每个对象元素的构造函数传递一个实参。

正确写法：在花括号中分别写出构造函数并指定实参。

```
Student Stud[3]={
Student(1001,18,87),
Student(1002,19,76),
Student(1003,18,72)
};
```

例9.6 对象数组的使用方法

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int h=10,int w=12,int len=15):
height(h),width(w),length(len){ }
int volume( );
private:
int height;
int width;
int length;
};
```

```
int Box::volume( )
{return(height*width*length);}
int main( )
{
   //Box a[3]={10,20,30};
   //Box a[3]={10,12,15,15,18,20,16,20,26}; //无构造函数,public成员可以这样使用
   Box a[3]={  Box(10,12,15),
               Box(15,18,20),
               Box(16,20,26)}; //定义对象数组
   cout<<"volume of a[0] is "<<a[0].volume( )<<endl;
   cout<<"volume of a[1] is "<<a[1].volume( )<<endl;
   cout<<"volume of a[2] is "<<a[2].volume( )<<endl;
   return 0;
}
```

```
//对象数组中字符串的初始化
#include<string>
#include<iostream>
using namespace std;
class Student                                          //声明Student类
{
public:
//              Student(int n,string nam,char s )                   //定义构造函数
                Student(int n,char nam[],char s )                   //定义构造函数
//              Student(int n,char* nam,char s )                    //定义构造函数
   {
                                num=n;
                                strcpy(name,nam);                   //仅用于数组
//                              name=nam;
                                sex=s;
                                cout<<"Constructor called."<<endl;  //输出提示信息
   }
   ~Student()                                          //定义析构函数
   {            cout<<"Destructor called."<<"  "<<num<<endl;}       //输出提示信息
   void display()                                      //定义成员函数
   {
                                cout<<"num: "<<num<<endl;
                                cout<<"name: "<<name<<endl;
                                cout<<"sex: "<<sex<<endl<<endl;
   }
private:
   int num;
//              string name;        //重点关注字符串的处理
                char name[20];      //重点关注字符串的处理
//              char* name;                                 //重点关注字符串的处理
   char sex;
};
//Student sss[10];
int main( )
{//Student stud[]={{10010,"Wang_li",'f'},{10011,"Zhang_fun",'m'}};//无构造函数可以这样使用
  Student stud[]={
                                Student(10010,"Wang_li",'f'),
                                Student(10011,"Zhang_fun",'m')};
                for(int i=0;i<2;i++)
                                stud[i].display( );     //输出学生的数据
   return 0;
}
```

为什么使用string变量多执行一次析构函数？

```
//对象数组中字符串的初始化
#include<string>
#include<iostream>
using namespace std;
class Student                                   //声明Student类
{
public:
Student(Student &b )//含string子对象的数组，初始化调用复制构造函数
  {
//num=b.num;
//name=b.name;
//sex=b.sex;
cout<<"Copy Constructor called."<<endl;//输出提示信息
  }
//          Student(int n,string nam,char s )                    //定义构造函数
            Student(int n,char nam[],char s )                    //定义构造函数
//          Student(int n,char* nam,char s )                     //定义构造函数
  {
                        num=n;
                        strcpy(name,nam);                        ////仅用于数组
//                      name=nam;
                        sex=s;
                        cout<<"Constructor called."<<endl;       //输出提示信息
  }
  ~Student()                                    //定义析构函数
  {     cout<<"Destructor called."<<"  "<<num<<endl;}            //输出提示信息
  void display()                                //定义成员函数
  {
                        cout<<"num: "<<num<<endl;
                        cout<<"name: "<<name<<endl;
                        cout<<"sex: "<<sex<<endl<<endl;
  }
private:
  int num;
//          string name;        //重点关注字符串的处理
            char name[20];      //重点关注字符串的处理
//          char* name;                       //重点关注字符串的处理
  char sex;
};
//Student sss[10];
int main( )
{//Student stud[]={{10010,"Wang_li",'f'},{10011,"Zhang_fun",'m'}};//无构造函数可以这样使用
  Student stud[]={
                        Student(10010,"Wang_li",'f'),
                        Student(10011,"Zhang_fun",'m')};
  for(int i=0;i<2;i++)
                        stud[i].display( );     //输出学生的数据
```

对象数组初始化调用复制构造函数?

```
#include <cstring>
#include <iostream>
using namespace std;
class example
{
   private:
      char *s;
   public:
      example(){cerr<<"constructor called"<<endl;}
      example(const char *);
      example(example const& ano){cerr<<"copy constructor called"<<endl;}
      ~example();
      void show() {cout << s << endl;};
};
example::example(const char *p)
{
  cerr<<"construct "<<this<<endl;
  s = new char[strlen(p) + 1];
  strcpy(s,p);
}
example::~example()
{
  cerr<<"delete "<<this<<endl;
  delete [] s;
}
//example exx[4];
int main()
{
   example ex[4] = {example("hello"),example("hi"),example("yeah"),example("good")};
   for (int i = 0; i < 4; i++)
      ex[i].show();
   return 0;
}
```

9.5 对象指针

9.5.1 指向对象的指针

对象空间的起始地址就是对象的指针。
定义对象指针变量的一般形式为
类名 *对象指针名;
Time t1; Time *pt =&t1;
可以通过对象指针访问对象和对象的成员。如

```
(*pt).hour            //即t1.hour
pt->hour              //即t1.hour
(*pt).get_time ()     //即t1.get_time
pt->get_time ()       //即t1.get_time
```

9.5.2 指向对象成员的指针

1. 指向对象数据成员的指针

定义指向对象数据成员的指针的一般形式为

数据类型名 *指针变量名;

```
int *p1;
```

如果Time类的数据成员hour为公用的整型数据，则可以在类外通过指向对象数据成员的指针变量访问对象数据成员hour。

```
p1=&t1.hour;
cout<<*p1<<endl;          //输出t1.hour的值
```

2. 指向对象成员函数的指针

定义指向成员函数的指针变量的形式：

void (Time::*p)( );　　//定义p2为指向Time类中公用成员函数的指针变量

要求指明：

①函数参数的类型和参数个数；

②函数返回值的类型；

③所属的类。

指针p可以指向一个公用成员函数，如

p=&Time::get_time;

例9.7 有关对象指针的使用方法

```
#include <iostream>
using namespace std;
class Time
{
public:
    Time(int,int,int);
    int hour;
    int minute;
    int sec;
    void get_time( );              //声明公有成员函数
};
Time::Time(int h,int m,int s)
{hour=h;minute=m;sec=s;}

void Time::get_time()              //定义公有成员函数
{ cout<<hour<<":"<<minute<<":" <<sec<<endl;}
```

```
int main( )
{
    Time t1(10,13,56);
     int *p1=&t1.hour;
     cout<<*p1<<endl;
     t1.get_time( );
     Time *p2=&t1;
     p2->get_time( );
     void (Time::*p3)( );
     p3=&Time::get_time;
     (t1.*p3)( );
    return 0;
}
```

9.5.3 this 指针

Box a(10,10,30), b (10,20,50) ;

a. volume( );

b. volume( );

如何保证volume ( )引用的是指定的对象的数据成员？

在每一个成员函数中都包含一个特殊的指针“this”，是指向本类对象的指针，指向当前调用成员函数的对象。

例如成员函数volume的定义如下：

```
int Box::volume( )
{return (height*width*length);}
```

C++把它处理为

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

//this作为成员函数的隐含参数

在调用成员函数a.volume()时，实际上是用以下方式调用的：

```
a.volume(&a);
```

a.volume(&a);

将对象a的地址传给形参this指针。this指向对象a，成员函数按照this的指向找到对象a的数据成员。

volume函数实际上是执行：

(this->height)*(this->width)*(this->length)

由于当前this指向a，因此相当于：

(a.height)*(a.width)*(a.length)

9.6 共用数据的保护

既要使数据能在一定范围内共享，又要保证它不被任意修改，可以使用**const**，把有关的数据定义为常量。

9.6.1 常对象

凡希望保证数据成员不被改变的对象，可以声明为常对象。

定义常对象的一般形式为

类名 const 对象名[(实参表列)];

或

const 类名 对象名[(实参表列)];

Time const t1(12,34,46);　　//t1是常对象常对象必须要有初值

常对象的非const型的成员函数(除了构造函数和析构函数)不能引用常对象中的数据成员。

例如，

```
t1.get_time( );        //非法，防止函数修改常对象数据成员
void get_time()const;        //将函数声明为const
```

常成员函数可以访问但不能修改常对象中的数据成员。

若一定要修改常对象中的某个数据成员的值，应将该数据成员声明为可变的数据成员，如

```
mutable int count;
```

这样就可以用声明为const的成员函数来修改它的值。

9.6.2 常对象成员

1. 常数据成员

```
const int hour;                    //声明hour为常数据成员
```

常数据成员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。

在类外定义构造函数，应写成以下形式：

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

常对象的数据成员都是常数据成员，因此常对象的构造函数只能用参数初始化表对常数据成员进行初始化。

2. 常成员函数

常成员函数只能引用而不能修改本类中的数据成员，常用于输出数据等。如

```
void get_time()const;          //注意const的位置
```

常成员函数可以引用const数据成员，也可以引用非const数据成员。

常数据成员可以被const成员函数引用，也可以被非const成员函数引用。

常成员函数不能调用另一个非const成员函数。

说明

(1) 如果部分数据成员的值不允许改变，则可以将其声明为const，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。

(2) 如果所有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。

9.6.3 指向对象的常指针

将指针变量声明为const型，指针始终指向同一个对象。如

```
Time * const ptr1 =&t1;      //const的位置在指针变量名前面
ptr1=&t2;          //错误，ptr1不能改变指向
```

注意： 常指针变量的值始终不变，但可以改变其所指向对象(如t1)的值。

往往用常指针作为函数的形参，目的是不允许在函数执行过程中改变指针变量的值，使其始终指向原来的对象。

9.6.4 指向常对象的指针变量

指向常变量的指针变量

const char *ptr;//注意const的位置

(1) 如果一个变量已被声明为常变量，只能用指向常变量的指针变量指向它。

(2) 指向常变量的指针变量可以指向非const变量。此时不能通过此指针变量改变该变量的值。

(3) 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。

如果函数的形参是指向const型变量的指针，实参可以是指向const变量或非const变量的指针。

指向常对象的指针变量

(1) 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它。

(2) 如果定义了一个指向常对象的指针变量，并使它指向一个非const的对象，则其指向的对象是不能通过指针来改变的。

(3) 指向常对象的指针最常用于函数的形参，目的是在保护形参指针所指向的对象，使它在函数执行过程中不被修改。

9.6.5 对象的常引用

一个变量的引用就是变量的别名。

实质上，变量名和引用名都指向同一段内存单元。

如果形参为变量的引用名，实参为变量名，则在调用函数时，把实参变量的地址传给形参(引用名)，这样引用名也指向实参变量。

如果不希望在函数中修改实参的值，可以把引用变量声明为const(常引用)，函数原型为

```
void fun(const Time &t);
```

在C++面向对象程序设计中，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，又可以提高程序运行效率。

例9.8 对象的常引用

```
#include <iostream>
using namespace std;
class Time
{
public:
    Time(int,int,int);
    int hour;
    int minute;
    int sec;
};
Time::Time(int h,int m,int s)  //定义构造函数
{
    hour=h;
    minute=m;
    sec=s;
}
```

```
void fun(Time &t)              //形参t是Time类对象的引用
{t.hour=18;}

//void fun(const Time &t) //形参t是Time类对象的常引用
//{t.hour=18;}  //error: l-value specifies const object

int main( )
{
    Time t1(10,13,56);         // t1是Time类对象
    fun(t1);                   //实参是Time类对象，可以通过引
用来修改实参t1的值
    cout<<t1.hour<<endl;  //输出t1.hour的值为18
    return 0;
}
```

9.6.6 const型数据的小结

形式	含义
Time const t1;	t1是常对象，其值在任何情况下都不能改变
void Time::fun( )const	fun是Time类中的常成员函数，可以引用，但不能修改本类中的数据成员
Time * const p;	p是指向Time对象的常指针，p的值(即p的指向)不能改变
const Time *p;	p是指向Time类常对象的指针，其指向的类对象的值不能通过指针来改变
Time &t1=t;	t1是Time类对象t的引用，二者指向同一段内存空间

9.7 对象的动态建立和释放

用前面介绍的方法定义的对象是静态的，在程序运行过程中，对象所占的空间是不能随时释放的。

用new运算符动态地分配内存，用delete运算符释放这些内存空间。

这也适用于对象，可以用new运算符动态建立对象，用delete运算符撤销对象。

9.7 对象的动态建立和释放

*7.1.7 运算符new和delete

new运算符的例子：

new int;　　//开辟一个存放整数的存储空间，返回一个指向该存储空间的地址(即指针)

new char[10];　　//开辟一个存放字符数组(包括10个元素)的空间，返回首元素的地址

float *p=new float(3.14159); //开辟一个存放单精度数的空间，并指定该实数的初值为3.14159，将返回的该空间的地址赋给指针变量p

Box *pt=new Box(12,15,18);

new运算符使用的一般格式为

new 类型 [初值]

用new分配数组空间时不能指定初值。

如果由于内存不足等原因而无法正常分配空间，则new会返回一个空指针NULL（0），用户可以根据该指针的值判断分配空间是否成功。

delete运算符使用的一般格式为

delete [ ] 指针变量

例如

delete p;

对于用“new char[10];”开辟的字符数组空间，则为：

delete [] p;

若指针pt先后指向不同的动态对象，应避免错删对象。

用new建立的动态对象一般不用对象名，只能通过指针访问。

例 动态数组的使用

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
    char *p,*p1;
    if ((p=new char[10])==0 ) cout<<"Not enough memory";
    //p="Wang Fun";
    p1=p;
    for(int i=0;i<10;i++)
            *p++=65+i;
    p=p1;
    //cout<<p<<endl;
    for( i=0;i<10;i++)
            cout<<*p++;
    cout<<endl;
    p=p1;
    delete p;        //撤销该空间
    return 0;
}
```

例 动态对象的使用

```
#include <iostream>
using namespace std;
class Time
{
public:
    Time(){hour=0;minute=0;sec=0;}
    Time(int,int,int);
    int hour;
    int minute;
    int sec;
};
Time::Time(int h,int m,int s)  //定义构造函数
{
    hour=h;
    minute=m;
    sec=s;
}
```

```
int main( )
{
    Time *p，t1;                    //定义指向Time类型指针变量p
    p=new Time;          //用new运算符开辟一个存放Time类型数据
    的空间
    p->hour=“15”;       //向Time类对象的成员赋值
    p->minute=10;
    p->sec=‘30';
    cout<<p-> hour<<":"<<p->minute<<":"
          <<p->sec<<endl;       //输出各成员的值
    //Time t2(p);
    delete p;                       //撤销该空间
    return 0;
}
```

9.8 对象的赋值和复制

9.8.1 对象的赋值

同类对象之间可以互相赋值。

对象之间的赋值通过“=”进行，这是通过运算符重载实现的。

赋值过程是通过成员复制来完成的，即将一个对象的数据成员值一一复制给另一对象的对应成员。对象赋值的一般形式为

对象名1 = 对象名2;

例9.9 对象的赋值

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int=10,int=10,int=10);        //声明有默认参数的构造函数
int volume( );
private:
int height;
int width;
int length;
};
Box::Box(int h,int w,int len)
{height=h;
width=w;
length=len;
}
```

```
int Box::volume( )
{return(height*width*length);     //返回体积
}

int main( )
{Box box1(15,30,25),box2;
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
box2=box1;                                  //将box1的值赋给box2
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
return 0;
}
//类的数据成员中不能包括动态分配的数据
```

9.8.2 对象的复制

用一个已有的对象快速地复制出多个完全相同的对象。如

Box box2(box1);

其一般形式为

类名 对象2(对象1);

用对象1复制出对象2。

对象的复制与对象的定义相似，但参数是对象。

在建立对象时调用一个特殊构造函数——复制构造函数(copy constructor)：

```
//The copy constructor definition.
Box::Box(const Box& b)
{ height=b.height;width=b.width;length=b.length;}
```

复制构造函数也是构造函数，它只有一个参数，这个参数是本类对象的引用。

Box box2(box1);

由于实参是对象，因此编译系统就调用复制构造函数(它的形参也是对象)。

实参box1的地址传递给形参b(b是box1的引用)，因此执行复制构造函数的函数体时，将box1对象中各数据成员的值赋给box2中各数据成员。

如果用户自己未定义复制构造函数，则编译系统会自动提供一个默认的复制构造函数，其作用只是简单地复制类中每个数据成员。

C++还提供另一种复制形式，用赋值号代替括号，如

```
Box box2=box1;          //用box1初始化box2
```

其一般形式为

类名 对象名1 = 对象名2;

可以在一个语句中进行多个对象的复制。如

```
Box box2=box1,box3=box2;
```

其作用都是调用复制构造函数。

请注意普通构造函数和复制构造函数的区别。

（1） 在形式上

类名(形参表列);

类名(类名& 对象名);

（2） 被调用时机

普通构造函数在程序中建立对象时被调用。

复制构造函数在用已有对象复制一个新对象时被调用。

复制对象的时机：

① 程序中需要新建立一个对象，并用另一个同类的对象对它初始化。

② 函数的参数为类的对象。在调用函数时需要将实参对象完整地传递给形参，也就是需要建立一个实参的拷贝，这就是按实参复制一个形参。

③ 函数的返回值是类的对象。在函数调用完毕将返回值带回函数调用处时。此时需要将函数中的对象复制一个临时对象并传给该函数的调用处。

Copy-constructor

1.Definition:

Copy-constructor is a special constructor, its format parameter is a reference of the self-class object.

2.Definition Method:

```
class 类名
{ public :
      类名（形参）；//构造函数
      类名（类名 &对象名）； //拷贝构造函数
        ...
};
类名:: 类名（类名 &对象名）//拷贝构造函数的实现
{   函数体   }
```

调用方法：

类名 对象名（对象名）

Copy-constructor

Note:

- **The name of copy-constructor is the same as class name, and can't assign it any type;**
- **Copy-constructor only has a parameter, and the parameter must be the reference of self-class object;**
- **If the user hasn't define a copy-constructor (that is self-define copy-constructor), you recall the default copy-constructor when defining a object.**

Examples1 of copy-constructor

```
class Point
{
  public:
     Point(int xx=0,int yy=0){X=xx; Y=yy;}
     Point(Point&  p);
     int GetX() {return X;}
     int GetY() {return Y;}
  private:
     int  X,Y;
};
```

```
Point::Point (Point&  p)
{
    X=p.X;
    Y=p.Y;
    cout<<"拷贝构造函数被调用"<<endl;
}
int main()
{
   Point  A(3,2),D;
   Point  B(A);  //拷贝构造函数的调用
   Point  C=A;  //拷贝构造函数的调用
   D=A;
   cout<<D.GetX()<<"  "<<D.GetY()<<endl;
   return 0;
}
```

9.9 静态成员

数据共享：

全局变量；

静态局部变量；

全局对象；

静态数据成员；

Static member

1. Note:

Static member is similar to static variable. But it doesn't belong to a certain object, but belongs to the whole class and shared by all objects of the class. Static member has two kinds: Static member data 、Static member function.

2. Static member data:

Declared by keyword "Static".

Used by all objects belonging to the class, so only updated once , all the static member data of all objects can be updated simultaneously, and its value is the same.

Initialized outside of the class with "类型 类名::静态数据成员=初始值"

3. Static member function:

- Declared by keyword “Static”.
- Used by all objects belonging to the class.
- Similar to a common member function, it can be declared and realized either within the class, or outside of the class.
- Different from a common member function, it can be cited either directly, or through the object.
- Citing Method:

  直接引用： 类名 静态成员函数(参数表)

  通过对象间接引用：

  对象名.静态成员函数名(参数表)

Note：A common member function must be cited as follows:

对象名.成员函数名(参数表)

9.9.1 静态数据成员

静态数据成员：

```
class Box
{public:
int volume( );
private:
static int height;        //height定义为静态的数据成员
int width;
int length;
};
```

说明：

(1) 静态数据成员不属于某一个对象，静态数据成员是在所有对象之外单独开辟空间。

(2) 静态数据成员在程序编译时被分配空间的，到程序结束时才释放空间。

(3) 静态数据成员只能在类外进行初始化。如

```
static int Box::height=10;          //表示对Box类中的数据成员初始化
```

(4) 静态数据成员既可以通过对象名引用，也可以通过类名来引用。

例9.10 引用静态数据成员

```
#include <iostream>
using namespace std;
class Box
{
public:
    Box(int,int) {width=w;length=len;}
    int volume(){return(height*width*length);}

    static int height;
    int width;
    int length;
};
```

```
int Box::height=10;                    //对静态数据成员height初始化

int main( )
{
    Box a(15,20),b(20,30);
    cout<<a.height<<endl;
    cout<<b.height<<endl;
    cout<<Box::height<<endl;
    cout<<a.volume( )<<endl;
    return 0;
}
```

9.9.2 静态成员函数

静态成员函数的声明：

static int volume( );

和静态数据成员一样，静态成员函数是类的一部分，而不是对象的一部分。

也允许用类名和域运算符“::”在类外调用公用静态成员函数。如

Box::volume( );

静态成员函数可以直接引用本类的静态数据成员，因为静态成员同样是属于类的

静态成员函数不属于某一对象，它没有this指针，无法对一个对象中的非静态成员进行默认访问(即在引用数据成员时不指定对象名)。

```
cout<<height<<endl;          //height已声明为static
cout<<a.width<<endl;         //width为非静态成员
```

例9.11 静态成员函数的应用

```
#include <iostream>
using namespace std;
class Student
{
public:
    Student(int n,int a,float s):num(n),age(a),score(s){ }
    void total( );
    static float average( );      //声明静态成员函数
private:
    int num;
    int age;
    float score;
    static float sum;             //静态数据成员
    static int count;             //静态数据成员
};
```

```
void Student::total()          //定义非静态成员函数
{
   sum+=score; count++;
}

float  Student::average()   //定义静态成员函数
{
   return(sum/count);
}

float Student::sum=0;          //对静态数据成员初始化
int Student::count=0;          //对静态数据成员初始化
```

```
int main( )
{
    Student stud[3]={
        Student(1001,18,70),
        Student(1002,19,78),
        Student(1005,20,98)};
    int n;
    cout<<"please input the number of students:";
    cin>>n;
    for(int i=0;i<n;i++)
        stud[i].total( );
    cout<<"the average score of "<<n<<" students
is“   <<Student::average( )<<endl;
    return 0;
}
```

- （1）可以实现某些特殊的设计模式：如Singleton；
- （2）由于没有this指针，可以把某些系统API的回调函数以静态函数的形式封装到类的内部。因为系统API的回调函数通常都是那种非成员函数（孤立函数），没有this指针的。比如你可以在类的内部写一个线程函数供CreateThread创建线程用，如果没有静态函数，那么这种回调函数就必须定义成全局函数（非静态成员函数指针无法转换成全局函数指针），从而影响了OO的“封装性”。
- （3）可以封装某些算法，比如数学函数，如ln，sin，tan等等，这些函数本就没必要属于任何一个对象，所以从类上调用感觉更好，比如定义一个数学函数类Math，调用Math::sin(3.14);如果非要用非静态函数，那就必须：

  Math　math;

  math.sin(3.14);

  行是行，只是不爽：就为了一个根本无状态存储可言的数学函数还要引入一次对象的构造和一次对象的析构，当然不爽。而且既然有了对象，说不得你还得小心翼翼的定义拷贝构造函数、拷贝赋值运算符等等，对于一些纯算法的东西显然是不合适的。
- （4）总之，从OOA/OOD的角度考虑，一切不需要实例化就可以有确定行为方式的函数都应该设计成静态的。

Friend

1. Note:

Typically，only the member functions of the class can access the protected member data. If we declared a common functions as a friend to the class, the common functions can also access the protected member data.

Friend is a mechanism to destroy the data encapsulation and concealment provided by C++. We can use friend functions and friend class.

In order to insure the integrities, encapsulation and concealment of member data, we advise that the friend should not or less be used in the class.

9.10 友元(friend)

9.10.1 友元函数

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。

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。

如果在本类以外的其他地方定义了一个函数，在类体中用**friend**对其进行声明，此函数就称为本类的友元函数。友元函数可以访问这个类中的私有成员。

2. Friend functions

- Friend function is a non-member function (common function). It can be declared in the class, What you should do is add keyword “friend” prior to the function.
- Similar to a common member function, friend function be realized within the class or outside the class.
- Friend functions’ accessing the member data must use object name.

Example of friend function -computing distance between two points

```
#include <iostream>
#include <math>
using namespace std;
class Point      //Point类声明
{
public:          //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;}
    int GetX(){return X;}
    int GetY(){return Y;}
    friend float fDist(Point &a, Point &b);
private:         //私有数据成员
    int X,Y;
};
```

```
double Distance( Point& a, Point& b) //友元函数，但不是point类的成员函数
{
    double dx=a.X-b.X;  //访问必须通过对象名
    double dy=a.Y-b.Y;
    return sqrt(dx*dx+dy*dy);
}
int main()
{
   Point p1(3.0, 5.0), p2(4.0, 6.0);
   double d=Distance(p1, p2); //调用友元函数
   cout<<"The distance is "<<d<<endl;
   return 0;
}
```

例9.12 普通友元函数

```
#include <iostream>
using namespace std;
class Time
{
public:
    Time(int,int,int);
    friend void display(Time &);
private:
    int hour;
    int minute;
    int sec;
};
```

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

void display(Time& t)
{ cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;}
//display不能默认引用Time类的数据成员
int main( )
{
   Time t1(10,13,56);
   display(t1);
   return 0;
}
```

例9.13 友元成员函数

```
#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 &);
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)
{ 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,int d,int y)          //类Date的构造函数
{month=m;day=d;year=y;}
int main( )
{
   Time t1(10,13,56);
   Date d1(12,25,2004);
   t1.display(d1);
   return 0;
}
```

调用友元函数访问有关类的私有数据的方法：

（1） 在函数名display的前面要加display所在的对象名(t1)；

（2） display成员函数的实参是Date类对象d1，否则就不能访问对象d1中的私有数据；

（3） 在Time::display函数中引用Date类私有数据时必须加上对象名，如d.month。

注意：一个函数可以被多个类声明为“朋友”，这样就可以引用多个类中的私有数据。

3. Friend class

- If class B is a friend of class A, then all member functions of class B can access the members in class A, vice versa.
- Declaration method: Declared class B with keyword “friend” in class A.

For example:

```
class A
{
    friend class B;
    public:…
}
```

9.10.2 友元类

将一个类(B类)声明为另一个类(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。

在A类的定义体中声明B类为其友元类：

friend B;

即声明友元类的一般形式为

friend 类名;

说明

(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。

除非确有必要，一般不把整个类声明为友元类。

关于友元利弊的分析： 友元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。

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。

*4.7 函数模板

函数模板(function template)即通用函数，其函数类型和形参类型不具体指定，用一个虚拟的类型来代表。

凡是函数的参数个数相同而类型不同，且函数体相同的函数都可以用这个模板来代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型。

例4.7 求2或3个整数中的最大值

```
#include <iostream>
using namespace std;
template<typename T>
//模板声明，其中T为类型参数

T max(T a,T b,T c)
//定义一个通用函数，用T作虚拟的类型名
{
    if(b>a) a=b;
    if(c>a) a=c;
    return a;
}
```

```
int main( )
{
    int i1=185,  i2=-76,  i3=567,  i;
    double d1=56.87,  d2=90.23,  d3=-3214.78,  d;
    long g1=67854,  g2=-912456,  g3=673456,  g;

    i=max(i1,i2,i3);
    d=max(d1,d2,d3);
    g=max(g1,g2,g3);

    cout<<"i_max="<<i<<endl;
    cout<<"f_max="<<f<<endl;
    cout<<"g_max="<<g<<endl;
    return 0;
}
```

9.11 类模板

```
class Compare_int          //整数作比较
{
public:
    Compare(int a,int b)
    {x=a;y=b;}
    int max( )
    {return(x>y)?x:y;}
    int min( )
    {return(x<y)?x:y;}
private:
    int x,y;
};
```

```
class Compare_float                    //浮点数作比较
{
public:
   Compare(float a,float b)
   {x=a;y=b;}
   float max( )
   {return(x>y)?x:y;}
   float min( )
   {return(x<y)?x:y;}
private:
   float x,y;
}
```

```
template<class numtype> //声明一个模板，虚拟类型名为numtype
class Compare                          //类模板名为Compare
{
public:
    Compare(numtype a,numtype b)
    { x=a;y=b;}
    numtype max( )
    {return (x>y)?x:y;}
    numtype min( )
    {return (x<y)?x:y;}
private:
    numtype x,y;
};
```

(1) 声明类模板时要增加一行

template <class 类型参数名>

(2) 原有的类型名换成虚拟类型参数名。

类模板包含类型参数，因此又称为参数化的类。

类是对象的抽象，对象是类的实例；

类模板是类的抽象，类是类模板的实例。

利用类模板可以建立含各种数据类型的类。

用类模板定义对象不能直接写成

Compare cmp(4,7); // Compare是类模板名

Compare是类模板名，而不是一个具体的类，类模板体中的类型numtype并不是一个实际的类型，只是一个虚拟的类型，无法用它去定义对象。必须用实际类型名去取代虚拟的类型：

Compare <int> cmp(4,7);

例9.14用类模板实现两个数的比较

```
#include <iostream>
using namespace std;
template<class numtype>                    //定义类模板
class Compare
{
public:
   Compare(numtype a,numtype b)
   {x=a;y=b;}
```

```
    numtype max( )
    {return (x>y)?x:y;}
    numtype min( )
    {return (x<y)?x:y;}
private:
    numtype x,y;
};

int main( )
{
    Compare<int> cmp1(3,7);          //定义对象cmp1，用于两个整数的比较
    cout<<cmp1.max( )<<" is the Maximum of two integer numbers."<<endl;
    cout<<cmp1.min( )<<" is the Minimum of two integer numbers."<<endl<<endl;
    Compare<float> cmp2(45.78,93.6); //定义对象cmp2，用于两个浮点数的比较
    cout<<cmp2.max( )<<" is the Maximum of two float numbers."<<endl;
    cout<<cmp2.min( )<<" is the Minimum of two float numbers."<<endl<<endl;
    Compare<char> cmp3('a','A');     //定义对象cmp3，用于两个字符的比较
    cout<<cmp3.max( )<<" is the Maximum of two characters."<<endl;
    cout<<cmp3.min( )<<" is the Minimum of two characters."<<endl;
    return 0;
}
```

说明： 上面列出的类模板中的成员函数是在类模板内定义的。如果改为在类模板外定义，不能用一般定义类成员函数的形式：

```
numtype Compare::max(){…}
//不能这样定义类模板中的成员函数
```

而应当写成类模板的形式：

```
template<class numtype>
numtype Compare<numtype>::max( )
{ return (x>y)?x:y;}
```

归纳以上的介绍，可以这样声明和使用类模板：

(1) 先写出一个实际的类。

(2) 将此类中准备改变的类型名(如int要改变为float或char)改用一个自己指定的虚拟类型名。

(3) 在类声明前面加入一行，格式为

template<class 虚拟类型参数>，如

```
template<class numtype>
//注意本行末尾无分号
class Compare
{…};                                    //类体
```

(4) 用类模板定义对象时用以下形式：

类模板名<实际类型名> 对象名;

类模板名<实际类型名> 对象名(实参表列);

Compare<int> cmp;

Compare<int> cmp(3,7);

(5) 如果在类模板外定义成员函数，应写成类模板形式：

template<class 虚拟类型参数>

函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表列) {…}

说明：

（1） 类模板的类型参数可以有一个或多个，每个类型前面都必须加class，如

```
template<class T1,class T2>
class someclass
{…};
```

在定义对象时分别代入实际的类型名，如

```
someclass<int,double> obj;
```

(2) 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。

(3) 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。

作业

- **P311：5、9**