C++程序设计(II)

浙江工业大学计算机学院

第8讲 类与对象的进一步讨论

1. 对象的生存周期
2. 对象的数组和指针
3. 对象的复制和赋值
4. 常量成员、静态成员和友元

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的生存周期
  - 对象生存期和变量一样，由声明决定。
  - 类中各个数据成员的生存期由对象决定，对象存在时它们就存在，对象撤销时它们就消失。
- 对象的初始化-构造函数
  - 对象初始化时的问题：在类声明中，不允许对数据成员用表达式初始化，导致声明一个对象时，其初始状态是不确定的，会带来错误的使用。
  - 如：
```
pool apool;
cout<<apool.rail_length();
```
或者
```
clock  aclock;
aclock.show_time();
```

第8讲 类与对象的进一步讨论
-对象的生存周期

❖对象的初始化-构造函数

- 问题的解决需求：对象是一个实体，它反映了客观事物的属性(例如时钟的时、分、秒的值)，在创建时就应该有确定的值；有时希望类中的某些操作仅在创建对象时执行一次，然后就不可再次执行这些操作。
- 问题的解决方案：c++语言为类提供构造函数，可自动完成对象的初始化任务。

第8讲 类与对象的进一步讨论
-对象的生存周期

❖ 对象的初始化-构造函数

- 是一种特殊的成员函数，不需要用户来调用它，而是在建立新对象时由程序自动调用执行。
- 构造函数的名字必须与类名字相同，以便编译系统能识别它并把它作为构造函数处理。
- 它不能指定任何返回类型，即使void也不允许，不返回任何值。
- 构造函数的功能是由用户定义的，用户可根据初始化的要求设计函数体和函数参数。
- 系统为没有构造函数的类提供缺省的构造函数。
- 任何声明的新对象都是通过构造函数初始化的；有正确对应的构造函数对象才能被正确的初始化。

第8讲 类与对象的进一步讨论
-对象的生存周期

❖对象的初始化-构造函数

```
class pool{
public:
   void build();
   double rail_length ();
   double rail_area();
private:
   double cir_area(double);
   double radius;
   double c;
};
```

```
class  clock{
public:
    void show_time();
    void set_time();
   double diff(const clock& T);
private:
    long normalize() const;
    int hour;
    int minute;
    int second;
 };
```

- 均没有构造函数，使用缺省的构造函数初始化对象。

第8讲 类与对象的进一步讨论
-对象的生存周期

❖ 对象的初始化-构造函数

- 缺省的构造函数什么样？没有任何形式的参数，并且函数体为空。

```
class pool{
public:
   pool(){ }
   void build();
   double rail_length ();
   double rail_area();
private:
   double cir_area(double);
   double radius;
   double c;
};
```

```
class  clock{
public:
    clock(){ }
    void show_time();
    void set_time();
   double diff(const clock& T);
private:
    long normalize() const;
    int hour;
    int minute;
    int second;
 };
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 自定义的无参构造函数(内联写法)，如

```
class  clock{
public:
    clock(){ hour=0; minute=0; second=0; }
    void show_time();
    void set_time();
   double diff(const clock& T);
private:
    long normalize();
    int hour;
    int minute;
    int second;
 };
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 自定义的无参构造函数(分离写法)，如

```
class  clock{
public:
    clock();
    ……
private:
   ……
};
```

```
clock::clock()
{  hour=0;
    minute=0;
    second=0;
}
```

```
clock::clock():hour(0),minute(0),second(0)
{    }
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 自定义的带参构造函数(内联写法)，如

```
class pool{
public:
   pool(double rv,double cv){radius=rv; c=cv;  }
   void build();
   double rail_length ();
   double rail_area();
private:
   double cir_area(double);
   double radius;
   double c;
};
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 自定义的带参构造函数(分离写法)，如

```
class pool{
public:
   pool(double,double);
   …..
private:
   ……
};
```

```
pool::pool(double rv,double cv)
{   radius=rv;
     c=cv;
}
```

```
pool::pool(double rv,double cv):radius(rv),c(cv)
{    }
```

第8讲 类与对象的进一步讨论
-对象的生存周期

❖对象的初始化-构造函数

- 当用户自定义构造函数后，系统不再提供缺省构造函数。

```
class pool{
public:
   pool(double,double);
   …..
private:
   ……
};
```

```
pool::pool(double rv,double cv)
{   radius=rv;
     c=cv;
}
```

- 需要显式添加无参构造函数

```
……
int main()
{  pool   apool(6.5, 2.3);  //V
    pool  bpool;  //X
}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 一个类可以拥有多个构造函数。

构造函数重载

```
class pool{
public:
   pool(double,double);
   pool();
   …..
private:
   ……
};
```

```
pool::pool(double rv,double cv)
{   radius=rv;
     c=cv;
}

pool::pool():radius(0),c(0)
{   }
```

```
……
int main()
{  pool   apool(6.5, 2.3);  //V
    pool  bpool;  //V
}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 一个类可以拥有多个构造函数。

使用默认参数的构造函数

```
class pool{
public:
   pool(double rv=0,double cv=0);
  …..
private:
   ……
};
```

```
pool::pool(double rv,double cv)
{   radius=rv;
     c=cv;
}
```

- 按声明对象的形式来选择一个匹配的构造函数。

```
……
int main()
{   pool   apool(6.5, 2.3);  //V
     pool  bpool;  //V
     pool  cpool(6.5); //V
}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 例: 有两个长方柱，其长、宽、高分别为：(1)12,20,25; (2)10,14,20。求它们的体积。编一个基于长方柱类的程序，在类中用带参数的构造函数。

```
class Box{
public:
   Box(int,int,int);
   int volumn();//计算体积
private:
   int length;//长
   int width;//宽
   int height;//高
};
```

```
Box::Box(int h,int w,int len)
{
   height=h;
   width=w;
   length=len;
}

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

```
//使用类Box
#include <iostream>
#include “box.h”
using namespace std;
int  main( )
{Box box1(12,25,30); //建立对象box1，并指定box1长、宽、高的值
 Box box2(15,30,21); //建立对象box2，并指定box2长、宽、高的值
 cout<<"The volume of box1 is"<<box1.volume()<<endl;
 cout<<"The volume of box2 is"<<box2.volume()<<endl;
 return 0;
}
```

程序运行结果如下：

```
The volume of box1 is 9000
The volume of box2 is 9450
```

- 带参数的构造函数中的形参，其对应的实参在定义对象时给定。
- 用这种方法可以方便地实现对不同的对象进行不同的初始化。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 修改例，使用构造函数重载。

```
class Box1{
public:
   Box1();
   Box1(int,int,int);
   int volumn();//计算体积
private:
   int length;//长
   int width;//宽
   int height;//高
};
```

```
Box1::Box1()
{ length=10;
   width=10;
   heigth=10;
}

Box1::Box1(int h,int w,int len)
{
    length=h;
    width=w;
    height=len;
}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数

```
//使用类Box1
#include <iostream>
#include “box.h”
using namespace std;
int  main( )
{Box1 box1; //建立对象box1，box1长、宽、高的值均为10
 Box1 box2(15,30,21); //建立对象box2，并指定box2长、宽、高的值
 cout<<"The volume of box1 is"<<box1.volume()<<endl;
 cout<<"The volume of box2 is"<<box2.volume()<<endl;
 return 0;
}
```

(1)调用构造函数时不必给出实参的构造函数，称为**默认构造函数(default constructor)**。无参构造函数属于默认构造函数。一个类只有一个默认构造函数。

(2)如果在建立对象时选用的是无参构造函数，注意正确书写定义对象的语句。

Box box1;

不要写成**Box box1（）;**

(3)尽管在一个类中可以包含多个构造函数，但对于每个对象来说，建立对象时只是执行一个构造函数，并非每个构造函数都被执行。

第8讲 类与对象的进一步讨论
-对象的生存周期

❖对象的初始化-构造函数

- 修改例，使用默认参数的构造函数。

```
class Box
{public:
   //在明构造函数时指定默认参数
   Box(int h=10,int w=10,int len=10);
   int volume( );
  private:
   int height;
   int width;
   int length;
};
//在定义函数时可以不指定默认参数
Box::Box(int h,int w,int len)
{height=h;
  width=w;
  length=len;
 }
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数

```
//使用类Box2
#include <iostream>
#include “box.h”
using namespace std;
int  main( )
{  Box box1; //没有给实参
   Box box2(15); //只给定一个实参
   Box box3(15,30); //只给定2个实参
   Box box4(15,30,20); //给定3个实参
   cout<<"The volume of box1 is"<<box1.volume()<<endl;
   cout<<"The volume of box2 is"<<box2.volume()<<endl;
   cout<<"The volume of box3 is"<<box3.volume()<<endl;
   cout<<"The volume of box4 is"<<box4.volume()<<endl;
   return 0;
}
```

- 构造函数中使用默认参数，提供建立对象的多种选择，作用相当于好几个重载的构造函数。

- 构造函数中使用默认参数的好处是： 即使调用构造函数时没有提供实参值，不仅不会出错，还确保按照默认参数值进行对象初始化。尤其在希望对每一个对象都有同样的初始化状况时用这种方法更为方便。

- 如果构造函数的全部参数都指定了默认值，则在定义对象时可以给一个或几个实参，也可不给出实参。

- 一个类中定义了全是默认参数的构造函数后，不能再定义重载构造函数。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化-构造函数
  - 回顾及思考：
    1）什么时候调用构造函数？
    2）构造函数能干些什么？
    3）构造函数定义的形式什么样？
    4）什么时候缺省的构造函数够用？
    5） 什么时候需要自己定义构造函数？

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的撤销-析构函数
  - 也是一个特殊的成员函数，与构造函数的功能相反。用于在对象撤销时执行一些清理任务。当对象的生命期结束时，会自动执行析构函数。
  - 析构函数的名字必须是类名前加一个波浪纹号“~”，以区别于构造函数。
  - 与构造函数相同，它不能指定任何返回类型，即使void也不允许，不返回任何值。
  - 与构造函数不同，定义析构函数时不能指定任何形式的参数。因为创建对象的时机是由程序员指定的，而撤销对象是在对象生存期结束时由系统自动完成，程序员无法提供析构函数的实际参数。----不能重载
  - 与构造函数类似，当类没有定义析构函数时，系统提供缺省的析构函数，对象的撤销一定是通过析构函数进行的。

- 对象的生命期结束时，会自动执行析构函数。

①如果一个函数定义了一个对象(自动局部对象)，当这个函数被调用结束时，自动执行析构函数。

②static局部对象在函数调用结束时对象并不释放，因此也不调用析构函数。只在main函数结束或调用exit函数结束程序时，才调用static局部对象的析构函数。

③如果定义了一个全局对象，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)，调用该对象的析构函数。

④如果用new运算符动态地建立了一个对象，用delete运算符释放该对象时，调用该对象的析构函数。

- 析构函数的作用并不是删除对象，而是在撤销对象占用的内存之前完成一些清理工作。
- 析构函数不返回任何值，没有函数类型没有函数参数。一个类只能有一个析构函数。
- 析构函数的作用并不仅限于释放资源方面，它还可以被用来执行“用户希望在最后一次使用对象后所执行的任何操作”。因为析构函数是在声明类的时候定义的，可完成类的设计者所指定的任何操作。
- 类的设计者应当在声明类的同时定义析构函数，以指定如何完成“清理”的工作。
- 如果用户没有定义析构函数，C++编译系统会自动生成一个析构函数，但它实际上什么操作都不进行。

第8讲 类与对象的进一步讨论
-对象的生存周期

❖对象的撤销-析构函数

- 缺省的析构函数与缺省的构造函数类似，无参数，函数体为空。

```
class pool{
public:
   pool(){  } //缺省的构造函数
……
  ~pool(){ }//缺省的析构函数
private:
   double cir_area(double);
   double radius;
   double c;
};
```

```
class  clock{
public:
    clock(){ }//缺省的构造函数
……
    ~clock(){  }//缺省的析构函数
private:
    long normalize() const;
    int hour;
    int minute;
    int second;
 };
```

第8讲 类与对象的进一步讨论
-对象的生存周期

❖ 对象的撤销-析构函数

- 一般的情况，不需要为类特别书写析构函数，默认的析构函数就足够。
- 但是，当类的数据成员为指针且关联动态派生的空间时，必须手工添加上析构函数。

```
class ex1{
public:
    ex1(int v=0){x=v;}
    void setx(int v){x=v;}
    int getx(){return x;}
private:
   int x;
};
```

```
class  ex2{
public:
     ex2(int v=0){x=new int(v);}
     void setx(int v){*x=v;}
     int  getx(){return *x;}
     ~ex2(){delete x;}
private:
    int *x;
};
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的撤销-析构函数
  - 回顾及思考：

    1）什么时候调用析构函数？

    2）析构函数能干些什么？

    3）析构函数定义的形式什么样？

    4）什么时候缺省的析构函数够用？

    5）什么时候需要自己定义析构函数？

例2.5 包含构造函数和析构函数的C++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;            //输出有关信息
 }
 ~Student( )                                    //定义析构函数
 {cout<<"Destructor called."<<endl;}            //输出有关信息
 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
 //定义对象 stud2
 Student stud2(10011,"Zhang_fun",'m');
stud2.display( );                          //输出学生2的数据
 return 0;
}
```

程序运行结果如下：

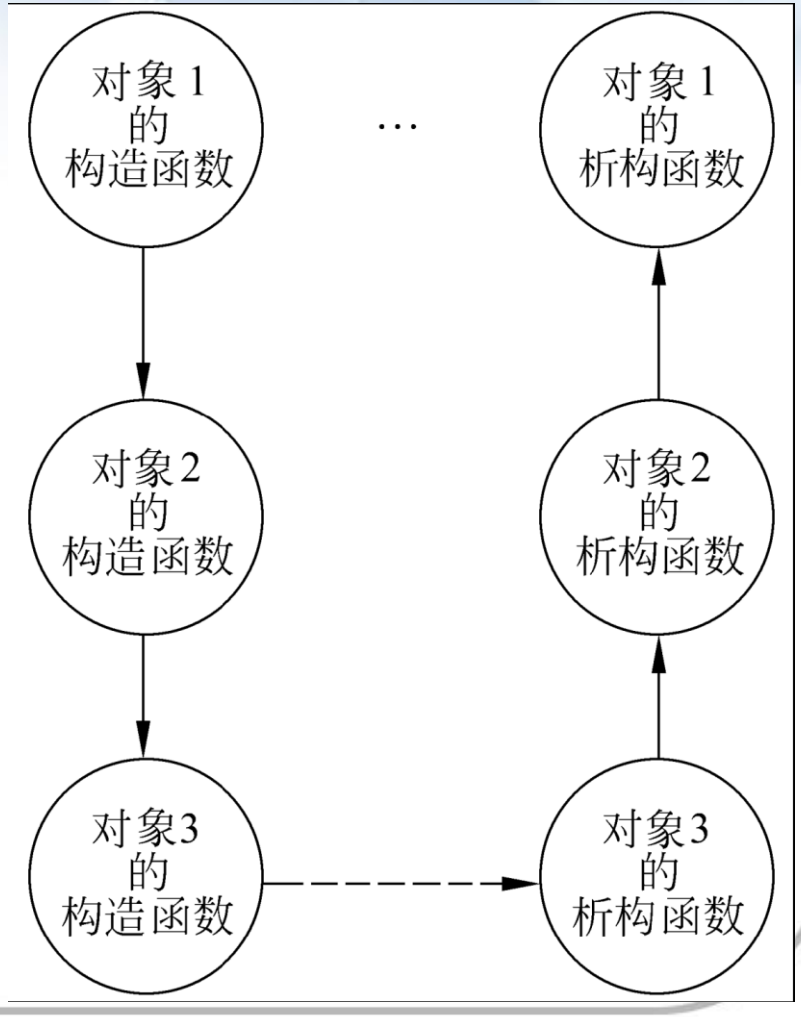
```
Constructor called.          (执行stud1的构造函数)
num: 10010                   (执行stud1的display函数)
name:Wang_li
sex: f

Constructor called.          (执行stud2的构造函数)
num: 10011                   (执行stud2的display函数)
name:Zhang_fun
sex:m
Destructor called.           (执行stud2的析构函数)
Destructor called.           (执行stud1的析构函数)
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化和撤销
  - 在一般情况下(相同作用域)，调用析构函数的次序正好与调用构造函数的次序相反：

最先被调用的构造函数，其对应的(同一对象中的)析构函数最后被调用，而最后被调用的构造函数，其对应的析构函数最先被调用。



第8讲 类与对象的进一步讨论
-对象的生存周期

- 对象的初始化和撤销
  - 测试构造函数和析构函数在何处调用的例程

```
class  ex2{
public:
    ex2(int v=0){
    x=new int(v);
    cout<<“构造对象！ ” <<x<<endl;}
    void setx(int v){*x=v;}
    int  getx(){return *x;}
    ~ex2(){
   cout<<“析构对象！ ” <<x<<endl;
    delete x;}
private:
   int *x;
};
```

```
#include “ex2.hpp”
main()
{ ex2 objA(3),objB,objC(4);
  cout<<“objA:”<<objA.getx()<<endl;
   cout<<“objB:”<<objB.getx()<<endl;
  cout<<“objC:”<<objC.getx()<<endl;
  objA.setx(5);
  objB.setx(6);
  objC.setx(6);
   cout<<“objA:”<<objA.getx()<<endl;
   cout<<“objB:”<<objB.getx()<<endl;
  cout<<“objC:”<<objB.getx()<<endl;
}
```

- 并不是在任何情况下都是按这一原则处理的。对象可以在不同的作用域中定义，可以有不同的存储类别。这些会影响调用构造函数和析构函数的时机。

(1)在全局范围中定义的对象(所有函数外定义的对象)，它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。

但如果一个程序中有多个文件，不同文件中都定义了全局对象，这些对象的构造函数执行顺序不确定。当main函数执行完或调exit函数时，调用析构函数。

(2) 如果定义的是局部自动对象(函数中定义对象)，则在建立对象时调用其构造函数。

如果函数被多次调用，则在每次建立对象时都要调用构造函数。在函数调用结束、对象释放时先调用析构函数。

(3) 如果在函数中定义静态(static)局部对象，只在程序第一次调用此函数建立对象时调用构造函数一次，在调用结束时对象并不释放，也不调用析构函数。

main函数结束或调用exit函数结束程序时，才调用析构函数。

对象数组

❖数组不仅可以由简单变量组成，也可以由对象组成。

❖一个班有**50**个学生，如果为每一个学生建立一个对象，需要分别取**50**个对象名。

❖这时可以定义一个“学生类”对象数组，每一个数组元素是一个“学生类”对象。

```
Student stud[50];
//假设已声明了Student类，定义stud数组，有50个元素
```

- 建立数组时要调用构造函数，50个元素需要调用50次构造函数。可以在定义数组时提供实参以实现初始化。

- 如果构造函数只有一个参数，定义数组时可以直接提供实参。

```
Student stud[3]={60,70,78};
//3个实参分别传递给3个数组元素的构造函数
```

- 如果构造函数有多个参数，则不能用在定义数组时直接提供所有实参的方法，因为一个数组有多个元素，对每个元素要提供多个实参，如果再考虑到构造函数有默认参数的情况，很容易造成实参与形参的对应关系不清晰。

- 如果构造函数有多个参数，在定义对象数组时应当怎样实现初始化呢？

- 花括号中分别写出构造函数并指定实参。如果构造函数3个参数分别代表学号、年龄、成绩，可以这样定义对象数组：

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

- 建立对象数组时，分别调用构造函数，对每个元素进行初始化。每个元素实参分别用括号包起来，对应构造函数的一组形参。

例2.6 对象数组的使用方法。

```
#include <iostream>
using namespace std;
class Box
{public:
//声明有默认参数的构造函数，用参数初始化表对数据成员初始化
    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]={                    //定义对象数组
    Box(10,12,15),             //调用构造函数Box，提供第1个元素的实参
    Box(15,18,20),             //调用构造函数Box，提供第2个元素的实参
    Box(16,20,26)              //调用构造函数Box，提供第3个元素的实参
   };
//调用a[0]的volume函数
cout<<"volume of a[0] is "<<a[0].volume( )<<endl;
//调用a[1] 的volume函数
cout<<"volume of a[1] is "<<a[1].volume( )<<endl;
//调用a[2] 的volume函数
cout<<"volume of a[2] is "<<a[2].volume( )<<endl;
}
```

运行结果如下：

volume of a[0] is 1800
volume of a[1] is 5400
volume of a[2] is 8320

对象指针

❖在建立对象时，编译系统会为每个对象分配存储空间。对象空间的起始地址就是对象的指针。可以定义一个指针变量，用来存放对象的指针。

```
class Time
{public:
   int hour;
   int minute;
   int sec;
   void get_time( );
 };

void Time::get_time( )
{cout<<hour<<":"<<minute<<":"<<sec<<endl;}
```

2.5.1 指向对象的指针

```
Time *pt;
Time t1;
pt=&t1;
```

- 定义指向类对象的指针变量的一般形式为
  类名 *对象指针名;

- 可以通过对象指针访问对象和对象的成员

```
*pt
(*pt).hour
pt->hour
(*pt).get_time ( )
 pt->get_time ( )
```

2.5.2 指向对象成员的指针

1. 指向对象数据成员的指针

数据类型名 *指针变量名；

❖ 如果Time类的数据成员hour为公用的整型数据，则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。

```
int *p1;
p1=&t1.hour; //定义指向数据成员指针变量,前提数据成员为public
cout<<*p1<<endl; //输出p1所指的数据成员t1.hour
```

2. 指向对象成员函数的指针

```
clock*  p2;
p2=t1.get_time;  //出现编译错误
```

- 指针变量必须与成员函数在以下三方面匹配：
  ①参数类型和参数个数；②函数返回值的类型；③所属类。

- 定义指向公用成员函数的指针变量的一般形式为
  数据类型名 (类名::*指针变量名)(参数列表);
  如 **void (Time::*p2)( );**
  //定义p2为指向Time类中公用成员函数的指针变量

- 使指针变量指向一个公用成员函数的一般形式为
  指针变量名=&类名::成员函数名;
  如 **p2=&Time::get_time;**
  //把公用成员函数的入口地址赋给一个指向公用成员函数的指针变量

(t1.*p3)( ); //使用指向成员函数的指针

例2.7 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( );
};

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

```
void Time::get_time( )
{cout<<hour<<":"<<minute<<":" <<sec<<endl;}

int main( )
{Time t1(10,13,56);

  int *p1=&t1.hour;        //定义指向整型数据的指针变量p1，指向t1.hour
  cout<<*p1<<endl;         //输出p1所指的数据成员t1.hour

  t1.get_time( );

  Time *p2=&t1;        //定义指向Time类对象的指针变量p2，指向t1
  p2->get_time( );

  void (Time::*p3)( );   //定义指向公用成员函数的指针变量p3
  p3=&Time::get_time;
  (t1.*p3)( );           //调用对象t1中p3所指的成员函数(即t1.get_time( ))
}
```

程序运行结果为

```
10                  (main函数第4行的输出)
10:13:56            (main函数第5行的输出)
10:13:56            (main函数第7行的输出)
10:13:56            (main函数第10行的输出)
```

❖成员函数入口地址的正确写法是：

&类名::成员函数名

❖main函数第8、9行可以合写为一行：

```
void (Time::*p3)( )=&Time::get_time;
```

2.5.3 this指针

- 每个对象中的数据成员都占有存储空间，如果对同一个类定义了n个对象，则n组同样大小的空间存放n个对象中的数据成员。但是不同对象都调用同一个函数代码段。
- Box类定义了两个同类对象a, b
  - a.volume( )应该是引用对象a中的height，width和length计算出长方体a的体积。
  - b.volume( )应该是引用对象b中的height，width和length，计算出长方体b的体积。
  - 现在都用同一个函数段，系统怎样使它分别引用a或b中的数据成员呢？

❖在每一个成员函数中都包含一个特殊的指针，这个指针的名字是固定的，称为**this**。它是指向本类对象的指针，
它的值是当前被调用成员函数所在对象的起始地址。

❖当调用成员函数**a.volume**时，编译系统就把对象**a**的起始地址赋给**this**指针。实际上是执行：

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

- 相当于执行：

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

❖同样如果有**b.volume( )**，编译系统就把对象**b**的起始地址赋给成员函数**volume**的**this**指针，显然计算出来的是长方体**b**的体积。

❖this指针是隐式使用的，作为参数被传递给成员函数。

```
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);
}
```

❖在调用该成员函数时，实际上是用以下方式调用的：

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

- 将对象a的地址传给形参this指针。然后按this的指向去引用其他成员。

❖在需要时也可以显式地使用**this**指针。

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

等价于

```
return(this->height * this->width * this->length);
```

❖可以用***this**表示被调用的成员函数所在的对象，***this**就是当前对象。

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

❖***this**两侧的括号不能省略

课堂练习

1.下列关于构造函数的描述中，【】是正确的。

A) 构造函数名必须与类名相同

B) 构造函数不可以重载

C) 构造函数不能带参数

D) 构造函数可以声明返回类型

❖2.下列的【】不是构造函数的特征。

A) 构造函数的函数名与类名相同

B) 构造函数可以重载

C) 构造函数可以带有参数

D) 可以指定构造函数的返回值类型

❖3.关于构造函数特点的描述中，错误的是　【】

A. 定义构造函数必须指出类型 。

B. 构造函数的名字与该类的类名相同

C. 一个类中可定义0至多个构造函数 。

D. 构造函数是一种成员函数 。

4. 对重载函数形参的描述中，错误的是

A. 参数的个数可能不同

B. 参数的类型可能不同

C. 参数的顺序可能不同

D. 参数的个数、类型、顺序都相同，只是函数的返回值类型不同

编程练习2

- 1．创建一个Student 类，要求：

  该类封装学生的姓名、性别和成绩等信息。

  通过构造函数给姓名和性别信息赋值。

  姓名和性别信息只能读不能更改，成绩信息通过成员函数进行读写，对成绩进行赋值时，若成绩大于100 分赋100 分，若成绩低于0 分赋0 分。

  主函数中利用数组stud1创建3个学生的信息，利用指针ps指向数组元素，实现学生信息的显示和修改功能。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 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，这时可以使用const，即把有关的数据定义为常量。
  - 常对象: 定义对象时指定对象为常量。

1)必须在定义时初始化;

2)在以后的使用中所有数据成员的值不能被修改;

3)不能调用该对象的非const型的成员函数(除了由系统自动调用的隐式的构造函数和析构函数)。

类名 const 对象名[(实参表列)];

或者

const 类名 对象名[(实参表列)];

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 常对象例：

```
const Box t1(10,15,36); //定义常对象t1
t1.volumn( ); //X
//企图调用常对象t1中的非const型成员函数，非法
```

必须在Box内将volumn修改为

```
double clock::diff(const clock& T)
{   long d=T.normalize()-normalize()-60;
    if(d<0)    return 0;
    int h=d\60,m=d%60;
    if(m<15) return h;
    if(m>=15&&m<30) return h+0.5;
    if(m>=30&&m<60) return h+1;
}
```

```
class  clock{
public:
    void show_time();
    void set_time();
   double diff(const clock& T);
private:
    long normalize() const;
    int hour;
    int minute;
    int second;
     };
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 常对象成员（常数据成员，常成员函数）
  - 常数据成员:用关键字const来声明。值是不能改变的。注意：1)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。2)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。

```
class CA{
public:
    CA(int
d):data(d){   }
private:
    const int
data;
```

```
class CB{
public:
    CB(CA
d):data(d){   }
private:
    const CA
data;
```

第8讲 类与对象的进一步讨论
-对象的生存周期

❖共用数据的保护(const在类机制中的使用)

- 常成员函数: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。注意：1)常成员函数可以引用const数据成员，也可以引用非const的数据成员。const数据成员可以被const成员函数引用，也可以被非const的成员函数引用。2) 常成员函数，则只能引用本类中的数据成员，而不能修改它们，例如只用于输出数据等。

不允许修改的
数据前加const

- 如何处理在一个类中，某些数据允许被修改，而另一些数据不允许被修改？
- 如何处理在一个类中，所有数据成员都不允许被修改？

1)所有数据前加const; 2)将对象声明为常对象（对象前加const)

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 常数据成员和常成员函数
  - 1）常对象只能调用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)。这是为了保证数据的安全；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。
  - 2）常对象中的成员函数不一定都是常成员函数。常对象只保证其数据成员是常数据成员，其值不被修改。如果在常对象中的成员函数未加const声明，编译系统把它作为非const成员函数处理。
  - 3）常成员函数不能调用另一个非const成员函数。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 指向对象的常指针/指向常对象的指针变量
  - 指向对象的常指针：将指针变量声明为const型，这样指针值始终保持为其初值，不能改变。

    clock t1(10,12,15), t2; //定义对象

    clock* const ptr1; //规定ptr1的值是常值

    ptr1=&t1; //ptr1指向对象t1，此后不能再改变指向

    ptr1=&t2; //X，ptr1不能改变指向
  - 定义指向对象的常指针的一般形式为

    类名 * const 指针变量名；

•用途：通常把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。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 指向对象的常指针/指向常对象的指针变量
  - 指向常对象的指针变量：const修饰类型，表示指针值关联的变量是一个不允许改变值的变量。

```
const clock t1(10,12,15); //定义常对象
clock t2; //定义对象
const clock* ptr1; //规定ptr1的值是常值
ptr1=&t1; //ptr1指向常对象t1，t1只能由这样的指针指向
ptr1=&t2; //√ ptr1指向t2，t2的值不能通过ptr1被修
```

•用途：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。

const 类名 * 指针变量名；

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 指向对象的常指针/指向常对象的指针变量
  - 当调用函数时实参对象的值不能被修改，该如何处理?
    - 1)指向常对象的指针 const clock* ptr
    - 2)对象的常引用 const clock& d
  - 要求对象在函数调用时不能被修改，且程序执行时都不能被修改，该如何处理？
    - 定义对象为常对象 const clock t1(12,0,5);

第8讲 类与对象的进一步讨论
-对象的生存周期

- 共用数据的保护(const在类机制中的使用)
  - 对象的常引用：拷贝构造函数的参数类型设计

    const 类名&

  - **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，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。用常指针和常引用作函数参数，可以提高程序运行效率。**

例2.8 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;}
```

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

❖如果不希望在函数中修改实参t1的值，可以把引用变量t声明为

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

❖用常指针和常引用作函数参数。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。

第8讲 类与对象的进一步讨论

❖共用数据的保护(const在类机制中的使用)

- const总结

形式	含义
const clock A(12,0,6);	A为常量，任何时候不许修改
const int data;	data为常量，值不允许改变
void clock::f()const	F为常成员函数，不能通过f来修改clock类的数据成员
clock const *ptr=&A;	ptr是一个常指针，ptr的值不能修改。
const clock * ptr;	ptr指向的对象是一个不允许通过ptr修改值的对象。
const clock& B=A;	B是一个clock对象的常引用，不能通过B修改A的值;B，A空间共享。

2.7 对象的动态建立和释放

❖有时人们希望在需要用到对象时才建立对象，不需要用该对象时就撤销它，释放它所占的内存空间以供别的数据使用。这样可提高内存空间的利用率。

❖**new**运算符动态建立对象，**delete**运算符撤销对象。

❖如果已定义了一个**Box**类，可以动态地建立一个对象：

new Box;

- 编译系统开辟一段内存空间存放一个Box类对象，同时调用该类的构造函数，以使该对象初始化。但是此时用户还无法访问这个对象，因为这个对象既没有对象名，用户也不知道其地址。这种对象称为无名对象。

❖new运算符动态地分配内存后，将返回一个指向新对象的指针值。需要定义一个指向本类的对象的指针变量存放该地址。如

```
Box *pt;
pt=new Box;
```

❖在程序中就可以通过pt访问这个新建的对象。如

```
cout<<pt->height;
cout<<pt->volume( );
```

❖可以在执行**new**时，对新建立的对象进行初始化。如

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

❖**new**建立的动态对象一般不用对象名，通过指针访问，它主要应用于动态的数据结构，如链表。

- 上一个结点中存放下一个结点的地址，由上一个结点找到下一个结点，构成链接的关系。

❖在执行new运算时，如果内存量不足，无法开辟所需的内存空间，目前大多数C++编译系统都使new返回一个0指针值。

❖ANSI C++标准提出，在执行new出现故障时，就“抛出”一个“异常”，用户可根据异常进行有关处理。

❖在不再需要使用由new建立的对象时，可以用delete运算符予以释放。如

```
delete pt;                    //释放pt指向的内存空间
```

❖撤销了pt指向的对象。如果用一个指针变量pt先后指向不同的动态对象，应注意指针变量的当前指向，以免删错了对象。

❖执行delete运算符时，在释放内存空间之前，自动调用析构函数，完成有关善后清理工作。

第8讲 类与对象的进一步讨论

对象的动态建立和释放

- 与普通的变量动态空间使用方法相同：new和delete

new

- Box *pt1,*pt2; //定义一个指向Box类对象的指针量pt
- pt1=new Box; //pt1,pt2中存放新建对象的起始地址
- cout<<pt1->volumn();
- pt2=new Box(15,16,17);
- cout<<pt2->volumn();

delete

- delete pt1;
- delete pt2;

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 用一个对象去初始化另一个对象----类内有特殊的成员函数来完成这个工作。→拷贝构造函数
- 说明：1. 拷贝构造函数也是系统会缺省给的。

  (和构造函数，析构函数类似)

  2. 有时候缺省的拷贝构造函数不够用。

  →需要自己显式的定义

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 拷贝构造函数什么样？

```
class ex1{
public:
    ex1(int v=0){x=v;}
    //缺省的拷贝构造函数
    ex1(const ex1& r){x=r.x;}
    void setx(int v){x=v;}
    int getx(){return x;}
private:
   int x;
};
```

```
class  ex2{
public:
     ex2(int v=0){x=new int(v);}
     //缺省的拷贝构造函数
     ex2(const ex2& r){x=r.x;}
     void setx(int v){*x=v;}
     int  getx(){return *x;}
     ~ex2(){delete x;}
private:
    int *x;
};
```

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 拷贝构造函数什么样？

1. 是构造函数的一种重载形式;
2. 类内语法：

```
类名(const 类名& obj)
{
   //拷贝函数体
}
```

3. 也可以在类内写声明，在类外实现
4. 缺省的拷贝函数，只完成对象数据成员的简单对拷。

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 为ex2写正确的拷贝构造函数，再测试

```
class  ex2{
public:
    ex2(int v=0){x=new int(v);}
   //拷贝构造函数的类内声明
    ex2(const ex2& r);
    void setx(int v){*x=v;}
    int  getx(){return *x;}
    ~ex2(){delete x;}
private:
   int *x;
};
```

```
ex2::ex2(const ex2&  r)
{
        ???
}
```

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 拷贝函数在用已有对象复制一个新对象时被调用
- 在以下3种情况下需要克隆对象：

1.声明语句中建立新对象时用已存在的对象进行初始化；

2.将一个对象作为函数的参数按值调用方式传递给另一个对象时生成对象副本；

3.生成一个临时对象作为函数的返回结果。

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 拷贝构造函数使用例：

```
void fun(Box b)   //形参是类的对象
{       }

int  main( )
{Box box1(12,15,18);
 fun(box1); //实参是类的对象
//调用拷贝构造函数将复制box1的一个新对象b
 return 0;
}
```

第8讲 类与对象的进一步讨论

❖对象的复制-拷贝构造函数

- 拷贝构造函数使用例：

```
Box  Box f( ) //函数f的返回值类型为Box类类型
{
   Box box1(12,15,18);
   return box1; //返回值是Box类的对象
}
int main( )
{Box box2; //定义Box类的对象box2
 f( ); //调用f函数，返回Box类的临时对象
}
```

❖对象的赋值

❖如果对一个类定义了两个或多个对象，这些同类的对象间可互相赋值。对象的值是指对象中所有数据成员的值。

❖对象之间的赋值也是通过赋值运算符“=”进行的，这是通过对赋值运算符的重载实现的。实际这个过程是通过成员复制来完成的，即将一个对象的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;

- 对象名1和对象名2必须属于同一个类

第8讲 类与对象的进一步讨论

❖对象的赋值-成员函数：赋值运算

- 使用赋值运算的举例：

```
#include “ex1.h”
//#include “ex2.h”
int main()
{
ex2 objA(3),objB(4),objC;
cout<<"objA:"<<objA.getx()<<endl;
cout<<"objB:"<<objB.getx()<<endl;
cout<<"objC:"<<objC.getx()<<endl;
objC=objA;
 cout<<"objA:"<<objA.getx()<<endl;
 cout<<"objB:"<<objB.getx()<<endl;
 cout<<"objC:"<<objC.getx()<<endl;
```

```
cout<<"set objA,objB,
                    ojbC :5,6,7."<<endl;
objA.setx(5);
objB.setx(6);
objC.setx(7);
cout<<"objA:"<<objA.getx()<<endl;
cout<<"objB:"<<objB.getx()<<endl;
cout<<"objC:"<<objC.getx()<<endl;
return 0;
}
```

第8讲 类与对象的进一步讨论

- 对象的赋值-成员函数：赋值运算
  - ex1工作正常；ex2工作不正常；
  - 类可以执行的赋值运算C++也会有缺省的；
  - 但是有时候不够用，需要自己定义
  - 缺省的赋值运算：

```
类名&    operator=(const 类名& right)
{
    (*this).成员=right.成员;
    return *this;
}
```

void行吗？

第8讲 类与对象的进一步讨论

❖对象的赋值-成员函数：赋值运算

- ex1,ex2中的缺省赋值运算成员函数

```
class ex1{
public:
    ex1(int v=0){x=v;}
    ex1(const ex& r);
    //缺省的赋值函数
    ex1& operator=(const ex1&
r){x=r.x; return (*this); }
    void setx(int v){x=v;}
    int getx(){return x;}
private:
   int x;
};
```

```
class  ex2{
public:
     ex2(int v=0){x=new int(v);}
     ex2(const ex2& r);
    //缺省的赋值函数
     ex2& operator=(const ex2&
r){x=r.x; return (*this); }
     void setx(int v){*x=v;}
     int  getx(){return *x;}
     ~ex2(){delete x;}
private:
    int *x;
};
```

第8讲 类与对象的进一步讨论

❖对象的赋值-成员函数：赋值运算

- 自己定义ex2中的赋值运算成员函数，再测试

```
class  ex2{
public:
     ex2(int v=0){x=new int(v);}
    //赋值函数的类内声明
    ex2& operator=(const ex2&
r);
     void setx(int v){*x=v;}
     int  getx(){return *x;}
     ~ex2(){delete x;}
private:
    int *x;
};
```

```
ex2& ex2::operator =(const
ex2&  r)
{
     *x=*r. x;
     return (*this);
}
```

第8讲 类与对象的进一步讨论

❖静态成员

- 静态数据成员:静态数据成员是一种特殊的数据成员。它以关键字static开头。

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

第8讲 类与对象的进一步讨论
-对象的生存周期

❖静态成员

- 静态数据成员:静态数据成员是一种特殊的数据成员。它以关键字static开头。

1) 属于类，不属于某个对象。(类的所有对象共享的数据)

2) 只能在类外初始化。不能用参数初始化表对静态数据成员初始化, 未初始化，默认为0。

数据类型类名::静态数据成员名=初值;

3) 一般数据成员是在对象建立时分配空间，在对象撤销时释放。静态数据成员是在程序编译时被分配空间的，到程序结束时才释放空间。----全局寿命

4)可以通过对象名引用，也可以通过类名引用。

第8讲 类与对象的进一步讨论
-对象的生存周期

❖静态成员

- 静态数据成员:静态数据成员是一种特殊的数据成员。它以关键字static开头。

```
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; //通过对象名a引用静态数据成员
  cout<<b.height<<endl; //通过对象名b引用静态数据成员
  cout<<Box::height<<endl; //通过类名引用静态数据成员
  cout<<a.volume( )<<endl;
  //调用volume函数，计算体积，输出结果
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态数据成员:让各对象之间的数据有了沟通的渠道，实现数据共享，因此可以不使用全局变量。全局变量破坏了封装的原则，不符合面向对象程序的要求。
  - 公用静态数据成员与全局变量的不同，静态数据成员的作用域只限于定义该类的作用域内。在此作用域内，可以通过类名和域运算符“::”引用静态数据成员，而不论类对象是否存在。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态成员函数:类声明中函数前面加static。

    static int volumn();
  - 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。如果要在类外调用公用的静态成员函数，要用类名和域运算符“::”。

    Box::volume( );
  - 也允许通过对象名调用静态成员函数。

    Box a;　　a.volume( );

    不代表函数是属于对象a的，而只是用a的类型而已。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态成员函数
  - 作用：不是为了对象之间的沟通，而是为了能处理静态数据成员。静态成员函数没有this指针。既然它没有指向某一对象，就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)。
  - 静态成员函数与非静态成员函数的根本区别是：非静态成员函数有this指针，而静态成员函数没有this指针。
  - 在C++程序中，静态成员函数主要用来访问静态数据成员，而不访问非静态成员。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态成员例：

```
# include <iostream>
using namespace std;
class Student  //定义Student类
{
public:
  student(int n,int a,float s):num(n),age(a),score(s){ }
  //定义构造函数
  void total( );
  static float average( ); //声明静态成员函数
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态成员例：

```
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; //累加总分
  count++; //累计已统计的人数
}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态成员例：

```
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]={ //定义对象数组并初始化
  Student(1001,18,70),Student(1002,19,78),Student(1005,20,98)};
  int n;
  cout <<"please input the number of students:";
  cin>>n; //输入需要求前面多少名学生的平均成绩
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 静态成员
  - 静态成员例：

```
for(int i=0;i<n;i++) //调用3次total函数
stud[i].total( );
cout<<"the average score of"<<n<<"students is"
<<Student::average( )<<endl;
//调用静态成员函数
return 0;
}
```

请思考：如果在average函数中引用stud[1]的非静态数据成员score，应该怎样修改？

请思考：如果不将average函数定义为静态成员函数行不行？程序能否通过编译？需要作什么修改？为什么要用静态成员函数？请分析其理由。

- Student类定义了两个静态数据成员sum和count，这是由于这两个数据成员的值需要进行累加，是由各对象元素共享的。无论对哪个对象元素而言，都是相同的，且始终不释放内存空间。
- total是公有的成员函数，可引用对象中的非静态数据成员score，也可引用静态数据成员sum和count。
- average是静态成员函数，可直接引用私有的静态数据成员sum和count 。
- 在main函数中，引用total函数要加对象名，引用静态成员函数average函数要用类名或对象名。

静态类成员课堂练习

- 用static数据成员维护类的对象个数
- 要求在构造函数中使用静态数据成员

第8讲 类与对象的进一步讨论
-对象的生存周期

❖ 友元

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。有一个例外——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。
- 友元函数：如果在本类以外的其他地方定义了一个函数(这个函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数)，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。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元函数例：

```
#include <iostream>
using namespace std;
class clock{
public:
  clock(int,int,int);
  friend void display(const clock &);
  //声明display函数为clock类的友元函数
private: //以下数据是私有数据成员
  int hour;
  int minute;
  int second;
};
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元函数例：

```
clock::clock(int h,int m,int s)
//构造函数，给hour,minute,second赋初值
{    hour=h;
     minute=m;
     second=s;
}

void display(Time& t) //这是友元函数，形参t是clock类对象的引用
{   cout<<t.hour<<":"<<t.minute<<":"<<t.second<<endl;    }

int main( ) //调用display函数，实参t1是clock类对象
{  clock t1(10,13,56);
   display(t1);
   return 0;}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元成员函数：可以是一般函数(非成员函数)--display；而且可以是另一个类中的成员函数。
  - 友元类：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。

    friend 类名；
  - 如在类A内将类声明为友元类，则在类A内写：

    friend B;

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元成员函数例：

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

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元成员函数例：

```
class Date //声明Date类
{
public:
  Date(int,int,int);
  friend void clock::display(const Date&); //声明clock中的display函数为友元成员函数
private:
  int month;
  int day;
  int year;
};
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元成员函数例：

```
clock::clock(int h,int m,int s) //类clock的构造函数
{  hour=h;
   minute=m;
  second=s;
}
void clock::display(const Date &d)
 //display的作用是输出年、月、日和时、分、秒
{ cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl;
   //引用Date类对象中的私有数据
  cout<<hour<<":"<<minute<<":"<<second<<endl;
  //引用本类对象中的私有数据
}
```

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元成员函数例：

```
Date::Date(int m,int d,int y) //类Date的构造函数
{  month=m;
   day=d;
   year=y;
}

int main( )
{  clock t1(10,13,56); //定义clock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;
}
```

- 运行时输出：
  12/25/2004　　(输出Date类对象d1中的私有数据)
  10:13:56　　(输出Time类对象t1中的私有数据)

- 对类作”提前引用”的声明，即在正式声明一个类之前，先声明一个类名，表示此类将在稍后声明。

- 只有在正式声明一个类后才能用它去定义类对象。编译系统在“见到”类体后，才确定应为对象预留多大空间。

- 在对一个类作了提前引用声明后，可以用该类的名字去定义指向该类型对象的指针变量或对象的引用变量。这是因为指针变量和引用变量本身的大小是固定的，与所指向的类对象的大小无关。

第8讲 类与对象的进一步讨论
-对象的生存周期

- 友元
  - 友元总结：

1)友元的关系是单向的而不是双向的；

A是B的友元(在B中声明friend A)，B不是A的友元

2)友元的关系不能传递；

A是B的友元，B是C的友元，A不是C的友元

3)在实际运用中，一般不把整个类声明为友元，只把需要的成员函数声明为友元函数，这样更安全；

4)友元是规则的破坏者(面向对象程序设计的基本原则之一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)；但是它有助于数据共享，能提高程序的效率。在使用友元时，要注意副作用，不要过多地使用。

前向引用声明

类的组合

- 类应该先声明，后使用
- 如果需要在某个类的声明之前，引用该类，则应进行前向引用声明。
- 前向引用声明只为程序引入一个标识符，但具体声明在其它地方。

前向引用声明举例

类的组合

```
class B;   //前向引用声明
class A
{  public:
        void f(B b);
};
class B
{  public:
        void g(A a);
};
```

前向引用声明注意事项

类的组合

- 使用前向引用声明虽然可以解决一些问题，但它并不是万能的。需要注意的是，尽管使用了前向引用声明，但是在提供一个完整的类声明之前，不能声明该类的对象，也不能在内联成员函数中使用该类的对象。请看下面的程序段：

```
class Fred;          //前向引用声明
class Barney {
   Fred x;  //错误：类Fred的声明尚不完善
};
class Fred {
   Barney y;
};
```

前向引用声明注意事项

```
class Fred;   //前向引用声明

class Barney {
public:
  void method()
  {
    x->yabbaDabbaDo();  //错误：Fred类的对象在定义之前被使用
  }
private:
  Fred* x;   //正确，经过前向引用声明，可以声明Fred类的对象指针
};

class Fred {
public:
  void yabbaDabbaDo();
private:
  Barney* y;
};
```

前向引用声明注意事项

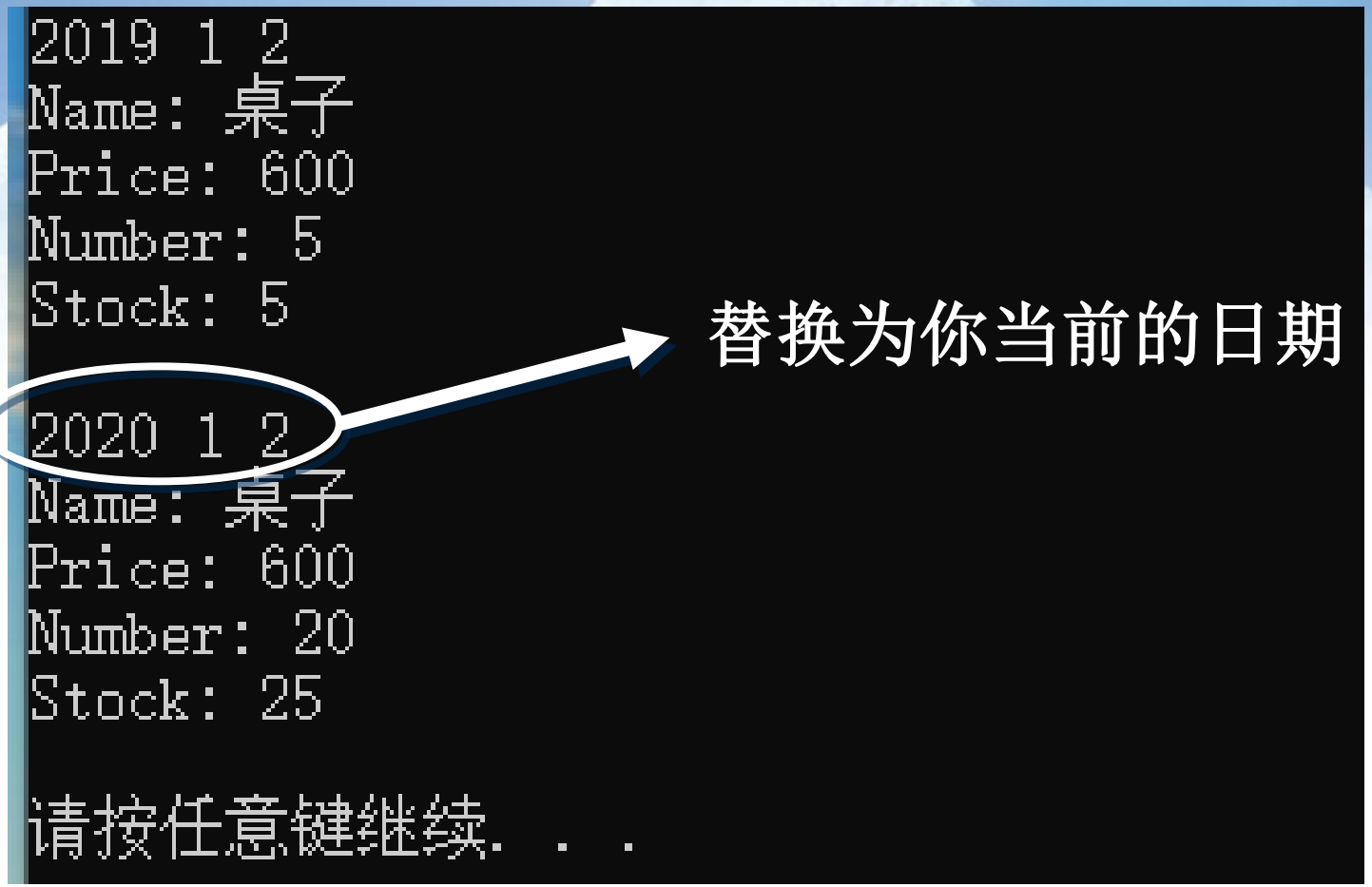
类的组合

- 应该记住：当你使用前向引用声明时，你只能使用被声明的符号，而不能涉及类的任何细节。

- 帮助家具公司设计管理系统。
- 该系统包含两个类，第一个是日期（Date）类，包含了private的数据成员month、date和year。
- 第二个是家具（Furniture）类。

1. 数据成员包括家具的名称（Name），价格（Price），该家具的数目指针（Number*），总库存量（Stock），其中，Stock为静态数据成员，该家具的数目采用动态内存分配方式。
2. 成员函数包括显示给定日期下的家具基本信息（Display(const Date &d)），库存量调整（ChangeStock），该家具数据调整（ChangeNum）。为系统写好拷贝构造函数、赋值运算符重载函数。

- 主函数使用指针数组处理furniture对象。家具1的信息包括（Name:桌子，Price: 600; Number: 5，进货日期：20190102），家具2拷贝家具1信息得到，并修改部分信息（数目：20，进货日期设置为当前日期），同时修改总库存量。输出如下图所示。


2019 1 2
Name: 桌子
Price: 600
Number: 5
Stock: 5
2020 1 2
Name: 桌子
Price: 600
Number: 20
Stock: 25
请按任意键继续. . .
替换为你当前的日期

C++程序设计（II）

浙江工业大学计算机学院