第3讲 关于类和对象的进一步讨论

3.1 构造函数

3.1.1 对象的初始化

类的数据成员是不能在声明类时初始化的。

可以在定义对象时对公有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
```

Time t1={14,56,30}; //将t1初始化为14:56:30

即：在一个花括号内顺序列出各公用数据成员的值，两个值之间用逗号分隔。

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。

3.1.2 构造函数的作用

C++提供了构造函数(constructor)来处理对象的初始化。

构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，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，而不能由用户任意命名，以便编译系统能识别它并把它作为构造函数处理。它不具有任何类型，不返回任何值。

构造函数的功能是由用户定义的，用户根据初始化的要求设计函数体和函数参数。

例3.1 在例9.3基础上定义构造成员函数。

```
#include <iostream>
using namespace std;
class Time
{public:
Time( ) //定义构造成员函数，函数名与类名相同
{hour=0; //利用构造函数对对象中的数据成员赋初值
minute=0;
sec=0;
}
void set_time( );          //函数声明
void show_time( );         //函数声明
private:
int hour;                  //私有数据成员
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( )
{Time t1; //建立对象t1，同时调用构造函数t1.Time( )
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
Time t2; //建立对象t2，同时调用构造函数t2.Time( )
t2.show_time( ); //显示t2的数据成员的值
return 0;}
```

程序运行的情况为：

10 25 54↙ **(从键盘输入新值赋给t1的数据成员)**

10:25:54 **(输出t1的时、分、秒值)**

0:0:0 **(输出t2的时、分、秒值)**

也可以只在类内对构造函数进行声明而在类外定义构造函数。在类内，改为下面一行：

```
Time( );          //对构造函数进行声明
```

在类外定义构造函数：

```
Time::Time( ) //类外定义构造函数，要加上类名Time和域限定符“::”
{hour=0;
minute=0;
sec=0;
}
```

有关构造函数的使用，有以下说明：

(1)构造函数没有返回值，因此也不需要在定义构造函数时声明类型，这是它和一般函数的一个重要的不同之点。

(2)构造函数不需用户调用，也不能被用户调用。

(3)在构造函数中主要实现对数据成员赋初值。

(4)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构造函数，但不执行初始化操作。

3.1.3 带参数的构造函数

采用带参数的构造函数，在调用不同对象的构造函数时，从外面将不同的数据传递给构造函数，以实现不同的初始化。

构造函数首部的一般格式为：

构造函数名(类型 1 形参1，类型2 形参2，…);

实参是在定义对象时给出的。定义对象的一般格式为

类名 对象名(实参1，实参2，…);

例3.2 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。编一个基于对象的程序，在类中用带参数的构造函数。

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int,int,int);        //声明带参数的构造函数
int volume( );           //声明计算体积的函数
 private:
int height;
int width;
int length;
};
Box::Box(int h,int w,int len)   //在类外定义带参数的构造函数
{height=h;
width=w;
length=len;
}
```

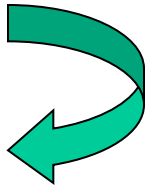
```
int Box::volume( )                    //定义计算体积的函数
{return(height*width*length);
}

int main( )
{Box box1(12,25,30); //建立对象，指定box1长、宽、高的值
cout<<"The volume of box1 is "<<box1.volume( )<<endl;
Box box2(15,30,21); //建立对象，指定box2长、宽、高的值
cout<<"The volume of box2 is "<<box2.volume( )<<endl;
return 0;
}
```

可以知道：

（1） 带参数的构造函数对应的实参在定义对象时给定。

（2） 用这种方法可以实现对不同的对象进行不同的初始化。

3.1.4 用参数初始化表对数据成员初始化



例3.2中定义构造函数可以改用以下形式：

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

这种写法方便、简练，尤其当需要初始化的数据成员较多时更显其优越性。

3.1.5 构造函数的重载

在一个类中可以定义多个构造函数，以便对类对象提供不同的初始化的方法，供用户选用。这些构造函数具有相同的名字，而参数的个数或参数的类型不相同。这称为构造函数的重载。

例3.3 在例3.2的基础上，定义两个构造函数，其中一个无参数，一个有参数。

```
#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，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;
}
```

还可以定义其他重载构造函数，其原型声明可以为

```
Box::Box(int h);                    //有1个参数的构造函数
Box::Box(int h,int w);              //有两个参数的构造函数
```

在建立对象时分别给定1个参数和2个参数。

说明：

(1) 无参的构造函数属于默认构造函数。一个类只能有一个默认构造函数。

(2)

尽管在一个类中可以包含多个构造函数，但是对于每一个对象来说，建立对象时只会执行其中一个构造函数，并非每个构造函数都被执行。

3.1.6 使用默认参数的构造函数

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int h=10,int w=10,int len=10); //声明构造函数时指定默认值
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值
{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;
}
```

即使在调用构造函数时没有提供实参值，不仅不会出错，而且还确保按照默认的参数值对对象进行初始化。

在一个类中定义了全部是默认参数的构造函数后，不能再定义重载构造函数。

3.2 析构函数

析构函数(destructor)也是一个特殊的成员函数，它的作用与构造函数相反，它的名字是类名的前面加一个“～”符号。

当对象的生命期结束时，会自动执行析构函数。一个类可以有多个构造函数，但只能有一个析构函数。

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析构函数体内指定的一些清理工作，使这部分内存可以被程序分配给新对象使用。

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。

例3.5 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
Student stud2(10011,"Zhang_fun",'m');
//定义对象stud2
stud2.display( ); //输出学生2的数据
return 0;
}
```

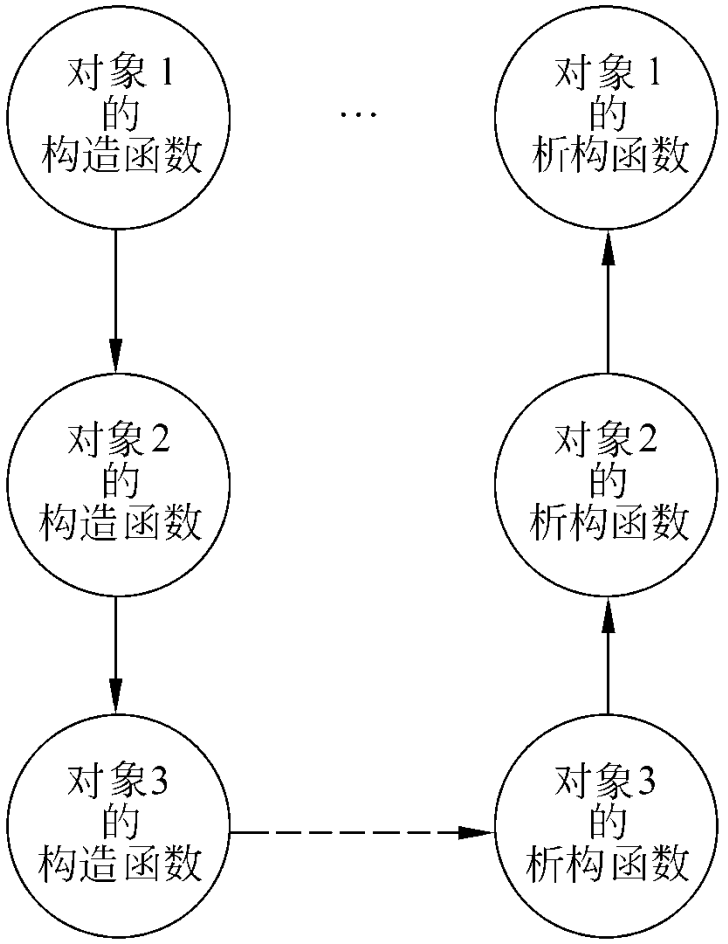
程序运行结果如下：

```
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.           (自动执行stud2的析构函数)
Destructor called.           (自动执行stud1的析构函数)
```

3.3 调用构造函数和析构函数的顺序

在一般情况下，调用析构函数的次序正好与调用构造函数的次序相反。

但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。



归纳一下什么时候调用构造函数和析构函数：

(1)在全局范围中定义的对象(即在所有函数之外定义的对象)，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。当main函数执行完毕或调用exit函数时(此时程序终止)，调用析构函数。

(2)如果定义的是局部自动对象(例如在函数中定义对象)，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，则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，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，才调用析构函数。

3.4 对象数组

在日常生活中，有许多实体的属性是共同的，只是属性的具体内容不同。例如一个班有50个学生，每个学生的属性包括姓名、性别、年龄、成绩等。如果为每一个学生建立一个对象，需要分别取50个对象名。用程序处理很不方便。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。例如

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

在建立对象数组时，分别调用构造函数，对每个元素初始化。

例3.6 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){ }
//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
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
};
cout<<"volume of a[0] is "<<a[0].volume( )<<endl;   //调用a[0]的volume函数
cout<<"volume of a[1] is "<<a[1].volume( )<<endl;   //调用a[1] 的volume函数
cout<<"volume of a[2] is "<<a[2].volume( )<<endl;   //调用a[2] 的volume函数
}
```

3.5 对象指针

3.5.1 指向对象的指针

在建立对象时，编译系统会为每一个对象分配一定的存储空间，以存放其成员。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。如果有一个类：

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

```
void Time∷get_time( )//定义成员函数
{cout<<hour<<":"<<minute<<":"<<sec<<endl;}
Time *pt; //pt为指向Time类对象的指针变量
Time t1; //定义t1为Time类对象
pt=&t1; //将t1的起始地址赋给pt
```

这样，pt就是指向Time类对象的指针变量，它指向对象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。如

*pt　　pt所指向的对象，即t1。

(*pt).hour　　pt所指向的对象中的hour成员，即t1.hour

pt->hour　　pt所指向的对象中的hour成员，即t1.hour

(*pt).get_time ( )　　调用pt所指向的对象中的get_time函数，即t1.get_time

pt->get_time ( )　　调用pt所指向的对象中的get_time函数，即t1.get_time

3.5.2 指向对象成员的指针

存放对象成员地址的指针变量就是指向对象成员的指针。

1. 指向对象数据成员的指针

与定义指向普通变量的指针变量方法相同。例如

```
int *p1;                    //定义指向整型数据的指针变量
```

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;
//将对象t1的数据成员hour的地址赋给p1，p1指向t1.hour
cout<<*p1<<endl;            //输出t1.hour的值
```

2. 指向对象成员函数的指针

定义指向对象成员函数的指针，要求在以下3方面都要匹配：

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。

例如：

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

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)(参数表列);

指向一个公用成员函数。如

p2=&Time::get_time;

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;

例3.7 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); //定义Time类对象t1
int *p1=&t1.hour;
//定义指向整型数据的指针变量p1，并使p1指向t1.hour
cout<<*p1<<endl; //输出p1所指的数据成员t1.hour
t1.get_time( ); //调用对象t1的成员函数get_time
Time *p2=&t1;
 //定义指向Time类对象的指针变量p2，并使p2指向t1
p2->get_time( ); //调用p2所指向对象(即t1)的get_time函数
void (Time::*p3)( );
//定义指向Time类公用成员函数的指针变量p3
p3=&Time::get_time;
//使p3指向Time类公用成员函数get_time
(t1.*p3)( );
//调用对象t1中p3所指的成员函数(即t1.get_time( ) )
}
```

3.5.3 this 指针

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); }
```

对象的成员函数为什么总是能使用对象自己的数据成员？？

即编译系统自动隐式地增加一个this指针。

Box a; a.volume(); 相当于a.volume(&a);

即调用对象的成员函数隐含：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。

3.6 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，这时可以使用**const**，即把有关的数据定义为常量。

3.6.1 常对象

类名 const 对象名 或者是 const 类名 对象名

1、对象中所有的数据成员的值都不能被修改。

2、不能调用常对象的非const型的成员函数。这是为了防止这些函数会修改常对象中的数据成员的值。

3、void get_time() const;//为常成员函数

可以访问常对象的数据成员，但不允许修改常对象中的数据成员的值。

4、mutable int count；//count声明为可变的数据成员

这样就可以用声明为const的成员函数来修改它的值。

3.6.2 常对象成员

可以将对象的成员声明为const，包括常数据成员和常成员函数。

1、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。例：

在类体中定义常数据成员hour：

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

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
```

2. 常成员函数

例：

void get_time( ) const;

//注意const的位置在函数名和括号之后

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

数据成员	非const成员函数	const成员函数
非const成员	可引用，可改变值	可引用，不可改变值
const成员	可引用，不可改变值	可以引用，不可改变值
const对象的成员	不允许引用和改变值	可以引用，不可改变值

3.6.3 指向对象的常指针

例：

```
Time t1(10,12,15),t2;              //定义对象
Time * const ptr1=&t1;             //指定ptr1指向t1
```

即：

类名 *const 指针变量名=对象地址；

指向对象的常指针变量的值不能改变，即始终指向同一个对象，但可以改变其所指向对象中数据成员的值。

如果想将一个指针变量固定与一个对象联系（即该指针变量始终指向一个对象），可以将它指定为const型的指针变量。

3.6.4 指向常对象的指针变量

首先了解指向常变量的指针变量(C语言学过)

const char *ptr;

注意const的位置在最左侧，它与类型名char紧连，表示指针变量ptr指向的char变量是常变量，不能通过ptr来改变其值的。

指向常对象的指针变量的概念和使用是与此类似。

指向常对象的指针变量定义：

const 类名 * 指针变量名;

3.6.5 对象的常引用

常引用：指向的内容不可变，和实参的地址相同。

1)变量的常引用定义：

const 类型名　&引用名=变量名;

2)对象的常引用定义：

const 类名　&引用名=对象名;

例：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);          // t1是Time类对象

fun(t1); //实参是Time类对象，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;
}
```

如果不希望在函数中修改实参t1的成员值，可以把引用变量t声明为const(常引用)，函数原型为

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

3.6.6 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**

3.7 对象的动态建立和释放

前面介绍了用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。如果已经定义了一个Box类，可以用下面的方法动态地建立一个对象：

```
new Box;
```

用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的值，即所分配的内存空间的起始地址。需要定义一个指向本类的对象的指针变量来存放该地址。如

```
Box *pt; //定义一个指向Box类对象的指针变量pt
pt=new Box; //在pt中存放了新建对象的起始地址
```

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
```

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

3.8 对象的赋值和复制

3.8.1 对象的赋值

同类的对象之间可以互相赋值，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。

对象名1 = 对象名2;

注意：对象名1和对象名2必须属于同一个类。

例 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; //定义两个对象box1和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;
}
```

对象的赋值只对其中的数据成员赋值，而不对成员函数赋值

运行结果如下：

The volume of box1 is 11250

The volume of box2 is 11250

说明：

对象的赋值只对其中的数据成员赋值，而不对成员函数赋值。

3.8.2 对象的复制

用已有的对象box1去克隆出一个新对象box2:

Box box2(box1);

上面括号中给出的参数不是一般的变量，而是对象。

其一般形式为

类名 对象2(对象1);

用对象1复制出对象2。

会自动调用一个特殊的构造函数——复制构造函数(copy constructor)。

复制构造函数也是构造函数，可自行定义：

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

特殊之处：这个参数是本类的对象

它只有一个参数，这个参数是本类的对象(不能是其他类的对象)，而且采用对象的引用的形式(一般约定加const声明，使参数值不能改变)。

Box box2(box1);

实际上也是建立一个新对象box2。由于在括号内给定的实参是对象，因此编译系统就调用复制构造函数(它的形参也是对象)，而不会去调用其他构造函数。

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++还可用赋值号代替括号，如

Box box2=box1;　　//用box1初始化box2

也可在一个语句中进行多个对象的复制。如

Box box2=box1,box3=box2;

按box1来复制box2和box3，其作用都是调用复制构造函数。

注意：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。

3.9 静态成员

3.9.1 静态数据成员

例

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

如果希望各对象中的height的值是一样的，就可以把它定义为静态数据成员，这样它就为各对象所共有，而不只属于某个对象的成员，所有对象都可以引用它。

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。

几点说明：

(1) 静态数据成员是在所有对象之外单独开辟空间。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。

(2) 静态数据成员不随对象的建立而分配空间，也不随对象的撤销而释放。

(3) 静态数据成员只能在类体外进行初始化，不必在初始化语句中加static。如

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

静态数据成员未赋初值，系统会自动赋初值0。

(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。请观察下面的程序。

例3.10 引用静态数据成员。

```
#include <iostream>
using namespace std;
class Box
{public:
Box(int,int);
int volume( );
static int height; //把height定义为公用的静态的数据成员
int width;
int length;
};
Box∷Box(int w,int len)        //通过构造函数对width和length赋初值
{width=w;
length=len;
}
```

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

上面3个输出语句的输出结果相同(都是10)。

(5) 有了静态数据成员，各对象之间的数据有了沟通的渠道，实现数据共享。全局变量破坏了封装的原则，不符合面向对象程序的要求，要少用。

3.9.2 静态成员函数

例

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。在类外调用公用的静态成员函数，如：

Box∷volume( );

也可以通过对象名调用静态成员函数，如：

a.volume( );

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++程序中最好养成这样的习惯： 只用静态成员函数引用静态数据成员，而不引用非静态数据成员。这样思路清晰，逻辑清楚，不易出错。如：

```
cout<<height<<endl;
//若height已声明为static，则引用本类中的静态成员，合法
cout<<width<<endl;
 //若width是非静态数据成员，不知是哪个对象的，不合法
cout<<a.width<<endl; //引用本类对象a的非静态成员,合法
```

例 静态成员函数的应用。

```
#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
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
}
float  Student::average( ) //定义静态成员函数，不定义成静态的就跟对象有关
{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
 for(int i=0;i<n;i++)                    //调用3次total函数
stud[i].total( );                        //跟对象有关
 cout<<"the average score of "<<n<<" students is
"<<Student::average( )<<endl;//调用静态成员函数,跟对象无关
return 0;
}
```

运行结果为

```
please input the number of students:3↙
the average score of 3 students is 82.3333
```

3.10 友元

一般地，在一个类中可以有公用的(public)成员和私有的(private)成员。在类外可以访问公用成员，只有本类中的函数可以访问本类的私有成员。

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。

3.3.1 友元函数

不属于任何类的普通函数，或者其他类的成员函数，在本类的类体中用**friend**对其进行声明，此函数就称为本类的友元函数。

友元函数可以访问本类中的私有成员。

1. 将普通函数声明为友元函数

例 友元函数的简单例子。

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

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

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

程序输出结果如下：

10:13:56

不是Time类的成员函数！

由于声明了display是Time类的friend函数，所以display函数可以引用Time中的私有成员hour, minute, sec。

2. 友元成员函数

例 友元成员函数的简单应用。

```
#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);//构造函数
void display(Date &);    //display是成员函数，形参是Date类对象的引用
 private:
int hour;
int minute;
int sec;
};
```

```
class Date                          //声明Date类
{public:
Date(int,int,int);
friend void Time::display(Date &);    //声明Time类的display函数为友元成员函数
 private:
int month;
int day;
int year;
};
Time::Time(int h,int m,int s)    //类Time的构造函数
{hour=h;
minute=m;
sec=s;
}
void Time::display(Date &d) //Time类的成员函数display定义
{cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl;   //引用Date类对象中的私有数据
cout<<hour<<":"<<minute<<":"<<sec<<endl; //引用本类对象中的私有数据
}
```

```
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);          //定义Time类对象t1
Date d1(12,25,2004);         //定义Date类对象d1
t1.display(d1);              //调用t1中的display函数，实参是Date类对象d1
return 0;
}
```

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)
```

说明：

1、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(如在本例中，定义了Date类对象的引用变量)。这是因为指针变量和引用变量本身的大小是固定的，与它所指向的类对象的大小无关。

2、在对一个类作了提前引用声明后，还不可以用该类的名字去定义一个对象，因为定义对象要分配空间，应该分配多少空间无法确定。

3. 普通函数和成员函数可以被多个类声明为友元函数，这样就可以引用多个类中的私有数据。

例如， 可以将上例中的display函数不放在Time类中，而作为类外的普通函数，然后分别在Time和Date类中将display声明为朋友。在主函数中就可以调用display函数来分别引用Time和Date两个类的对象的私有数据，输出年、月、日和时、分、秒。

3.3.2 友元类

可以将一个类(例如B类)声明为另一个类(例如A类)的“朋友”。这时B类就是A类的友元类。

友元类B中的所有函数都是A类的友元函数，可以访问A类中的所有成员。

在A类的定义体中用以下语句声明B类为其友元类：

```
friend B;
```

在实际工作中一般并不把整个类声明为友元类，而只将确实有需要的成员函数声明为友元函数，这样更安全一些。

友元可以访问其他类中的私有成员，不能不说这是对封装原则的一个小的破坏。但是它能有助于数据共享，能提高程序的效率，必要时可使用友元。

3.11 类模板

两个或多个类，其功能是相同的，仅仅是数据类型不同，可考虑使用类模板。

例 通过调用成员函数max和min得到两个整数中的大者和小者：

```
class Compare_int
{public:
Compare_int(int a,int b){x=a;y=b;}        //构造函数
int max( )
{return(x>y)?x:y;}
int min( )
{return(x<y)?x:y;}
private:
int x,y;
};
```

浮点数(float型)数据，需要另外声明一个类：

```
class Compare_float
{public:
Compare_float(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;
}
```

对以上两个类可以综合写出以下的类模板：

```
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;
};
```

声明一个模板，自己指定虚拟类型名如numtype

用类模板定义对象的方法是：

Compare <int> cmp(4,7);

即在类模板名之后在尖括号内指定实际的类型名，在进行编译时，编译系统就用int取代类模板中的类型参数numtype，这样就把类模板实例化。

例 声明一个类模板，利用它分别实现两个整数、浮点数和字符的比较，求出大数和小数。

```
#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;
}
```

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，形式为：

```
template<class numtype>        //需要添加此句，numtype为虚拟类型参数
numtype Compare<numtype>::max( )  //此句句首的numtype是函数类型
{return (x>y)?x:y;}
```

说明：

类模板的类型参数可以有多个，每个类型前面都必须加class，如

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

在定义对象时分别代入实际的类型名，如

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

作业：

每两周交一次，提交至ftp://172.21.85.11,用户名:16kjhomework,密码:11111111，提交要求请登录FTP查看。

- **P120-121**
- **第5,9,11题**