第3章 类与对象

类构成了实现C++面向对象程序设计的基础。类用来定义对象的属性和行为，类是C++封装的基本单元。本章将结合实例详细讨论类及对象。

3.1 类

C++语言的类就是一种用户自己定义的数据类型，和其他数据类型不同的是，组成这种类型的不仅可以有数据，而且可以有对数据进行操作的函数。

C++规定，任何数据类型都必须先定义后使用，类也不例外。

3.1.1 类的定义

为了在程序中创建对象，必须首先定义类。C++语言用保留字class定义一个类，一般形式为：

```
class 类名
{
    public:
        <公有数据和函数>
    protected:
        <保护数据和函数>
    private:
        <私有数据和函数>
};
```

其中，类名必须是一个有效的C++标识符，不能是C++语言的关键字，类的名字一般都以大写字母开始。类所说明的内容以一对花括号括住，构成类体。右花括号后的分号“;”作为类声明的结束标志是不能漏掉的。类中定义的数据和函数分别称为数据成员和成员函数。

数据成员用来描述对象的属性，可以像声明普通变量的方式来声明，并且允许是任何数据类型，包括用户自定义的类类型(但不允许是当前正在定义的类，除非使用指针形式)。类中数据成员在声明时不允许初始化。

成员函数用来描述对象的行为，与普通函数一样，它可以重载，可以使用默认参数，还可以声明为内联函数。

3.1.2 类成员的访问控制

关键字public、protected和private均用于控制类中成员在程序中的可访问性。关键字

public、protected 和 private 以后的成员的访问权限分别是公有、保护和私有的。所有成员默认定义为 private 的,但为了提高程序的可读性,不主张使用这种默认定义方式。

公有成员定义了类的外部接口。私有成员是被隐藏的数据,只有该类的成员函数或友元函数才可以引用它。保护成员具有公有成员和私有成员的双重性质,可以被该类或派生类的成员函数或友元函数引用。

关键字 public、protected 和 private 出现的顺序和次数可以是任意的。但初学者应该养成一种良好的习惯,将访问控制方式相同的成员放在一起,并且先列出 public 成员,再列出 protected 和 private 成员。对一个类的使用者来说,最关心的是那些可被访问的外部接口。

3.1.3 成员函数的实现

成员函数的实现,可以放在类体内,见例 3.1。也可以放在类体外,但必须在类体内给出原型说明,见例 3.2。放在类体内定义的函数被默认为内联函数,而放在类体外定义的函数是一般函数,如果要定义为内联函数则需在前面加上关键字 inline。

与普通函数不同的是,成员函数是属于某个类的,在类体外定义成员函数的一般形式为:

```
<返回类型><类名>::<成员函数名>(<参数说明>)
{
    函数体
}
```

其中"::"称为作用域运算符,"<类名>::"表明其后的成员函数是在这个类中声明的。在"函数体"中可以直接访问类中说明的成员,以描述该成员函数对它们所进行的操作。

【例 3.1】 定义一个点类(Point),示例类体内实现成员函数。

```
// 程序 Li3_1.cpp
// 定义一个点类(Point),示例类体内实现成员函数
#include<iostream>
using namespace std;
class Point
{
   public:
       void setxy(int x,int y)
       {
          X = x;
          Y = y;
       }
       void displayxy()
          {
              cout<<"("<<X<<","<<Y<<")"<<endl;
          }
   private:
      int X,Y;
};
```

【例 3.2】 定义一个点类(Point),示例类体外实现成员函数。

```
// 程序 Li3_2.cpp
// 定义一个点类(Point),示例类体外实现成员函数
// 点类的界面部分
#include<iostream>
using namespace std;
class Point
{
   public:
        void setxy(int,int);
        void displayxy();
   private:
        int X,Y;
};
// 点类的实现部分
void Point::setxy(int x,int y)
{
   X = x;
   Y = y;
}
void Point::displayxy()
{
   cout<<"("<<X<<","<<Y<<")"<<endl;
}
```

提示:在一个类的定义中,可以将一部分成员函数的实现放在类体内,将一部分成员函数的实现放在类体外。一般将代码少的成员函数的实现放在类体内。

为了减少代码的重复,加快编译速度,在大型程序设计中,C++的类结构常常被分成两部分:一部分是类的界面,另一部分是类的实现。在类的界面中仅包括类的所有数据成员以及成员函数的函数原型,放在头文件中,供所有相关应用程序共享。而对于类的实现,即成员函数实现则放在与头文件同名的源文件中,便于修改。这种做法还有利于为一个类的同一界面提供不同的内部实现。

【例 3.3】 按类的界面与类的实现两部分来重新定义一个点类(Point)。

```
// 程序 Li3_3.h
// 点类的界面部分
class Point
{
   public:
      void setxy(int,int);
      void displayxy();
   private:
      int X,Y;
};
```

```
// 点类的实现部分
#include<iostream>
#include "Li3_3.h"
using namespace std;
void Point::setxy(int x,int y)
{
   X = x;
   Y = y;
}
void Point::displayxy()
{
   cout<<"("<<X<<","<<Y<<")"<<endl;
}
```

3.2 对　　象

类是一种程序员自定义的数据类型称为类类型,程序员可以使用这个新类型在程序中声明新的变量,具有类类型的变量称为对象。

3.2.1 对象的声明

类和对象的关系就相当于基本数据类型与它的变量的关系,所以很多书,包括本书有时将普通变量和类类型的对象都统称为对象。

对象的声明与普通变量非常相似,一般格式为:

```
<类名><对象名表>;
```

其中,<类名>是所定义的对象所属类的名字。<对象名表>中可以是一般的对象名,也可以是指向对象的指针名或引用名,还可以是对象数组名。指向对象的指针称为对象指针,对象的引用称为对象引用。

例如,声明类 Point 的对象如下所示:

```
Point p1,p2, * pdate,p[3], &rp = p1;
```

其中,Point 是类名,p1 和 p2 是两个一般对象名,pdate 是指向类 Point 的对象指针名,p[3]是对象数组,该数组是具有 3 个元素的一维数组,每个数组元素是类 Point 的一个对象,rp 是一个对象引用名,它被初始化后,rp 是对象 p1 的引用。

3.2.2 对象的创建和销毁

对象在创建时,往往要考虑“对象的数据存放在何处”、“如何控制对象的生命周期”等问题(所谓生命周期,是指对象从诞生到结束的这段时间)。在程序运行时,通过为对象分配存储空间来创建对象。创建对象时,类被用作样板,对象称为类的实例(Instance),所以有时对象与实例两个概念会混用。为对象分配存储空间主要有静态分配和动态分配两种方式。堆对象是在程序运行时根据需要随时可以被创建或删除的对象,只有堆对象采用动态分配

方式。

静态分配方式，在声明对象时就分配存储空间，在对象生命期结束时收回所分配存储空间。在这种分配方式下，对象的创建和销毁是由程序本身决定的。例如声明：

```
Point p1,p2,p[3];
```

时，即创建对象 p1，p2 和对象数组 p。

动态分配方式，如果需要建立新的对象，就要使用运算符 new 在堆中为其分配内存空间；当对象使用完毕需要销毁时，要使用运算符 delete 来释放它所占用的自由内存空间。在这种分配方式下，对象的创建和销毁是由程序员决定的。例如声明：

```
Point *pdate;
```

时，只创建了对象指针 pdate，并没有创建 pdate 所指向的对象。

注意：对象引用不分配存储空间。

3.2.3 对象成员的访问

一个对象的成员就是该对象所属类的成员。对象的成员与它所属类的成员一样，有数据成员和成员函数。声明了对象，我们就可以访问对象的公有成员了。

用成员选择运算符“.”访问一般对象的成员，语法格式如下：

```
<对象名>.<数据成员名>
<对象名>.<成员函数名>(<参数表>)
```

用成员选择运算符“.”访问对象引用的成员，语法格式如下：

```
<对象引用名>.<数据成员名>
<对象引用名>.<成员函数名>(<参数表>)
```

用成员选择运算符“->”访问对象指针的成员，语法格式如下：

```
<对象指针名>->数据成员名>
<对象指针名>->成员函数名>(<参数表>)
```

或者

```
(*<对象指针名>).<数据成员名>
(*<对象指针名>).<成员函数名>(<参数表>)
```

【例 3.4】 利用例 3.3 中定义的类，示例对象的声明和对成员访问方法。

```
// 程序 Li3_4.cpp
// 对象的声明和对成员的访问
#include<iostream>
#include "Li3_3.h"
using namespace std;
void Point::setxy(int x,int y)
{
   X = x;
   Y = y;
```

```
}
void Point::displayxy()
{
   cout<<"("<<X<<","<<Y<<")"<<endl;
}
int main()
{
   Point p1, * p2;
   p1.setxy(3,4);
   cout<<"第1个点的位置是:";
   p1.displayxy();
   p2 = &p1;
   p2->setxy(5,6);
   cout<<"第2个点的位置是:";
   ( * p2).displayxy();
   return 0;
}
```

程序输出结果为:

```
第1个点的位置是:(3,4)
第2个点的位置是:(5,6)
```

程序分析:该程序声明了一个普通对象 p1,一个对象指针 p2。p1 用成员选择运算符“.”访问成员函数 setxy()和 displayxy()。p2 用成员选择运算符“->”访问成员函数 setxy(),用另外一种形式访问成员函数 displayxy()。

3.3 构造函数与析构函数

一个对象的数据成员反映了该对象的内部状态,但在类声明中,无法用表达式初始化这些数据成员,因而数据成员的初始值是不确定的,导致声明一个对象时,该对象的初始状态不确定。

就像声明基本类型变量时可以同时进行初始化一样,在声明对象的时候,也可以同时对它的数据成员赋初值。在声明对象的时候进行的数据成员设置,称为对象的初始化。在特定对象使用结束时,还经常需要进行一些清理工作。C++程序中的初始化和清理工作,分别由两个特殊的成员函数来完成,它们就是构造函数和析构函数。

3.3.1 构造函数

1. 构造函数的特点

构造函数是一种特殊的成员函数,对象的创建和初始化工作可以由它来完成,其格式如下:

```
<类名>::<类名>(<形参表>)
{
```

```
    <函数体>
}
```

构造函数应该被声明为公有函数，因为它是在创建对象的时候被自动调用。构造函数有如下特点：

- 它的函数名与类名相同。
- 它可以重载。
- 不能指定返回类型，即使是 void 类型也不可以。
- 它不能被显式调用，在创建对象的时候被自动调用。

2. 默认构造函数

默认构造函数就是无参数的构造函数。既可以是自己定义的，也可以是编译系统自动生成的。

前面的例 3.4、例 3.5 中都没有定义构造函数，那么它们的对象是怎么被创建的呢？事实上，当没有为一个类定义任何构造函数的情况下，编译系统就会自动生成一个无参数、空函数体的默认构造函数。其格式如下：

```
<类名>::<类名>()
{
}
```

在程序中声明一个没有使用初始值的对象时，系统自动调用默认构造函数创建该对象，当该对象是外部的或静态的时，它的所有数据成员被初始化为 0 或空。当该对象是自动的时，它的所有数据成员的值是无意义的。

【例 3.5】 修改例 3.2 中定义的类，示例构造函数的用法。

```
// 程序 Li3_5.cpp
// 示例构造函数的用法
// 点类的界面部分
#include<iostream>
using namespace std;
class Point
{
   public:
      Point();                    // 默认构造函数
      Point(int);                 // 有 1 个参数构造函数
      Point(int,int);             // 有两个参数构造函数
      void displayxy();
   private:
      int X,Y;
};
// 点类的实现部分
Point::Point ()
{
   X = 7;
   Y = 8;
```

```
    cout<<"Default constructor is called!";
    displayxy();
}
Point::Point (int x)
{
    X = x;
    Y = 8;
    cout<<"Constructor is called!";
    displayxy();
}
Point::Point (int x,int y)
{
    X = x;
    Y = y;
    cout<<"Constructor is called!";
    displayxy();
}
void Point::displayxy()
{
    cout<<"("<<X<<","<<Y<<")"<<endl;
}
int main()
{
    Point p1(3,4),p2[2] = {5,6},p3;
    return 0;
}
```

提示：由于例3.5已经自己定义了构造函数，编译系统自动不会生成默认构造函数。为了创建没有初始值的对象p3，必须自己定义默认构造函数。

程序输出结果为：

```
Constructor is called! (3,4)
Constructor is called! (5,8)
Constructor is called! (6,8)
Default constructor is called! (7,8)
```

程序分析：该程序声明了3个对象p1、p2和p3，类中有3个构造函数。创建对象p1时调用有两个参数构造函数，输出第1行结果。创建对象p2时调用有1个参数构造函数，由于对象p2是对象数组，每个数组元素被创建时都要调用构造函数，所以1个参数构造函数被调用了两次，输出第2、3行结果。创建对象p3时调用默认构造函数，输出第4行结果。

3.3.2 析构函数

1. 析构函数的特点

析构函数也是一种特殊的成员函数，它的作用是在对象消失时执行一项清理任务，例

如，可以用来释放由构造函数分配的内存等。其格式如下：

```
<类名>::~<类名>()
{
  <函数体>
}
```

析构函数也只能被声明为公有函数，因为它是在释放对象的时候被自动调用。析构函数有如下特点：

- 析构函数的名字同类名，与构造函数名的区别在于析构函数名前加“~”，表明它的功能与构造函数的功能相反。
- 析构函数没有参数，不能重载，一个类中只能定义一个析构函数。
- 不能指定返回类型，即使是 void 类型也不可以。
- 析构函数在释放一个对象时候被自动调用。与构造函数不同的是，它能被显式调用，但不提倡。

2. 默认析构函数

如果一个类中没有定义析构函数时，系统将自动生成一个默认析构函数，其格式如下：

```
<类名>::~<类名>()
{
}
```

前面的例子中都没有定义析构函数，调用的就是系统自动生成的默认析构函数。

【例 3.6】 修改例 3.2 中定义的类，示例析构函数的用法。

```
// 程序 Li3_6.cpp
// 示例析构函数的用法
// 点类的界面部分
#include<iostream>
using namespace std;
class Point
{
   public:
      Point(int,int);
      void displayxy();
      ~Point();
   private:
      int X,Y;
};
// 点类的实现部分
Point::Point (int x,int y)
{
   X = x;
   Y = y;
   cout<<"Constructor is called!";
   displayxy();
```

```
}
void Point::displayxy()
{
   cout<<"("<<X<<","<<Y<<")"<<endl;
}
Point::~Point()
{
   cout<<"Destructor is called!";
   displayxy();
}
int main()
{
   Point p1(3,4),p2(5,6);
   return 0;
}
```

程序输出结果为：

```
Constructor is called! (3,4)
Constructor is called! (5,6)
Destructor is called! (5,6)
Destructor is called! (3,4)
```

程序分析：该程序声明了两个对象 p1 和 p2。创建对象时调用构造函数，输出前两行结果。当程序结束时，释放两个对象，析构函数被调用，输出后两行结果。从结果可以看出，构造函数的调用顺序与声明对象的顺序一致，而析构函数的调用顺序与构造函数的调用顺序正好相反。

如果声明了对象数组，数组元素创建了多少，最后就要释放多少，即析构函数要被调用多少次。

3.3.3 拷贝构造函数

1. 拷贝构造函数的特点

拷贝构造函数是一种特殊的构造函数，它的作用是用一个已经存在的对象去初始化另一个对象，为了保证所引用的对象不被修改，通常把引用参数声明为 const 参数。其格式如下：

```
<类名>::<类名>(const <类名>&<对象名>)
{
  <函数体>
}
```

拷贝构造函数具有一般构造函数的特性，特点如下：

- 拷贝构造函数名字与类名相同，并且不能指定返回类型。
- 拷贝构造函数只有一个参数，并且该参数是该类的对象的引用。
- 它不能被显式调用，在以下 3 种情况下都会被自动调用：

① 当用类的一个对象去初始化该类的另一个对象时。

② 当函数的形参是类的对象，进行形参和实参结合时。

③ 当函数的返回值是类的对象，函数执行完成返回调用者时。

2. 默认拷贝构造函数

如果一个类中没有定义拷贝构造函数，则系统自动生成一个默认拷贝构造函数。该函数的功能是将已知对象的所有数据成员的值拷贝给对应的对象的所有数据成员。

【例 3.7】 分析下面程序的执行过程，了解拷贝构造函数的用法。

```
// 程序 Li3_7.cpp
// 了解拷贝构造函数的用法
// 点类的界面部分
#include<iostream>
using namespace std;
class Point
{
   public:
      Point(int = 0,int = 0);
      Point(const Point&);
      void displayxy();
      ~Point();
   private:
      int X,Y;
};
// 点类的实现部分
Point::Point (int x,int y)
{
   X = x;
   Y = y;
   cout<<"Constructor is called!";
   displayxy();
}
Point::Point(const Point& p)
{
   X = p.X;
   Y = p.Y;
   cout<<"Copy constructor is called!";
   displayxy();
}
Point::~Point()
{
   cout<<"Destructor is called!";
   displayxy();
}
void Point::displayxy()
{
```

```
    cout<<"("<<X<<","<<Y<<")"<<endl;
}
Point func(Point p)
{
    int x = 10 * 2;
    int y = 10 * 2;
    Point pp(x,y);
    return pp;
}
int main()
{
    Point p1(3,4);
    Point p2 = p1;  // 用类的一个对象去初始化该类的另一个对象
    p2 = func(p1);  // 函数的形参是类的对象,且函数的返回值也是类的对象
    return 0;
}
```

程序输出结果为:

```
Constructor is called! (3,4)
Copy constructor is called! (3,4)
Copy constructor is called! (3,4)
Constructor is called! (20,20)
Copy constructor is called! (20,20)
Destructor is called! (20,20)
Destructor is called! (3,4)
Destructor is called! (20,20)
Destructor is called! (20,20)
Destructor is called! (3,4)
```

程序分析:

(1) 从程序运行后的输出结果中,可以清楚地看出该程序共调用了3次拷贝构造函数。第1次是在执行语句

```
Point p2 = p1;
```

时调用拷贝构造函数,用对象p1创建新对象p2。第2次是在执行语句

```
p2 = func(p1);
```

时,在func()中,调用拷贝构造函数,当用实参p1初始化形参p时,给对象p赋值。第3次是在执行语句return pp;时,调用拷贝构造函数,用对象pp创建一个临时对象。

(2) 临时对象是在执行到函数func()中的语句

```
return pp;
```

时被创建的。由于pp是被定义在func()函数内的局部对象,当退出func()时,pp将被释放。因此,在退出该函数之前,用pp创建一个临时对象,保存pp的数据。在主函数main()中,在临时对象被释放之前,将它的内容赋值到对象p2中。一般规定,临时对象在整个创建它的

外部表达式范围内有效。因此，该程序中的临时对象是在执行完语句

```
p2 = func(p1);
```

之后被释放的。在该程序输出结果中，从前向后数，第 3 个“Destructor is called!”是释放临时对象时调用析构函数所留下的信息。

临时对象所起的作用如图 3.1 所示，它起一个暂存作用。

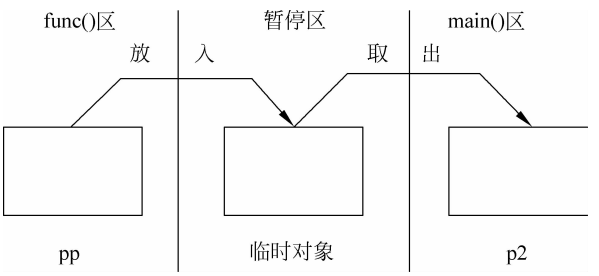

图 3.1　临时对象所起的作用

3.4　this 指针

同一类的各个对象创建后，都在类中产生自己成员的副本。而为了节省存储空间，每个类的成员函数只有一个副本，成员函数由各个对象调用。那么对象在副本中如何与成员函数建立关系呢？C++为成员函数提供了一个称为 this 的指针，当创建一个对象时，this 指针就初始化指向该对象。当某一对象调用一个成员函数时，this 指针将作为一个变量自动传给该函数。所以，不同的对象调用同一个成员函数时，编译器根据 this 指针来确定应该引用哪一个对象的数据成员。

this 指针是由 C++编译器自动产生且较常用的一个隐含对象指针，它不能被显式声明。this 指针是一个局部量，局部于某个对象。this 指针是一个常量，它不能作为赋值、递增、递减等运算的目标对象。此外，只有非静态类成员函数才拥有 this 指针，并通过该指针来处理对象。实际中，通常不去显式使用 this 指针引用数据成员和成员函数。

【例 3.8】　分析下面程序，体会 this 指针的隐式使用。

```
// 程序 Li3_8.cpp
// 体会 this 指针的隐式使用
#include<iostream>
using namespace std;
class Point
{
   public:
      Point(int = 0, int = 0);
      void displayxy();
   private:
      int X,Y;
};
```

```
Point::Point(int x,int y)
{
   X = x;
   Y = y;
}
void Point::displayxy()
{
   cout<<X<<endl;                    // 相当于 cout<<this->X<<endl;
   cout<<Y<<endl;                    // 相当于 cout<<this->Y<<endl;
}

int main()
{
   Point obj1(10,20), obj2(8,9), *p;
   p = &obj1;                        // p 指向对象 obj1
   p->displayxy();
   p = &obj2;                        // p 指向对象 obj2
   p->displayxy();
   return 0;
}
```

程序输出结果为：

```
10
20
8
9
```

程序分析：当语句 Point obj1(10,20), obj2(8,9) 被执行时，obj1.X、obj1.Y 和 obj2.X、obj2.Y 分别被赋值，当语句 p->displayxy();被执行时，系统根据 this 指针当时所指的对象是 obj1 还是 obj2，决定输出哪个 X、Y。实际上，编译器所认识的成员函数 displayxy()的形式为：

```
void Point::displayxy()
{
  cout<<this->X<<endl;
  cout<<this->Y<<endl;
}
```

当一个成员函数内需要标识被该成员函数操作的对象时，就需要显式使用 this 指针。下面通过例子来说明 this 指针的这种用法。

【例 3.9】 分析程序结果，体会 this 指针的显式使用。

```
// 程序 Li3_9.cpp
// 体会 this 指针的显式使用
#include<iostream>
using namespace std;
```

```
class Point
{
   public:
      Point(int x,int y) {X = x;Y = y;}     // 有参构造函数
      Point(){X = 0;Y = 0;}                 // 无参构造函数
      void copy(Point& obj);
      void displayxy();
   private:
      int X,Y;
};
void Point::copy(Point& obj)
{
   if (this! = &obj)                        // this 指针的显式使用,避免无意义的更新
    * this = obj;
}
void Point::displayxy()
{
   cout<<X<<"";
   cout<<Y<<endl;
}

int main()
{
   Point obj1(10,20), obj2;
   obj2.copy(obj1);
   obj1.displayxy();
   obj2.displayxy();
   return 0;
}
```

程序输出结果为：

```
10 20
10 20
```

程序分析：成员函数 copy()用一个 Point 类的对象(参数 obj 所引用的对象)的值更新正在操作的对象,为避免下面无意义的更新：

```
obj2.copy(obj2);
```

成员函数 copy()使用了表达式：

```
this! = &obj
```

来判断这种情况。在实际编程中,这是一种常用的检测手段。

this 指针是一个常量,它不能作为赋值、递增、递减等运算的目标对象。例如,下面的程序,由于试图给 this 赋值,所以是错误的。

```
void Point::copy(Point& obj)
```

```
{
  if (this! = &obj)
  this = &obj; // 错误,因为不能给常量 this 赋值
}
```


3.5 子对象和堆对象

在实现一个类时,可以利用已有的类型实现新的复杂的类型。这种复杂类型的对象由可标识的子对象构成。就像火车对象可以看成是由火车头、车厢和车轮等组成一样。

在设计程序时,可以确定有些什么类型的对象,但并不知道在程序运行时有多少对象存在,另外,从虚拟机制的考虑出发,也可以更经济有效地使用存储,所以,更经常的情况是,在运行需要对象时,程序使用特定的操作建立对象,在对象使用完之后,再使用特定的操作删除对象。这样创建的对象就称为堆对象。

3.5.1 子对象

1. 子对象的声明

一个对象作为另一个类的成员时,该对象称为类的子对象。子对象实际上是某个类的数据成员。说明子对象的一般形式为:

```
class<X>{
                …
     <类名 1>    <子对象 1>
     <类名 2>    <子对象 2>
      ⋮           ⋮
     <类名 n>    <子对象 n>
};
```

例如:

```
class A
{
  ⋮
};
class B
{
  ⋮
  private:
    A a;
  ⋮
};
```

其中,类 B 中成员 a 就是一个子对象,因为 a 是类 A 的一个对象。

对上述类声明,问题的关键是,在建立该类型的对象时,怎样初始化子对象。下面主要针对这个问题展开讨论。

2. 子对象的初始化

为初始化子对象,X 的构造函数要调用这些对象成员所在类的构造函数,于是 X 类的

构造函数中就应包含数据成员初始化列表，用来给子对象进行初始化。X 类的构造函数的定义形式如下：

```
<X>::<X>(<参数表 0>)：<成员 1>(<参数表 1>),<成员 2>(<参数表 2>),…, <成员 n>(<参数表 n>)
{
⋮
}
```

冒号后由逗号隔开的项组成数据成员初始化列表，成员初始化列表是由一个或多个选项组成的，多个选项之间用逗号分隔。成员初始化列表中的选项可以是对子对象进行初始化的，也可以是对该类其他数据成员进行初始化的，其中的参数表给出为调用相应成员所在类的构造函数时应提供的参数。这些参数一般来自“参数表 0”，可以使用任意复杂的表达式，其中可以有函数的调用。如果某项的参数表为空，则表中相应的项可以省略。

对子对象的构造函数的调用顺序取决于这些子对象在类中说明的顺序，与它们在成员初始化列表中给出的顺序无关。

当建立 X 类的对象时，先调用子对象的构造函数，初始化子对象，然后才执行 X 类的构造函数，初始化 X 类中的其他成员。

析构函数的调用顺序与构造函数正好相反。

【例 3.10】 分析下面程序中构造函数与析构函数的调用顺序。

```
// 程序 Li3_10.cpp
// 子对象的初始化
#include<iostream>
using namespace std;
class Part
{
   public:
      Part();                          // Part 的无参构造函数
      Part (int x);                    // Part 的有参构造函数
      ~Part();                         // Part 的析构函数
   private:
      int val;
};
Part::Part()
{
   val = 0;
   cout<<"Default constructor of Part"<<endl;
}
Part::Part(int x)
{
   val = x;
   cout<<"Constructor of Part"<<","<<val<<endl;
}
Part::~Part()
{
```

```
    cout<<"Destructor of Part"<<","<<val<<endl;
}
class Whole
{
    public:
        Whole(int i);                    // Whole的有参构造函数
        Whole(){};                       // Whole的无参构造函数
        ~Whole();                        // Whole的析构函数
    private:
        Part p1;                         // 子对象
        Part p2;                         // 子对象
};
Whole::Whole(int i):p1(),p2(i)
{
    cout<<"Constructor of Whole"<<endl;
}
Whole::~Whole()
{
    cout<<"Destructor of Whole"<<endl;
}
int main()
{   Whole w(3);                          // 调用有参构造函数
    return 0;
}
```

程序输出结果为:

```
Default constructor of Part
Constructor of Part,3
Constructor of Whole
Destructor of Whole
Destructor of Part,3
Destructor of Part,0
```

程序分析:该程序的Whole类中出现了类Part的2个对象p1和p2作为该类的数据成员,则p1和p2被称为子对象。当建立的Whole类的对象w时,子对象p1和p2被建立,所指定的构造函数被执行。由于p1在Whole类中先说明,所以先执行它所使用的构造函数,即类Part的默认构造函数,接着p2执行它所使用的有参构造函数,当所有子对象被构造完之后,对象w的构造函数才被执行,从而得到前3行输出结果。后3行是执行相应析构函数的输出结果,可见,析构函数的调用顺序与构造函数的调用顺序正好相反。

Whole类的默认构造函数没有给出成员初始化列表,这表明子对象将使用默认构造函数进行初始化。例如:

```
int main()
{   Whole w; // 调用默认构造函数
    return 0;
```

```
}
```

程序输出结果为：

```
Default constructor of Part
Default constructor of Part
Destructor of Whole
Destructor of Part,0
Destructor of Part,0
```

注意：在这种情况下，Whole 类必须定义一个默认构造函数。

该例类 Whole 中数据成员只含有 2 个子对象，它的构造函数的成员初始化列表中含有 2 个对子对象进行初始化的选项。如果该类中还有其他数据成员，其初始化也可通过成员初始化列表进行。例如：

```
class Whole
{
   public:
      Whole(int i);
      Whole(){};
      ~Whole();
   private:
      Part p1;
      Part p2;
      int data;
};
```

为了初始化数据成员 data，这时该构造函数也可以定义成如下格式：

```
Whole::Whole(int i,j):p1(),p2(i),data(j)
{
   ⋮
}
```

3.5.2 堆对象

堆对象是在程序运行时根据需要随时可以被创建或删除的对象。在虚拟的程序空间中存在一些空闲存储单元，这些空闲存储单元组成所谓的堆，它使程序能够在运行时创建堆对象。当创建堆对象时，堆中的一个存储单元从未分配状态变为已分配状态，当删除所创建的堆对象时，这个存储单元从分配状态又变为未分配状态，这样，这个存储单元可供以后创建堆对象时使用。C++程序的内存格局通常分为 4 个区：

- 数据区(Data Area)；
- 代码区(Code Area)；
- 栈区(Stack Area)；
- 堆区(即自由存储区)(Heap Area)。

全局变量、静态数据、常量存放在数据区，所有类成员函数和非成员函数代码存放在代

码区,为运行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区,余下的空间都被作为堆区。

创建或删除堆对象分别使用如下两个运算符:new 和 delete。

使用 new 还可以为数组动态分配内存空间,这时需要在<类型说明符>后面缀上数组大小。释放动态分配的数组存储区时,可使用 delete[]。

1. 使用运算符 new 创建堆对象

使用 new 运算符可以动态地创建对象,即堆对象。其使用语法为:

```
new <类型说明符>(<初始值列表>)
```

运算符 new 的返回值是一个指针,使用该运算符给某对象分配一个地址值,由<类型说明符>给定对象的类型,该对象可以是类的对象,也可以是某种类型的变量。括号内的<初始值列表>将给出该对象的初始值。如果省略了<初始值列表>,则所创建的对象采用默认值。在使用该运算符创建对象时,系统自动调用该类的构造函数,并根据<初始值列表>中初始值的个数来选择对应参数个数的构造函数。

例如:

```
HeapObjectClass * pa;
pa = new HeapObjectClass(3,7);
```

这里,pa 是一个指向类 HeapObjectClass 的对象指针,运算符 new 创建一个类 HeapObjectClass 的对象,将它的地址值赋给 pa,并对该对象进行初始化,调用具有两个参数的构造函数,初始值为 3 和 7。

2. 使用运算符 delete 删除堆对象

该运算符是专门用来释放由运算符 new 所创建的对象的。其使用语法为:

```
delete <指针名>
```

其中,<指针名>必须是指向 new 所创建的堆对象,且必须是 new 所返回的值。当使用 new 创建堆对象时,构造函数被执行,以便初始化所创建的对象。同样,当使用 delete 删除堆对象时,析构函数被执行。

例如:

```
delete pa;
```

这里,pa 是一个指向 HeapObjectClass 类对象的指针,前面使用 new 给它分配了内存单元,这里使用运算符 delete 释放了 pa 指针。

因为堆是有限的,它可能变得拥挤。如果堆中没有足够的自由空间以满足内存的需要时,那么此需要失败,并且 new 返回一个空指针。因此,必须在使用 new 生成的指针之前进行检查。下面通过例子说明 new 和 delete 的用法。

【例 3.11】 分析下列程序的输出结果,注意运算符 new 和 delete 的用法。

```
// 程序 Li3_11.cpp
// new 和 delete 的用法
#include<iostream>
using namespace std;
```

```
class Heapclass
{
   public:
      Heapclass(int x);
      Heapclass();
      ~Heapclass();
   private:
      int i;
};
Heapclass::Heapclass(int x)
{
   i = x;
   cout<<"Constructor is called."<<i<<endl;
}
Heapclass::Heapclass()
{
   cout<<"Default Constructor is called."<<endl;
}
Heapclass::~Heapclass()
{
   cout<<"Destructor is called."<<endl;
}
int main()
{
   Heapclass * pa1, * pa2;
   pa1 = new Heapclass(4);          // 分配空间
   pa2 = new Heapclass;             // 分配空间
   if (!pa1 || !pa2)                // 检查空间
   {
      cout<<"Out of Memory!"<<endl;
      return 0;
   }
   cout<<"Exit main"<<endl;
   delete pa1;
   delete pa2;
   return 0;
}
```

程序输出结果为：

```
Constructor is called. 4
Default Constructor is called.
Exit main
Destructor is called.
Destructor is called.
```

程序分析：pa1，pa2 中是 2 个指向类 Heapclass 的对象指针，在能够赋给它们足够内存的情况下，使用运算符 new 给它们赋值，同时对它们所指向的对象进行初始化。该程序中

又使用了运算符 delete 释放了这两个指针所指向的对象,最后得到上面的输出结果。如果不能够赋给 pa1 或 pa2 足够内存,则输出"Out of Memory!"。

从程序的输出结果可以看出,使用运算符 new 时系统自动调用构造函数,根据初始值的不同,在创建对象时,调用了不同的构造函数。使用运算符 delete 时系统自动调用析构函数。

3. 使用运算符 new[]创建对象数组

使用运算符 new 还可以创建对象数组,其使用语法为:

```
new <类型说明符>[<算术表达式>]
```

其中,<算术表达式>给出数组的大小,后面不能再跟构造函数参数,所以,从堆上分配对象数组,只能调用默认的构造函数,不能调用其他任何构造函数。例如:

```
ObjectArrayClass * ptr;
ptr = new ObjectArrayClass[15];
```

其中,ObjectArrayClass 是一个已知的类名,ptr 是一个指向类 ObjectArrayClass 对象的指针;使用运算符 new[]创建一个对象数组,该数组有 15 个元素,每个元素都是 ObjectArrayClass 类的对象,ptr 是指向对象数组首元素的指针。

4. 使用运算符 delete []删除对象数组

使用运算符 delete 还可以释放由 new 创建的数组,其格式如下:

```
delete[] <指针名>
```

其中,<指针名>必须是指向 new[]所创建的对象数组,且必须是 new[]所返回的值。

delete[]是要告诉系统,该指针指向的是一个数组。如果在"[]"中填上了数组的长度信息,系统将忽略,并把它作为"[]"对待。但如果忘了写"[]",则程序将会产生运行错误。例如:

```
delete[] ptr;
```

这里,ptr 是一个指向 ObjectArrayClass 类对象的指针,前面使用 new 给它分配了内存单元,这里使用运算符 delete 释放了 ptr 指针。

注意:运算符 delete 必须用于由运算符 new 返回的指针;对一个指针只能使用一次运算符 delete;指针名前只能用一对方括号,而不管所释放数组的维数,并且在方括号内不能写任何东西;该运算符也适用于空指针。

下面考查使用 new[]和 delete[]创建和删除对象数组的示例程序。

【例 3.12】 分析下列程序的输出结果,注意运算符 new[]和 delete[]的用法。

```
// 程序 Li3_12.cpp
// 运算符 new[]和 delete[]的用法
#include<iostream>
using namespace std;
class Heapclass
{
   public:
      Heapclass();
      ~Heapclass();
```

```
    private:
        int i;
};
Heapclass::Heapclass()
{
    cout<<"Default Constructor is called."<<endl;
}
Heapclass::~Heapclass()
{
    cout<<"Destructor is called."<<endl;
}

int main()
{
    Heapclass * ptr;
    ptr = new Heapclass[2];          // 分配空间
    if (!ptr)// 检查空间
    {
        cout<<"Out of Memory!"<<endl;
        return 0;
    }
    cout<<"Exit main"<<endl;
    delete[]ptr;
    return 0;
}
```

程序输出结果为：

```
Default Constructor is called.
Default Constructor is called.
Exit main
Destructor is called.
Destructor is called.
```

程序分析：程序中先使用 new 创建了一个对象数组，接着又对该数组的元素进行赋值，这是使用动态分配内存单元获得对象数组的一种方法。程序最后使用 delete 释放了由 new 所创建的对象数组。从输出结果可清楚地看到，每个元素对象在被创建时，构造函数被调用，而每个元素对象在被删除时要调用析构函数。

注意：使用 new 建立对象数组时，由于只能调用默认的构造函数，所以不能为数组指定初始值。此时类中只能有一个无参或带默认参数的构造函数。

一般来说，堆空间相对其他内存空间比较空闲，随要随拿，给程序运行带来了较大的自由度。但是管理堆区是一件十分复杂的工作，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。使用堆空间往往由于：

- 直到运行时才能知道需要多少对象空间。
- 不知道对象的生存期到底有多长。
- 直到运行时才知道一个对象需要多少内存空间。

3.6 类的静态成员

每创建一个对象时,系统就为该对象分配一块内存单元来存放类中的所有数据成员。这样各个对象的数据成员可以分别存放、互不相干。但在某些应用中,需要程序中属于某个类的所有对象共享某个数据。虽然可以将所要共享的数据说明为全局变量,但这种解决办法将破坏数据的封装性。较好的解决办法是将所要共享的数据说明为类的静态成员。静态成员是指声明为 static 的类成员,包括静态数据成员和静态成员函数,在类的范围内所有对象共享该数据。

3.6.1 静态数据成员

静态数据成员不属于任何对象,它不因对象的建立而产生,也不因对象的析构而删除,它是类定义的一部分,所以使用静态数据成员不会破坏类的隐蔽性。类中的静态数据成员不同于一般的静态变量,也不同于其他类数据成员。它在程序开始运行时创建而不是在对象创建时创建。它所占空间的回收也不是在析构函数时进行而是在程序结束时进行。

1. 静态数据成员的初始化

必须对静态数据成员进行初始化,因为只有这时编译程序才会为静态数据成员分配一个具体的存储空间。

静态数据成员的初始化与一般数据成员不同,它的初始化不能在构造函数中进行。静态数据成员初始化的格式为:

＜数据类型＞＜类名＞::＜静态数据成员名＞=＜初始值＞;

这里的作用域运算符“::”用来说明静态数据成员所属类。

2. 静态数据成员的引用

静态数据成员可说明为公有的、私有的或保护的。若为公有的可直接访问,引用静态数据成员的格式为:

＜类名＞::＜静态数据成员＞

由于静态数据成员是类的数据成员,用对象名引用也可以。但通常使用上述格式来引用,因为静态数据成员不从属于任何一个具体对象。

【例 3.13】 统计点类的对象数,示例静态数据成员的计数作用。

```
// 程序 Li3_13.cpp
// 统计点类的对象数
#include<iostream>
using namespace std;
// 定义点类
class Point
{
   public:
      static int countP;              // 静态数据成员说明
      Point(int = 0,int = 0);
```

```
    ~Point();
  private:
    int X,Y;
};
Point::Point (int x,int y)
{
  X = x;
  Y = y;
  cout<<"Constructor is called!"<<endl;
  countP++;                          // 每创建一个对象,点数加 1
}
Point::~Point()
{
  cout<<"Destructor is called!"<<endl;
  countP--;                          // 每析构一个对象,点数减 1
  cout<<"现在对象数是:"<<countP<<endl;
}
// 静态数据成员定义和初始化
int Point::countP = 0;
// 主函数
int main()
{
  Point A(4,5);                      // 第 1 个对象
  cout<<"现在对象数是:"<<A.countP<<endl;
  Point B(7,8);                      // 第 2 个对象
  cout<<"现在对象数是:"<<Point::countP<<endl;
  return 0;
}
```

程序输出结果为：

```
Constructor is called!
现在对象数是:1
Constructor is called!
现在对象数是:2
Destructor is called!
现在对象数是:1
Destructor is called!
现在对象数是:0
```

程序分析：

(1) 在程序中,可用

```
cout<<"现在对象数是:"<<A.countP<<endl;
cout<<"现在对象数是:"<<Point::countP<<endl;
```

两种不同的格式来引用静态数据成员。想一想,是不是用后面格式更能说明问题,对象数应该针对类。

(2) 程序执行时,先创建对象 A,调用构造函数,静态数据成员 countP 的值加 1,由初值 0 变为 1。再创建对象 B,调用构造函数,静态数据成员 countP 的值加 1,变为 2。每析构一个对象,静态数据成员 countP 的值减 1。正如运行结果显示的那样。由此可见,静态数据成员 countP 是从属于整个类的。

3.6.2 静态成员函数

静态成员函数的定义和其他成员函数一样。静态成员函数与静态数据成员类似,从属于类,在一般函数定义前加上 static 关键字。调用静态成员函数的格式为:

<类名>::<静态成员函数名>(<参数表>);

或

<对象名>.<静态成员函数名>(<参数表>);

静态成员函数的主要作用是用来访问同类中的静态成员,维护对象之间共享的数据。

【例 3.14】 改写例 3.13,用静态成员函数输出点类的对象数。

```
// 程序 Li3_14.cpp
// 统计点类的对象数
#include<iostream>
using namespace std;
// 定义点类
class Point
{
   public:
      Point(int = 0,int = 0);
      ~Point();
      static void dispcount();
   private:
      int X,Y;
      static int countP;            // 静态数据成员说明
};
Point::Point (int x,int y)
{
   X = x;
   Y = y;
   cout<<"Constructor is called!"<<endl;
   countP ++ ;                      // 每创建一个对象,点数加 1
}
Point::~Point()
{
   cout<<"Destructor is called!"<<endl;
   countP--;                        // 每析构一个对象,点数减 1
   cout<<"现在对象数是:"<<Point::countP<<endl;
}
void Point::dispcount()
```

```
{
    cout<<"现在对象数是:"<<Point::countP<<endl;
}
// 静态数据成员定义和初始化
int Point::countP = 0;
// 主函数
int main()
{
    Point A(4,5);                    // 第 1 个对象
    Point::dispcount();
    Point B(7,8);                    // 第 2 个对象
    Point::dispcount();
    return 0;
}
```

程序输出结果与例 3.13 一样。由于静态成员函数没有 this 指针,它只能直接访问该类的静态数据成员、静态成员函数和类以外的函数和数据,访问类中的非静态数据成员必须通过参数传递方式得到对象名,然后通过对象名来访问,参看下面的例 3.15。但静态数据成员和静态成员函数可由任意访问权限许可的函数访问。

【例 3.15】 用静态成员函数输出点的位置。

```
// 程序 Li3_15.cpp
// 静态成员函数的使用
#include<iostream>
using namespace std;
class Point
{
    public:
        Point(int = 0,int = 0);
        static void displayxy(Point p);
    private:
        int X,Y;
};
Point::Point (int x,int y)
{
    X = x;
    Y = y;
}

void Point::displayxy(Point p)
{
    cout<<"("<<p.X<<","<<p.Y<<")"<<endl;        // 引用非静态数据成员
}
// 主函数
int main()
{
```

```
    Point A(4,5);                                    // 第1个对象
    cout<<"第1个点的位置是:";
    Point::displayxy(A);
    Point B(7,8);                                    // 第2个对象
    cout<<"第2个点的位置是:";
    Point::displayxy(B);
    return 0;
}
```

程序输出结果为:

```
第1个点的位置是:(4,5)
第2个点的位置是:(7,8)
```

程序分析:

(1) 主函数中的语句

```
Point::displayxy(A); Point::displayxy(B);
```

也可改为:

```
A.displayxy(A);B.displayxy(B);
```

(2) 由于displayxy()是静态成员函数,所以对非静态数据成员X、Y只能像下面的语句那样用对象名来引用。

```
cout<<"("<<p.X<<","<<p.Y<<")"<<endl;
```

3.7 类的友元

有时候,需要普通函数直接访问一个类的保护或私有数据成员。例如要求两点之间的距离、判断两个矩形的面积是否相等,这需要访问前面点类中的点的坐标X和Y、矩形类中的面积area。我们已从3.1.2节对类成员的访问控制的介绍中了解到友元是C++提供给外部的类或函数访问类的私有成员和保护成员的另一种途径,它提供在不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。友元可以是一个函数,称为友元函数,也可以是一个类,称为友元类。

3.7.1 友元函数

在类里声明一个普通函数,加上关键字friend,就成了该类的友元函数,它可以访问该类的一切成员。其原型为:

```
friend <类型><友元函数名>(<参数表>);
```

友元函数声明的位置可在类的任何地方,既可在公有区,也可在保护区,意义完全一样。友元函数的实现则在类的外部,一般与类的成员函数定义放在一起。使用方法如例3.16所示。

【例3.16】 用友元函数求两点的距离。

```
// 程序Li3_16.cpp
```

```
// 友元函数的使用
#include<iostream>
#include<cmath>
using namespace std;
class Point
{
   public:
      Point(double xi, double yi) {X = xi; Y = yi;}
      friend double length(Point &a,Point &b);
   private:
      int X, Y;
};

double length(Point &a, Point &b)
{
     double dx = a.X - b.X;
     double dy = a.Y - b.Y;
     return sqrt(dx * dx + dy * dy);
}

int main()
{
   Point p1(3, 5), p2(4, 6);
   double d = length(p1, p2);
   cout<<"The distance is"<<d<<endl;
   return 0;
}
```

程序输出结果为：

```
The distance is 1.41421
```

程序分析：

(1) 在该程序中，声明友元函数的语句

```
friend double length(Point &a,Point &b);
```

放在类中的公有部分，它也可以放在类中的其他部分，无论放在哪里，它都不是成员函数。

(2) 程序段

```
double length(Point &a, Point &b)
{
    double dx = a.X - b.X;
    double dy = a.Y - b.Y;
    return sqrt(dx * dx + dy * dy);
}
```

是友元函数的实现部分，它与普通函数的实现完全一样。

(3) 主程序中的

```
double d = length(p1, p2);
```


调用友元函数的方式也与普通函数的实现完全一样。

可见,友元函数是一个放在类中的普通函数。它可以像成员函数那样访问类中所有成员,又可以像普通函数那样使用。

为什么不将 length()直接设计为类的成员函数呢,这是因为如果这样设计就体现不出类与该函数的关系。如在类中设计一个成员函数求两点的距离,距离是点类的行为抽象吗?

在某些情况下,友元函数有很大的价值。如函数需要访问若干个类的私有或受保护数据才可完成某一任务。又如与其他程序设计语言(如汇编语言、C 语言等)混合编程时只能编写一个函数而不能用类。在第 6 章运算符重载中,我们还会看到它在重载某些运算符时也很有好处。

一个普通的函数可以定义成类的友元函数,一个类的成员函数也可以定义成另一个类的友元函数。友元成员函数的使用与一般友元函数的使用基本相同,只是通过相应的类或对象名来引用。

3.7.2 友元类

除了函数之外,一个类也可被声明为另一个类的友元,该类被称为友元类。假设有类 A 和类 B,若在类 B 的定义中将类 A 声明为友元,那么,类 A 被称作类 B 的友元类,它所有的成员函数都可以访问类 B 中的任意成员。友元类的声明格式为:

```
friend class<类名>;
```

下面例 3.17 中,将整个教师类 teacher 看成是学生类 student 的友元类,教师可以给学生设置学号,输入学生成绩。

【例 3.17】 示例友元类的使用。

```
// 程序 Li3_17.cpp
// 友元类的使用
#include<iostream>
#include<cmath>
using namespace std;
class student
{
   public:
       friend class teacher;                    // teacher 是 student 的友元类
       student(){};
   private:
       int number,score;                        // 学号,成绩
};
class teacher
{
   public:
        teacher(int i,int j);
        void display();
```

```
    private:
        student a;
};
teacher::teacher(int i,int j)
{
    a.number = i;
    a.score = j;
}
void teacher::display()
{
    cout<<"No = "<<a.number<<"";
    cout<<"score = "<<a.score<<endl;
}

int main()
{
    teacher t1(1001,89),t2(1002,78);
    cout<<"第 1 个学生的信息";
    t1.display();
    cout<<"第 2 个学生的信息";
    t2.display();
    return 0;
}
```

程序输出结果为：

```
第 1 个学生的信息 No = 1001 score = 89
第 2 个学生的信息 No = 1002 score = 78
```

程序分析：

(1) teacher 类是 student 类的友元类。teacher 类所有的成员函数都可以访问类中 student 类的任意成员。

(2) teacher 类的成员函数 display()引用了 student 类的两个私有成员 number 和 score。

(3) 程序的 teacher 类中，声明了一个子对象 a。通过子对象 a 引用 student 类的两个私有成员 number 和 score。

友元的作用主要是为了提高效率和方便编程，但友元破坏了类的整体性，也破坏了封装，使用时要权衡利弊。

3.8 应 用 实 例

用面向对象的方法重新编写一个学生成绩管理程序。要求能添加、编辑、查找、删除学生有关信息。

目的：区分面向过程与掌握面向对象的思想，掌握面向对象的思路及基本概念。

3.8.1 Student 类的定义

```
#include<iostream>
#include<string>
using namespace std;
class Student                                          // 类的定义
{
   int no;                                             // 学生的学号
   string name;                                        // 学生的姓名
   float score;                                        // 学生的成绩
   Student *per;                                       // 当前结点指针
   Student *next;                                      // 下一个结点指针
   public:
      Student();                                       // 构造函数
      Student *find(int i_no);                         // 查找指定学号的学生
      void edit(string i_newname,float i_score);       // 修改学生的信息
      void erase();                                    // 删除指定学号的学生
      int add(Student *i_newStudent);                  // 增加学生
      int getno();                                     // 获得学生的学号
      string getname();                                // 获得学生的名字
      float getscore();                                // 获得学生的成绩
      static int maxno;                                // 当前最大学号
};
```

3.8.2 Student 类中函数的实现

1. 构造函数

```
Student::Student()
{
    score = 0.0;
    per = NULL;
    next = NULL;
}
```

2. 查找指定学号的学生函数

```
Student *Student::find(int i_no)
{
    if(i_no == no)
       return this;
    if(next! = NULL)
       return next->find(i_no);
    return NULL;
}
```

3. 修改学生的名字函数

```
void Student::edit(string i_name,float i_score)
{
    if(i_name == "")
        return;
    name = i_name;
    score = i_score;
}
```

4. 删除指定学号的学生函数

```
void Student::erase()
{
    if(no<0)
        return;
    if(per! = NULL)
        per ->next = next;
    if(next! = NULL)
        next ->per = per;
    next = NULL;
    per = NULL;
}
```

5. 增加学生函数

```
int Student::add(Student * i_newStudent)
{
   int no = maxno + 1;
   while(true)
   {
       if(NULL == find(no))
           break;
       no = no + 1;
   }
   Student * tmp = this;
   while(true){
       if(tmp ->next == NULL)
           break;
       tmp = tmp ->next;
       }
   tmp ->next = i_newStudent;
   i_newStudent ->next = NULL;
   i_newStudent ->per = tmp;
   i_newStudent ->no = no;
   return no;
}
```

6. 得到相关信息函数

```
int Student::getno(){return no;}                    // 获得学生的学号
string Student::getname(){return name;}             // 获得学生的名字
float Student::getscore(){return score;}            // 获得学生的成绩
```

3.8.3 静态成员的初始化及程序的主函数

```
int Student::maxno = 1000;
int main()
{
   Student * studentroot = new Student();
   string input1;
   float input2;
   Student * tmp = NULL;
   while(true){
       cout<<"输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)"<<endl;
       cin>>input1;
       if(("F" == input1) || ("f" == input1))
       {
          cout<<"输入学号:";
          int id = - 1;
          cin>>id;
          tmp = studentroot ->find(id);
          if(tmp == NULL)
          {
             cout<<"没找到"<<endl;
             continue;
          }
          cout<<"学号:"<<tmp ->getno();
          cout<<" 姓名:";
          string name;
          if((name = tmp ->getname())! = "")
             cout<<name<<endl;
          else
             cout<<"未输入"<<endl;
             cout<<" 成绩:"<<tmp - >getscore()<<endl;
       }
       else if((input1 == "A") || (input1 == "a"))
       {
          cout<<"输入姓名,成绩：";
          cin>>input1>>input2;
          tmp = new Student();
          tmp ->edit(input1,input2);
          cout<<"学号:"<<studentroot ->add(tmp)<<endl;
       }
       else if((input1 == "E") || (input1 == "e"))
```

```
        {
            cout<<"输入学号:";
            int id=0;
            cin>>id;
            tmp=studentroot->find(id);
            if(tmp==NULL)
            {
            cout<<"空号"<<endl;
            continue;
            }
          cout<<"新姓名,新成绩:";
          cin>>input1>>input2;
          tmp->edit(input1,input2);
          cout<<"更改成功."<<endl;
      }
      else if((input1=="D")||(input1=="d"))
      {
          cout<<"输入学号:";
          int id=0;
          cin>>id;
          tmp=studentroot->find(id);
          tmp->erase();
          cout<<"已成功删除"<<endl;
          delete tmp;
      }
      else if((input1=="Q")||(input1=="q"))
      {
          break;
      }
      else
      {
          cout<<"输入有误!"<<endl;
        }
   }
   delete studentroot;
   return 0;
}
```

程序输出结果为：

```
输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)
a
输入姓名,成绩：马兰花 90
学号：1001
输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)
a
输入姓名,成绩：李君 80
学号：1002
```

```
输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)
f
输入学号：1002
学号：1002  姓名:李君  成绩:80
输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)
e
输入学号：1002
新姓名,新成绩：大山 70
更改成功.
输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)
d
输入学号：1001
已成功删除
输入指令：查找(F),增加(A),编辑(E),删除(D),退出(Q)
q
```

习　　题

一、填空题

(1) 类定义中关键字 private、public 和 protected 以后的成员的访问权限分别是_____、_____和_____。如果没有使用关键字,则所有成员默认定义为_____权限。具有_____访问权限的数据成员才能被不属于该类的函数所直接访问。

(2) 定义成员函数时,运算符"::"是_____运算符,"MyClass::"用于表明其后的成员函数是在"_____"中说明的。

(3) 在程序运行时,通过为对象分配内存来创建对象。在创建对象时,使用类作为_____,故称对象为类的_____。

(4) 假定 Dc 是一个类,则执行"Dc a[10],b(2)"语句时,系统自动调用该类构造函数的次数为_____。

(5) 对于任意一个类,析构函数的个数最多为_____个。

(6) _____运算符通常用于实现释放该类对象中指针成员所指向的动态存储空间的任务。

(7) C++程序的内存格局通常分为 4 个区：_____、_____、_____和_____。

(8) 数据定义为全局变量,破坏了数据的_____；较好的解决办法是将所要共享的数据定义为类的_____。

(9) 静态数据成员和静态成员函数可由_____函数访问。

(10) _____和_____统称为友元。

(11) 友元的正确使用能提高程序的_____,但破坏了类的封装性和数据的隐蔽性。

(12) 若需要把一个类 A 定义为一个类 B 的友元类,则应在类 B 的定义中加入一条语句：_____。

二、选择题(至少选一个,可以多选)

(1) 以下不属于类存取权限的是(　　)。

A. public　　B. static　　C. protected　　D. private

(2) 有关类的说法不正确的是(　　)。

A. 类是一种用户自定义的数据类型

B. 只有类的成员函数才能访问类的私有数据成员

C. 在类中,如不做权限说明,所有的数据成员都是公有的

D. 在类中,如不做权限说明,所有的数据成员都是私有的

(3) 在类定义的外部,可以被任意函数访问的成员有(　　)。

A. 所有类成员　　B. private 或 protected 的类成员

C. public 的类成员　　D. public 或 private 的类成员

(4) 关于类和对象的说法(　　)是错误的。

A. 对象是类的一个实例

B. 任何一个对象只能属于一个具体的类

C. 一个类只能有一个对象

D. 类与对象的关系和数据类型与变量的关系相似

(5) 设 MClass 是一个类,dd 是它的一个对象,pp 是指向 dd 的指针,cc 是 dd 的引用,则对成员的访问,对象 dd 可以通过(　　)进行,指针 pp 可以通过(　　)进行,引用 cc 可以通过(　　)进行。

A. ::　　B. .　　C. &　　D. ->

(6) 关于成员函数的说法中不正确的是(　　)。

A. 成员函数可以无返回值

B. 成员函数可以重载

C. 成员函数一定是内联函数

D. 成员函数可以设定参数的默认值

(7) 下面对构造函数的不正确描述是(　　)。

A. 系统可以提供默认的构造函数

B. 构造函数可以有参数,所以也可以有返回值

C. 构造函数可以重载

D. 构造函数可以设置默认参数

(8) 假定 A 是一个类,那么执行语句"A a,b(3),*p;"调用了(　　)次构造函数。

A. 1　　B. 2　　C. 3　　D. 4

(9) 下面对析构函数的正确描述是(　　)。

A. 系统可以提供默认的析构函数

B. 析构函数必须由用户定义

C. 析构函数没有参数

D. 析构函数可以设置默认参数

(10) 类的析构函数是(　　)时被调用的。

A. 类创建　　B. 创建对象　　C. 引用对象　　D. 释放对象

(11) 创建一个类的对象时,系统自动调用(　　);撤销对象时,系统自动调用(　　)。

A. 成员函数　　B. 构造函数　　C. 析构函数　　D. 拷贝构造函数

(12) 通常拷贝构造函数的参数是(　　)。

A. 某个对象名　　B. 某个对象的成员名

C. 某个对象的引用名　　D. 某个对象的指针名

(13) 关于 this 指针的说法正确的是(　　)。

A. this 指针必须显式说明

B. 当创建一个对象后,this 指针就指向该对象

C. 成员函数拥有 this 指针

D. 静态成员函数拥有 this 指针

(14) 下列关于子对象的描述中,(　　)是错误的。

A. 子对象是类的一种数据成员,它是另一个类的对象

B. 子对象可以是自身类的对象

C. 对子对象的初始化要包含在该类的构造函数中

D. 一个类中能含有多个子对象作其成员

(15) 对 new 运算符的下列描述中,(　　)是错误的。

A. 它可以动态创建对象和对象数组

B. 用它创建对象数组时必须指定初始值

C. 用它创建对象时要调用构造函数

D. 用它创建的对象数组可以使用运算符 delete 来一次释放

(16) 对 delete 运算符的下列描述中,(　　)是错误的。

A. 用它可以释放用 new 运算符创建的对象和对象数组

B. 用它释放一个对象时,它作用于一个 new 所返回的指针

C. 用它释放一个对象数组时,它作用的指针名前须加下标运算符[]

D. 用它可一次释放用 new 运算符创建的多个对象

(17) 关于静态数据成员,下面叙述不正确的是(　　)。

A. 使用静态数据成员,实际上是为了消除全局变量

B. 可以使用“对象名.静态成员”或者“类名::静态成员”来访问静态数据成员

C. 静态数据成员只能在静态成员函数中引用

D. 所有对象的静态数据成员占用同一内存单元

(18) 对静态数据成员的不正确描述是(　　)。

A. 静态成员不属于对象,是类的共享成员

B. 静态数据成员要在类外定义和初始化

C. 调用静态成员函数时要通过类或对象激活,所以静态成员函数拥有 this 指针

D. 只有静态成员函数可以操作静态数据成员

(19) 下面的选项中,静态成员函数不能直接访问的是(　　)。

A. 静态数据成员　　B. 静态成员函数

C. 类以外的函数和数据　　D. 非静态数据成员

(20) 在类的定义中,引入友元的原因是(　　)。

A. 提高效率　　B. 深化使用类的封装性

C. 提高程序的可读性　　D. 提高数据的隐蔽性

(21) 友元类的声明方法是(　　)。

A. friend class＜类名＞;　　B. youyuan class＜类名＞;

C. class friend＜类名＞;　　D. friends class＜类名＞;

(22) 下面对友元的错误描述是(　　)。

A. 关键字 friend 用于声明友元

B. 一个类中的成员函数可以是另一个类的友元

C. 友元函数访问对象的成员不受访问特性影响

D. 友元函数通过 this 指针访问对象成员

(23) 下面选项中,(　　)不是类的成员函数。

A. 构造函数　　B. 析构函数　　C. 友元函数　　D. 拷贝构造函数

三、简答题

(1) 类与对象有什么关系?

(2) 类定义的一般形式是什么? 其成员有哪几种访问权限?

(3) 类的实例化是指创建类的对象还是定义类?

(4) 什么是 this 指针? 它的主要作用是什么?

(5) 什么叫做拷贝构造函数? 拷贝构造函数何时被调用?

四、程序分析题(写出程序的输出结果,并分析结果)

(1)
```
#include<iostream>
using namespace std;
class Test
{
private:
   int num;
public:
   Test();
   Test(int n);
};
Test::Test()
{
   cout<<"Init defa"<<endl;
   num = 0;
}
Test::Test(int n)
{
   cout<<"Init"<<""<<n<<endl;
   num = n;
}
int main()
{
   Test x[2];
   Test y(15);
```

```
    return 0;
}
```

(2)
```
#include<iostream>
using namespace std;
class Xx
{
    private:
        int num;
    public:
        Xx(int x){num = x;}
        ~Xx(){cout<<"dst"<<num<<endl;}
};
int main()
{
    Xx w(5);
    cout<<"Exit main"<<endl;
    return 0;
}
```

(3) 将例3.10中的Whole类如下修改,其他部分不变,写出输出结果。

```
class Whole
{
    public:
        Whole(int i);    // Whole的有参构造函数
        Whole(){};       // Whole的无参构造函数
        ~Whole();        // Whole的析构函数
    private:
        Part p1;         // 子对象1
        Part p2;         // 子对象2
        Part p3;         // 子对象3
};
Whole::Whole(int i):p2(i),p1()
{
    cout<<"Constructor of Whole"<<endl;
}
Whole::~Whole()
{
    cout<<"Destructor of Whole"<<endl;
}
```

(4)
```
#include<iostream>
using namespace std;
class Book
{
    public:
        Book(int w);
```

```
    static int sumnum;
  private:
    int num;
};
Book::Book(int w)
{
  num = w;
  sumnum -= w;
}
int Book::sumnum = 120;
int main()
{
  Book b1(20);
  Book b2(70);
  cout<<Book::sumnum<<endl;
  return 0;
}
```

五、程序设计题

(1) 声明一个 Circle 类,有数据成员 radius(半径)、成员函数 area(),计算圆的面积,构造一个 Circle 的对象进行测试。

(2) 重新编写程序分析题(4)的程序,设计一个静态成员函数,用来输出程序分析题(4)中静态数据成员的值。