The C++ Programming Language

类和对象
构造函数和析构函数

陈辰
chenc@fudan.edu.cn
复旦大学软件学院

课程要点

- 类的构成
- 成员函数的声明
- 对象的定义和使用
- 构造函数与析构函数

1. 类的构成

从结构到类

结构是C的一种自定义的数据类型，它把相关联的数据元素组成一个单独的统一体。例如下面声明了一个日期结构：

```
struct Date{
    int year;
    int month;
    int day;
};
```

例2.1有关日期结构的例子

```
#include <iostream.h>
struct Date {
        int year;
        int month;
        int day;
};
int main()
{       Date date1;
        date1.year=2003;
   date1.month=8;
   date1.day=25;
        cout<<date1.year<<"."<<date1.month<<". “
                <<date1.day<<endl;
        return 0;
}
```

class与struct的比较

- 类是C++对C中结构的扩展。
- C语言中的struct是数据成员集合，而C++中的类，则是数据成员和成员函数的集合。
- struct是用户定义的数据类型，是一种构造数据类型。类和struct一样，也是一种用户定义的数据类型，是一种构造数据类型。
- C结构无法对数据进行保护和权限控制，所以结构中的数据是不安全的。C++中的类将数据和与之相关联的数据封装在一起，形成一个整体，具有良好的外部接口可以防止数据未经授权的访问，提供了模块间的独立性。

类的构成

类的成员分两部分：一部分对应数据的状态，称为数据成员，另一部分为作用于该数据状态的函数，称为成员函数。

类声明的一般格式如下：

```
class  类名 {
public:
   公有数据成员;
   公有成员函数;
protected:
   保护数据成员;
   保护成员函数;
private:
   私有数据成员;
   私有成员函数;
 };
```

3个关键字：private、protected和public,称为访问权限关键字。每个关键字下面又都可以有数据成员和成员函数。

- private部分称为类的私有部分，这一部分的数据成员和成员函数称为类的私有成员。私有成员只能由本类的成员函数访问，而类外部的任何访问都是非法的。
- public部分称为类的公有部分，这部分的数据成员和成员函数称为类的公有成员。公有成员可以由程序中的函数访问，它对外是完全开放的。
- protected部分称为类的保护部分，这部分的数据成员和成员函数称为类的保护成员。保护成员可以由本类的成员函数访问，也可以由本类的派生类的成员函数访问，而类外的任何访问都是非法的。

注 意：

(1)类声明格式中的3个部分并非一定要全有，但至少要有其中的一个部分。

一般一个类的数据成员应该声明为私有成员，成员函数声明为公有成员。

(2) 类声明中的private、protected和public三个关键字可以按任意顺序出现任意次。但是，如果把所有的私有成员、保护成员和公有成员归类放在一起，程序将更加清晰。

(3) private处于类体中第一部分时，关键字private可以省略。

(4) 数据成员可以是任何数据类型，但不能用自动(auto)、寄存器(register)或外部(extern)进行声明。

(5) 不能在类声明中给数据成员赋值。C++规定，只有在类对象定义之后才能给数据成员赋初值

用一个类来描述日期,其形式如下:

```
class Date {
    public:
     void setDate(int y,int m,int d);
     void showDate();
    private:
      int m_year;
      int m_month;
      int m_day;
};
```

2. 成员函数的声明

成员函数的声明通常采用以下两种方式:

(1) 普通成员函数形式

即在类的声明中只给出成员函数的原型，而成员函数体写在类的外部。

返回类型　成员函数（参数列表）；

类外定义的一般形式是:

```
返回类型　类名::成员函数名(参数表)
{
        // 函数体
}
```

例如，以下是表示坐标点的类Coord的声明

```
class  Coord {
    public:
        void setCoord (int,int); // 设置坐标点
        int  getx();             // 取x坐标点
        int  gety();             // 取y坐标点
    private:
        int x,y;
    };

void Coord::setCoord(int a,int b){ x=a; y=b;}
int Coord::getx(){  return  x;}
int Coord::gety(){  return y;}
```

内联函数和外联函数

- 类的成员函数可以分为内联函数和外联函数。内联函数是指那些定义在类体内的成员函数，即该函数的函数体放在类体内。而说明在类体内，定义在类体外的成员函数叫外联函数。外联函数的函数体在类的实现部分。
- 内联函数在调用时不是像一般函数那样要转去执行被调用函数的函数体，执行完成后再转回调用函数中，执行其后语句，而是在调用函数处用内联函数体的代码来替换，这样将会节省调用开销，提高运行速度。

内联函数和外联函数

- 内联函数与带参数的宏定义进行一下比较，它们的代码效率是一样的，但是内联函数要优于宏定义，因为内联函数遵循函数的类型和作用域规则，它与一般函数更相近，在一些编译器中，一旦关上内联扩展，将与一般函数一样进行调用，调试比较方便。
- 外联函数变成内联函数的方法很简单，只要在函数头前面加上关键字inline就可以了。

(2) 将成员函数以内联函数的形式进行说明。

有两种格式将成员函数声明为类的内联函数:

① 隐式声明　直接将函数声明在类内部。

例如:

```
class Coord{
  public:
       void setCoord(int a,int b)
       { x=a; y=b;}
       int getx()
       { return x;}
       int gety()
       { retrun y;}
 private:
       int x,y;
};
```

② 显式声明

在类声明中只给出成员函数的原型，而成员函数体写在类的外部。为了使它起内联函数的作用，在成员函数返回类型前冠以关键字“inline”。

这种成员函数在类外定义的一般形式是:

```
inline 返回类型   类名::成员函数名(参数表)
{
        // 函数体
}
```

例如上面的例子改为显式声明可变成如下形式:

```
class  Coord{
    public:
        void setCoord(int,int);
        int getx();
        int gety();
    private:
        int x,y;
    };
    inline void Coord::setCoord(int a,int b)
      { x=a; y=b;}
    inline int Coord::getx(){ return  x;}
    inline int Coord::gety(){ return y; }
```

说明

(1)使用inline说明内联函数时，必须使函数体和inline说明结合在一起，否则编译器将它作为普通函数处理。

例：inline void Coord:: setCoord(int ,int);

不能说明这是一个内联函数。

(2) 通常只有较短的成员函数才定义为内联函数，对较长的成员函数最好作为一般函数处理。

```
class Date{
public:
void showDate();
private:
  int year;
int month;
 int day;
};
inline  void Date::showDate()
{cout<<year<<'.';<<month<<'.'<<day<<endl;}
```

3. 对象的定义和使用

类与对象的关系

在C++中，可以把相同数据结构和相同操作集的对象看成属于同一类。 在C++中，类也是一种用户自定义数据类型，类的对象可以看成是该类类型的一个实例，定义一个对象和定义一个变量相似。类与对象间的关系，可以用数据类型int和整型变量i之间的关系类比。C++把类的变量叫做类的对象，对象也称为类的实例。

对象的定义

对象的定义，也称对象的创建

在C++中可以用以下两种方法定义对象：

(1) 在声明类的同时,直接定义对象。

例如:

```
class  Coord {
public:
    void setCoord(int,int);
    int getx();
    int gety();
private:
    int  x,y;
}   op1,op2;
```

对象的定义

（2）声明了类之后,在使用时再定义对象

例如:

```
class  Coord {
    //…
};
    // …
main()
{
  Coord  op1,op2;
    // …
}
```

说明：

- 在声明类的同时定义的对象是一种全局对象，在它的生存期内任何函数都可以使用它。
- 声明了一个类便声明了一种类型，它并不接收和存储具体的值，只作为生成具体对象的一种“样板”，只有定义了对象后，系统才为对象分配存储空间。

对象中成员的访问

当定义了一个类的对象后，就可以访问对象的成员了。在类的外部可以通过类的对象对公有成员进行访问，访问对象成员要使用操作符“.”。访问的一般形式是：

对象名.数据成员名

或

对象名.成员函数名(参数表)

其中“.”叫做对象选择符，简称点运算符。

例2.2 使用类Coord的完整程序

```
#include<iostream.h>
class Coord {
public:
    void setCoord(int a,int b)
        { x=a; y=b; }
   int getx()
        { return x; }
   int gety()
      { return y; }
private:
        int x,y;
     };
     void main()
     {
        Coord op1,op2;
        int i,j;
        op1.setCoord(5,6);          // 调用op1的setCoord()，初始化对象op1
        op2.setCoord(7,8);          // 调用op2的setCoord()，初始化对象op2
        i=op1.getx();               // 调用op1的getx()，取op1的x值
        j=op1.gety();               // 调用op1的gety()，取op1的y值
        cout<<"op1 i= "<<i<<"  op1 j= "<<j<<endl;
        i=op2.getx();               // 调用op2的getx()，取op2的x值
        j=op2.gety();               // 调用op2的gety()，取op2的y值
        cout<<"op2 i= "<<i<<"  op2 j= "<<j<<endl;
     }
```

对象中成员的访问

说明：

- 对象名.成员名实际上是一种缩写形式，它表达的意义是：

对象名.类名::成员名

```
void  main( )
    {
     Date  date1;
     date1.setDate(2006,9,21);
     //.....
    }
```

对象中成员的访问

- 在类的内部所有成员之间都可以通过成员函数直接访问，但是类的外部不能访问对象的私有成员。

【例2.3】一个存在错误的程序

```
#include <iostream.h>
class Date {
public:
      void setDate(int y,int m,int d);
      void showDate();
private:
   int year;
   int month;
   int day;
};
```

```
void Date ∷ setDate(int y,int m,int d)
{
        year=y;
        month=m;
        day=d;
}
inline void Date ∷ showDate()
{
   cout<<year<<"."<<month<<"."<<day<<endl;
}
void main()
{
        Date date1,date2;
        cout<<"Date1 set and output:"<<endl;
        date1.setDate(1998,4,28);
   cout<<date1.year<<"."<<date1.month<<"."<<date1.day<<endl;       //错误
        cout<<"Date2 set and output：  "<<endl;
        date2.setDate(2002,11,14);
        cout<<date2.year<<"."<<date2.month<<"."<<date2.day<<endl;     //错误
}
```

```
date1.showDate();
date2.showDate();
```

- 在定义对象时，若定义的是指向对象的指针，则访问此对象的成员时，要用“->”操作符。例如:

```
 void  main( )
{
 Date  *date3;
 date3->setDate(2001,8,15);
 //.....
}
```

类成员的访问属性

类成员有三种访问属性:公有（public）、私有(private)和保护（protected）。

- 说明为公有的成员不但可以被类中成员函数访问;还可在类的外部,通过类的对象进行访问。
- 说明为私有的成员只能被类中成员函数访问,不能在类的外部,通过类的对象进行访问。
- 说明为保护的成员除了类本身的成员函数可以访问外,该类的派生类的成员也可以访问,但不能在类的外部,通过类的对象进行访问。

类成员的访问属性

- 类的成员对类对象的可见性和对类的成员函数的可见性是不同的。
- 类的成员函数可以访问类的所有成员，而类的对象对类的成员的访问是受类成员的访问属性的制约的。
  - 成员函数的定义;
  - 对象的访问;

类中成员的访问方式

- 类中成员互访
  - 直接使用成员名
- 类外访问
  - 使用“对象名.成员名”方式访问 public 属性的成员

```
class Sample{
    public:
       int k;
       int geti(){return i;}
       int getj(){return j;}int getk(){return k;}
    private:
       int i;
    protected:
       int j;
    };
//.....
   Sample  a;  //定义类Sample的对象a
   a.i;  //非法，类Sample的对象a不能访问类的私有成员i
   a.j; //非法，类Sample的对象a不能访问类的保护成员j
   a.k; //合法，类Sample的对象a能访问类的公有成员k
    //.....
```

一般来说，公有成员是类的对外接口，而私有成员和保护成员是类的内部数据和内部实现，不希望外界访问。将类的成员划分为不同的访问级别有两个好处：一是信息隐蔽，即实现封装；二是数据保护，即将类的重要信息保护起来，以免其它程序不恰当地修改。

对象赋值语句

两个同类型的变量之间可以相互赋值。同类型的对象间也可以进行赋值，

当一个对象赋值给另一个对象时，所有的数据成员都会逐位拷贝。

例2.4

```
#include<iostream.h>
    class abc{
  public:
        void init(int i,int j) { a=i; b=j; }
        void show(){ cout<<a<<"  "<<b<<endl; }
  private:
        int a,b;
  };
  main()
  {     abc o1,o2;
        o1.init(12,34);
        o2=o1;          // 将对象o1数据成员的值赋给对象o2
        o1.show();
        o2.show();
        return 0;
  }
```

说明:

- 在使用对象赋值语句进行对象赋值时，两个对象的类型必须相同，如果对象的类型不同，编译时将出错。
- 两个对象之间的赋值，仅仅使这些对象中数据成员相同，而两个对象仍是分离的。
- 例2.5(1)的对象赋值是通过缺省的赋值运算符函数实现的。
- 当类中存在指针时，使用缺省的赋值运算符进行对象赋值，可能会产生错误。

类的作用域

所谓类的作用域就是指在类声明中的一对花括号所形成的作用域。

- 一个类的所有成员都在该类的作用域内,一个类的任何成员可以访问该类的其他成员。
- 一个类的成员函数可以不受限制地访问类的成员，而在类的外部，对该类的数据成员和成员函数的访问则要受到一定的限制,有时甚至是不允许的,这体现了类的封装功能。

例2.5 理解类的作用域

```
# include<iostream.h>
   class myclass{
   public:
      int  i;
      void  init(int);
      void  show(){ cout<< “i=” <<i<<endl;}   // 可以访问类中的
                                                //  数据成员i
    };
    void  myclass::init(int si){  i=si;}   // 可以访问类中的数据成员i
    int fun(){  return i; }                  // 非法,不能直接访问类中的i
    void  main()
    {    myclass ob;
         ob.init(5);                         // 给数据成员i赋初值5
         ob.show();
         i=8;              // 非法,不能直接访问类中的i,可改写成ob.i=8
         ob.show();
    }
```

4. 构造函数与析构函数

```
class Date {
   int day, month, year;
public:
    void InitDate(int d, int m, int y);  //初始化？？
    ...
};
```

程序员有的时候会忘记了调用初始化函数
或者调用了多次。这都是不好的现象。

4. 构造函数与析构函数

- 构造函数的作用

C++为类设计了构造函数(constructor)机制，它可以达到初始化数据成员的目的。

类的构造函数是类的一个特殊成员函数，它没有返回类型（void也不行），可以有参数，函数名和类名一样。

当创建类的一个新对象时，自动调用构造函数，完成初始化工作（需要注意构造函数是否有参数，以及参数的个数、类型）。

4. 构造函数与析构函数

对象的初始化

1） 数据成员是不能在声明类时初始化

2） 类型对象的初始化方法：

（1）调用对外接口（public成员函数）实现

声明类→定义对象→调用接口给成员赋值

（2）应用构造函数（constructor）实现

声明类→定义对象→同时给成员赋值

构造函数

1）特殊成员函数：不需要用户来调用它，而是在*建立对象时自动执行*。

2）两个特殊性：*（1）必须与类同名；（2）且不返回任何值*，也不需要指定void型（与一般函数有类型声明不同！）。

4.1 构造函数

构造函数是一种特殊的成员函数,它主要用于为对象分配空间,进行初始化。构造函数具有一些特殊的性质:

(1) 构造函数的名字必须与类名相同。

(2) 构造函数可以有任意类型的参数,但不能指定返回类型。它有隐含的返回值,该值由系统内部使用。

(3) 构造函数是特殊的成员函数,函数体可写在类体内,也可写在类体外。

(4) 构造函数可以重载,即一个类中可以定义多个参数个数或参数类型不同的构造函数。构造函数是不能继承

(5) 构造函数被声明为公有函数,但它不能像其他成员函数那样被显式地调用,它是在定义对象的同时被调用的。

- 在声明类时如果没有定义类的构造函数，编译系统就会在编译时自动生成一个默认形式的构造函数，
- 默认构造函数是构造对象时不提供参数的构造函数。
- 除了无参数构造函数是默认构造函数外，带有全部默认参数值的构造函数也是默认构造函数。
- 自动调用：构造函数在定义类对象时自动调用，不需用户调用，也不能被用户调用。在对象使用前调用。
- 调用顺序：在对象进入其作用域时（对象使用前）调用构造函数。

- 定义构造函数的一般形式为:

```
class 类名
{ public:
   类名（形参表）;          //构造函数的原型
   //类的其它成员
};
类名::类名（形参表）        //构造函数的实现
{
   //函数体
}
```

- 构造函数有两种方式初始化数据成员:

1）在构造函数体内用赋值语句的方式;

```
Date::Date(int y, int m, int d)
{  year = y;
   month = m;
   day = d;
}
```

2）构造函数的初始化列表的方式

```
Date::Date(int y,int m,int d):year(y), month(m),day(d)
{}
```

通常,利用构造函数创建对象有以下两种方法:

(1) 利用构造函数直接创建对象.其一般形式为:

类名 对象名[(实参表)];

这里的“类名”与构造函数名相同,“实参表”是为构造函数提供的实际参数。

例2.6 为类Date建立一个构造函数

```
#include <iostream.h>
class Date {
    public:
          Date(int y,int m,int d);     // 构造函数
          void setDate(int y,int m,int d);
          void showDate();
    private:
        int year, month, day;
    };
Date::Date(int y,int m,int d)    // 构造函数的实现
{  year=y; month=m; day=d; }
void Date::setDate(int y,int m,int d)
{ year=y; month=m; day=d; }
inline void Date::showDate()
{  cout<<year<<"."<<month<<"."<<day<<endl; }
```

例2.7 利用构造函数直接创建对象

```
#include <iostream.h>
class Date {
        //  省略
};
        //  省略
void main()
{
    Date date1(1998,4,28);    // 定义类Date的对象date1,
                              // 自动调用date1的构造函数,初始化对象date1
    cout<<"Date1 output1:"<<endl;
    date1.showDate();   // 调用date1的showDate(),显示date1的数据
    date1.SetDate(2002,11,14);   // 调用date1的setDate(),
                                 // 重新设置date1的数据
    cout<<"Date1 output2:"<<endl;
    date1.showDate();    // 调用date1的showDate(),显示date1的数据
}
```

```
constructing...
Date1 output1:
1998.4.28
Date1 output2:
2002.11.14
```

(2) 利用构造函数创建对象时,通过指针和new来实现。
其一般语法形式为:

类名 *指针变量 = new 类名[(实参表)];

例如:

Date *date1=new Date(1998,4,28);

就创建了对象(*date1)。

将例2.7的主函数改成用这种方法来实现,其运行结果与原例题完全相同。

```
void main()
{
        Date *date1;
        date1=new Date(1998,4,28);
              //可合写成:Date *date1=new Date(1998,4,28);
        cout<<"Date1 output1:"<<endl;
        date1->showDate();
        date1->setDate(2002,11,14);
        cout<<"Date1 output2:"<<endl;
        date1->showDate();
        delete date1;
}
```

- **说明：**
- 构造函数的名字必须与类名相同，否则编译器将把它当作一般的成员函数来处理。
- 构造函数是不能说明它的返回值类型的，甚至说明为void类型也不行。
- 构造函数可以是不带参数的。

构造函数可以是不带参数

```
class  A{
public:
     A();
     //...
private:
     int x;
};
A::A()
{
     cout<<"initialized \n";
     x=50;
}
```

在main函数中定义对象的方法：A a;

```
main()
{
A a;
…
}
```

例 有两个长方柱，其长、宽、高分别为：(1)12,25,30; (2)15,30,21。求它们的体积。要求：编一个基于对象的程序，在类中用带参数的构造函数。

```
class Box
{
public:
    Box(int,int,int);
    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;
}

int main( )
{
    Box box1(12,25,30);
    cout << box1.volume( ) << endl;
    Box box2(15,30,21);
    cout << box2.volume( ) << endl;
  return 0;
}
```

4.2 成员初始化表

对于常量类型和引用类型的数据成员,不能在构造函数中用赋值语句直接赋值,C++提供初始化表进行置初值。

带有成员初始化表的构造函数的一般形式如下:

```
类名::构造函数名([参数表])[:(成员初始化表)]
{
    // 构造函数体
}
```

成员初始化表的一般形式为:

数据成员名1(初始值1),数据成员名2(初始值2),……

例2.8 成员初始化表的使用

```
#include<iostream.h>
class A{
public:
A(int x1):x(x1),rx(x),pi(3.14)  // rx(x)相当于rx=x ,
{   }                                  // pi(3.14)相当于pi=3.14
void print()
{  cout<<"x="<<x<<"  "<<"rx="<<rx<<"  "<<"pi="<<pi; }
    private:
   int x; int& rx; const float pi;
};
main()
{   A a(10);
     a.print();
     return 0;
}
```

- 构造函数采用成员初始化表对数据成员进行初始化，是一些程序员喜欢使用的方法。

```
class B{
    int  i;
    char j;
    float  f;
public:
    B(int I,  char J,  float F)
    {  i=I; j=J; f=F; };
};
```

```
class  B{
public：
  B(int I,char J,float F):i(I),j(J),f(F)
    {  }
private:
    int  i;
    char j;
    float  f;
};
```

说明

- 如果需要将数据成员存放在堆中或数组中，则应在构造函数中使用赋值语句，即使构造函数有成员初始化表也应如此。

```
class  C{
public:
    C(int I,char Ch,float F,char N[]):i(I),ch(Ch),f(F)
    { strcpy (name,N);}
private:
    int i;
    char ch;
    float f;
    char name[25];
};
```

类成员是按照它们在类里被声明的顺序初始化的,与它们在初始化表中列出的顺序无关。

【例2.9】

```
#include<iostream.h>
class D {
public:
        D(int i):mem2(i),mem1(mem2+1)
        {
            cout<<"mem1: "<<mem1<<endl;
            cout<<"mem2: "<<mem2<<endl;
    }
private:
        int mem1;
        int mem2;
};
void main()
{
        D d(15);
}
```

```
mem1: -858993459
mem2: 15
```

缺省参数的构造函数

例2.10

```
#include<iostream.h>
class Coord {
public:
    Coord(int a=0,int b=0){ x=a; y=b;}   // 带有缺省参数的构造函数
    int getx(){ return x; }
     int gety(){ return y; }
  private:  int x,y;
};
  void main()
  {   Coord op1(5,6); Coord op2(5); Coord op3;
      int i,j;
      i=op1.getx();j=op1.gety();
      cout<<"op1 i= "<<i<<" op1 j= "<<j<<endl;
      i=op2.getx();j=op2.gety();
      cout<<"op2 i= "<<i<<" op2 j= "<<j<<endl;
       i=op3.getx();j=op3.gety();
      cout<<"op3 i= "<<i<<" op3 j= "<<j<<endl;  }
```

缺省参数的构造函数

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

构造函数的重载

1) 构造函数重载：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）

2）关于构造函数重载的说明：

(1) 默认构造函数：一个调用构造函数时不必给出实参的构造函数。显然，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，建立对象时只执行其中一个构造函数，并非每个构造函数都被执行。

```
class Box
{
public:
    Box(int h, int w, int l): height(h),width(w),length(l)  { }
    Box();
    int volume( );
private:
    int height;
    int width;
    int length;
};
```

```
Box::Box()
{
    height = 10;
    width = 10;
    lenght = 10;
}
int Box::volume( )
{
    return height*width*length;
}
```

```
int main( )
{
    Box box1;  // 书上为 box1();
    cout << box1.volume( ) << endl;

    Box box2(15,30,25);
    cout << box2.volume( ) << endl;
    return 0;
}
```

重载构造函数

例2.11 重载构造函数应用例程。

```
#include <iostream.h>
class Date
{
    public:
        Date();                  // 无参数的构造函数
      Date(int y,int m,int d); // 带有参数的构造函数
      void showDate();
    private:
        int year, month, day;
};
Date::Date()
{ year=2019;  month=4; day=28; }
Date::Date( int y, int m, int d)
{   year=y;  month=m;  day=d; }
inline void Date::showDate()
{  cout<<year<<"."<<month<<"."<<day<<endl; }
```

```
void main()
{
    Date date1;                          // 声明类Date的对象date1，
                                         // 调用无参数的构造函数
    cout<<"Date1 output："<<endl;
    date1.showDate();     // 调用date1的showDate()，显示date1的数据
    Date date2(2020，3，14);  // 定义类Date的对象date2，
                                            // 调用带参数的构造函数
    cout<<"Date2 output："<<endl;
    date2.showDate();     // 调用date2的showDate()，显示date2的数据
}
```

运行结果：

```
Date1 output：
2019.4.28
Date2 output：
2020.3.14
```

例2.12关于计时器的例子

```
#include<iostream.h>
    #include<stdlib.h>
    class timer{
    public:
        timer()                                  // 无参数构造函数,给seconds清0
   { seconds=0; }
        timer(char* t)                           // 含一个数字串参数的构造函数
        { seconds=atoi(t); }
        timer(int t)                             // 含一个整型参数的构造函数
        { seconds=t; }
        timer(int min,int sec)       // 含两个整型参数的构造函数
        { seconds=min*60+sec; }
        int gettime()
        { return seconds; }
private:
   int seconds;
};
main()
{
        timer a,b(10),c("20"),d(1,10);
        cout<<"seconds1="<<a.gettime()<<endl;
        cout<<"seconds2="<<b.gettime()<<endl;
        cout<<"seconds3="<<c.gettime()<<endl;
        cout<<"seconds4="<<d.gettime()<<endl;
        return 0;
}
```

```
class  x {
public:
    x();                    // 没有参数的构造函数
    x(int i=0);             // 带缺省参数的构造函数
};
    //…
void main()
{
    x one(10);              // 正确,调用x(int i=0)
    x two;                  // 存在二义性
    //…
}
```

拷贝构造函数

拷贝构造函数是一种特殊的构造函数,其形参是本类对象的引用。其作用是使用一个已经存在的对象去初始化另一个同类的对象。

- 通过等于号复制对象时，系统会自动调用拷贝构造函数。
- 拷贝构造函数与原来的构造函数实现了函数的重载。

拷贝构造函数具有以下特点：

(1) 因为该函数也是一种构造函数，所以其函数名与类名相同，并且该函数也没有返回值类型。

(2) 该函数只有一个参数，并且是同类对象的引用。

(3) 每个类都必须有一个拷贝构造函数。程序员可以根据需要定义特定的拷贝构造函数，以实现同类对象之间数据成员的传递。如果程序员没有定义类的拷贝构造函数，系统就会自动生成产生一个缺省的拷贝构造函数。

缺省的拷贝构造函数，也将拷贝对象的各个数据成员拷贝给被拷贝对象的各个数据成员。

这样一来，两个对象的内存映像是一模一样的。

1. 自定义拷贝构造函数

自定义拷贝构造函数的一般形式如下:

```
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    }
```

用户自定义拷贝构造函数

```
class  Coord{
    int x,y;
public:
    Coord(int a，  int b)          // 构造函数
    {
        x=a;
        y=b;
        cout<<"Using normal constructor\n";
    }
    Coord(const Coord& p)                  // 拷贝构造函数
    {
        x=2*p.x;
        y=2*p.y;
        cout<<"Using copy constructor\n";
    }
        //…
};
```

如果p1、p2为类Coord的两个对象，p1已经存在，则coord p2(p1)调用拷贝构造函数来初始化p2

例2.13 自定义拷贝构造函数的使用

```
#include<iostream.h>
class Coord        {
public:
    Coord(int a,int b)              // 构造函数
    { x=a; y=b; cout<<"Using normal constructor\n";}
    Coord(const Coord& p)     // 拷贝构造函数
    { x=2*p.x;  y=2*p.y; cout<<"Using copy constructor\n";}
    void print(){  cout<<x<<"  "<<y<<endl; }
private:
    int x,y;  };
main()
{   Coord p1(30,40);   // 定义对象p1,调用了普通的构造函数
    Coord p2(p1);        // 以“代入”法调用拷贝构造函数,
                           用对象p1初始化对象p2
    p1.print();  p2.print();
    return 0;
}
```

- 除了用“代入法”调用拷贝构造函数外，还可以采用“赋值”法调用拷贝构造函数，如：

```
main()
{
    Coord  p1(30,40);
    Coord  p2=p1; //以"赋值"法调用拷贝构造函数，用对象p1初始化对象p2
    //…
}
```

2. 缺省的拷贝构造函数

如果没有编写自定义的拷贝构造函数,C++会自动地将一个已存在的对象复制给新对象,这种按成员逐一复制的过程由是缺省拷贝构造函数自动完成的。

例2.14 调用缺省的拷贝构造函数

```
#include<iostream.h>
class Coord{
public:
    Coord(int a,int b)
    {  x=a; y=b; cout<<"Using normal constructor\n"; }
    void print(){ cout<<x<<"  "<<y<<endl;}
private:
    int x,y;  };
main()
{   Coord p1(30,40);    // 定义类Coord的对象p1,
                          // 调用了普通构造函数初始化对象p1
     Coord p2(p1);       // 以“代入”法调用缺省的拷贝构造函数,
                          // 用对象p1初始化对象p2
    Coord p3=p1;         // 以“赋值”法调用缺省的拷贝构造函数,
                          // 用对象p1初始化对象p3
    p1.print(); p2.print(); p3.print();
    return 0;
}
```

3.调用拷贝构造函数的三种情况

(1)　当用类的一个对象去初始化该类的另一个对象时。

如例2.20主函数main()中的下述语句:

```
Coord p2(p1);    // 用对象p1初始化对象p2,
                                拷贝构造函数被调用(代入法)
Coord p3=p1;    // 用对象p1初始化对象p3,
                                拷贝构造函数被调用(赋值法)
```

这时需要调用拷贝构造函数。

(2) 当函数的形参是类的对象,调用函数,进行形参和实参结合时。

例如:

```
        //…
fun1(Coord p)      // 函数的形参是类的对象
{
   p.print();
}
main()
{
  Coord p1(10,20);
  fun1(p1);          // 当调用函数,进行形参和实参结合时,
                        调用拷贝构造函数
  return 0;
}
```

(3) 当函数的返回值是对象,函数执行完成,返回调用者时。例如:

```
// …
Coord fun2()
{    Coord p1(10,30);
    return p1;    }    // 函数的返回值是对象

main()
{    Coord p2;
    P2=fun2(); // 函数执行完成,返回调用者时,调用拷贝构造函数
    return 0;  }
```

浅拷贝和深拷贝

所谓浅拷贝，就是由缺省的拷贝构造函数所实现的数据成员逐一赋值，若类中含有指针类型数据，则会产生错误。

为了解决浅拷贝出现的错误，必须显示地定义一个自己的拷贝构造函数，使之不但拷贝数据成员，而且为对象1和对象2分配各自的内存空间，这就是所谓的深拷贝。

例2.15 浅拷贝例子

```
#include<iostream.h>
#include<string.h>
class Student {
public:
    Student(char *name1,float score1);
        ~Student();
private:
    char *name;                          // 学生姓名
        float score;                     // 学生成绩
};
Student :: Student(char *name1,float score1)
{
    cout<<"Constructing..."<<name1<<endl;
        name=new char[strlen(name1)+1];
        if (name !=0)
    {
        strcpy(name,name1);
        score=score1;
    }
}
```

```
Student :: ~Student()
{
        cout<<"Destructing..."<<name<<endl;
        name[0]='\0';
        delete name;
}
void main()
{
    Student stu1("liming",90); // 定义类Student的对象stu1
    Student stu2=stu1;          // 调用缺省的拷贝构造函数
}
```

- Constructing... liming
- Destructing... liming
- Destructing...

例2.15 深拷贝例子

```
#include<iostream.h>
#include<string.h>
class Student {
private:
    char *name;              // 学生姓名
        float score;         // 学生成绩
public:
    Student(char *name1,float score1);
    Student(Student& stu);
        ~Student();
};
Student :: Student(char *name1,float score1)
{
    cout<<"constructing..."<<name1<<endl;
        name=new char[strlen(name1)+1];
        if (name !=0)
    {
        strcpy(name,name1);
            score=score1;
    }
}
```

```
Constructing…liming
Copy constructing…liming
Destructing…liming
Destructing…liming
```

```
Student :: Student(Student& stu)
{
    cout<<"Copy constructing..."<<stu.name<<endl;
        name=new char[strlen(stu.name)+1];
        if (name !=0)
    {
        strcpy(name,stu.name);
        score=stu.score;
    }
}
Student :: ~Student()
{
        cout<<"Destructing..."<<name<<endl;
    name[0]='\0';
        delete name;
}
void main()
{
    Student stu1("liming",  90);      // 定义类Student的对象stu1,
    Student stu2=stu1;                              // 调用自定义的拷贝构造函数
}
```

4.4 析构函数

析构函数也是一种特殊的成员函数。它执行与构造函数相反的操作,通常用于撤消对象时的一些清理任务,如释放分配给对象的内存空间等。

析构函数有以下一些特点:

① 析构函数与构造函数名字相同,但它前面必须加一个波浪号(~);

② 析构函数没有参数,也没有返回值,而且不能重载。因此在一个类中只能有一个析构函数;

③ 当撤消对象时,编译系统会自动地调用析构函数。如果程序员没有定义析构函数，系统将自动生成和调用一个默认析构函数，默认析构函数只能释放对象的数据成员所占用的空间，但不包括堆内存空间。

例2.16 重新说明类Date

```
#include <iostream.h>
class Date
{ public:
      Date(int y,int m,int d);   // 构造函数
       ~Date();                          // 析构函数
      void setDate(int y,int m,int d);
      void showDate();
   private:
       int year, month, day;
};
Date::Date(int y,int m,int d) // 构造函数的实现
{ cout<<"constructing..."<<endl;  year=y; month=m; day=d; }
```

```
Date::~Date()                                   //  析构函数的实现
{   cout<<"destruting..."<<endl;     }
void Date::setDate(int y,int m,int d)
{   year=y;month=m;day=d; }
inline void Date::showDate()
{ cout<<year<<"."<<month<<"."<<day<<endl; }
void main()
{
    Date date1(1998,4,28);  // 定义类Date的对象date1,
                            // 调用date1的构造函数,初始化对象date1
    cout<<"Date1 output1:"<<endl;
    date1.showDate(); // 调用date1的showDate(),显示date1的数据
    date1.setDate(2002,11,14);  // 调用date1的setDate(),
                                // 重新设置date1的数据
    cout<<"Date1 output2:"<<endl;
    date1.showDate(); // 调用date1的showDate(),显示date1的数据
}
```

析构函数被调用的两种情况

1）若一个对象被定义在一个函数体内，当这个函数结束时，析构函数被自动调用。

2）若一个对象是使用new运算符动态创建，在使用delete释放时，自动调用析构函数。

【例2.17】较完整的学生类例子

```
#include<iostream.h>
#include<string.h>
class Student {
public:
    Student(char *name1,char *stu_no1,float score1);       // 构造函数
    ~Student();                                            // 析构函数
    void modify(float score1);                             // 修改数据
    void show();                                           // 显示数据
private:
    char *name;                                            // 学生姓名
    char *stu_no;          // 学生学号
    float score;           // 学生成绩
};
```

```
Student :: Student(char *name1,char *stu_no1,float score1)
{   name=new char[strlen(name1)+1];
    strcpy(name,name1);
    stu_no=new char[strlen(stu_no1)+1];
    strcpy(stu_no,stu_no1);
    score=score1;

Student :: ~Student()
{       delete []name;
        delete []stu_no;    }
void Student :: modify(float score1)
{      score=score1;   }
void Student :: show()
{       cout<<"\n name: "<<name;
        cout<<"\n stu_no: "<<stu_no;
        cout<<"\n score: "<<score;   }
```

```
void main()
{
    Student stu1("Liming","990201",90); // 定义类Student的对象stu1,
                                        // 调用stu1的构造函数,初始化对象stu1
        stu1.show();            // 调用stu1的show(),显示stu1的数据
        stu1.modify(88);        // 调用stu1的modify(),修改stu1的数据
        stu1.show();            // 调用stu1的show(),显示stu1修改后的数据
}
name:Liming
stu_no:990201
score:90
name:Liming
stu_no:990201
score:88
```

缺省的析构函数

每个类必须有一个析构函数。若没有显式地为一个类定义析构函数,编译系统会自动地生成一个缺省的析构函数,其格式如下:

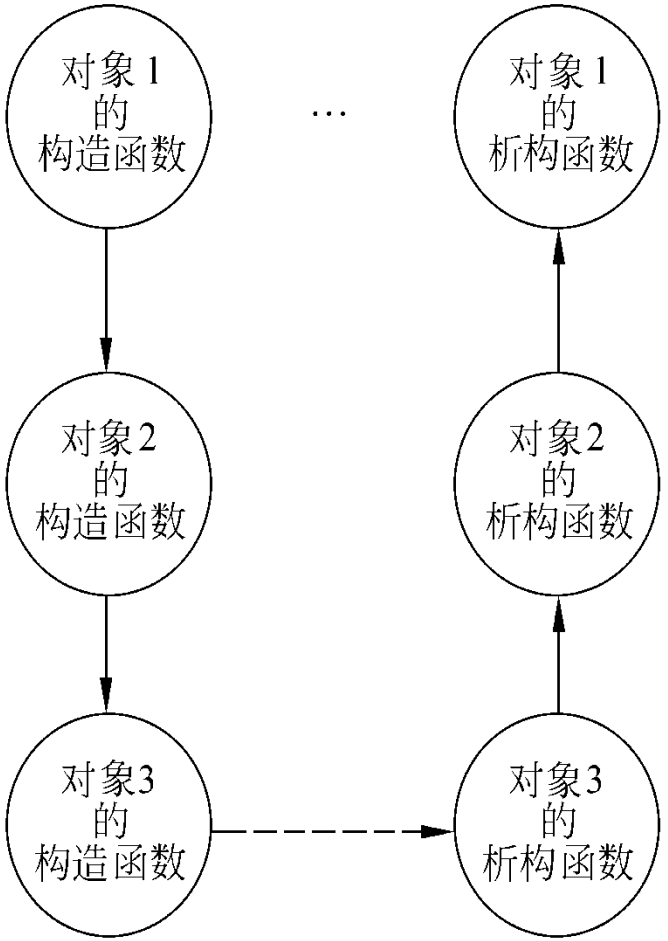
类名::析构函数名( )

{ }

```
class  string_data {
public:
    string_data(char *)
    { str=new char[max_len];}
    ~string_data()
    { delete []str;}
    void  get_info(char *);
    void  sent_info(char *);
private:
    char  *str;
    int  max_len;
};
```

调用构造函数和析构函数的顺序

1) 一般顺序：调用
的次序正好与调
数的次序相反：
用的构造函数，
（同一对象中的
数最后被调用，
调用的构造函数
的析构函数最先
如图示意



2) 全局对象：在全局范围中定义的对象(即在所有函数之外定义的对象)，它的构造函数在所有函数(包括main函数)执行之前调用。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。

3) auto局部对象：局部自动对象(例如在函数中定义的对象)，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。

4) static局部对象：如果在函数中定义静态局部对象，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。

对象的生存期

- 对象按生存期的不同分为如下几种:
- (1)局部对象: 当对象被定义时，调用构造函数，该对象被创建；当程序退出该对象所在的函数体或程序块时，调用析构函数，对象被释放。
- (2)全局对象: 当程序开始运行时，调用构造函数，该对象被创建；当程序结束时，调用析构函数，该对象被释放。
- (3)静态对象: 当程序中定义静态对象时，调用构造函数，该对象被创建；当整个程序结束时，调用析构函数，对象被释放。
- (4)动态对象: 执行new运算符调用构造函数，动态对象被创建；用delete释放对象时，调用析构函数。

- (1)局部对象是被定义在一个函数体或程序块内的，它的作用域限定在函数体或程序块内，生存期较短。
- (2)静态对象是被定义在一个文件中，它的作用域从定义是起到文件结束时为止。生存期较长。
- (3)全局对象是被定义在某个文件中，它的作用域包含在该文件的整个程序中，生存期是最长的。
- (4)动态对象是由程序员掌握的，它的作用域和生存期是由new和delete之间的间隔决定的。

浅拷贝与深拷贝

- 浅拷贝
  - 实现对象间数据元素的一一对应复制。
- 深拷贝
  - 当被复制的对象数据成员是指针类型时，不是复制该指针成员本身，而是将指针所指的对象进行复制。

例 对象的浅拷贝

```
#include <iostream>
using namespace std;
class Point{
public:
    Point():x(0),y(0){
        cout<<"Default Constructor called."<<endl;
    }
    Point(int x, int y):x(x),y(y){
        cout<< "Constructor called."<<endl;
    }
    ~Point(){cout<<"Destructor called."<<endl;}
    int getX() const { return x;}
    int getY() const { return y;}
    void move(int newX, int newY) {
        x = newX;
        y = newY;
    }
private:
    int x, y;
};
```

例 对象的浅拷贝

```
class ArrayOfPoints {                    //动态数组类
public:
    ArrayOfPoints(int size) : size(size) {
        points = new Point[size];
    }
    ~ArrayOfPoints() {
        cout << "Deleting..." << endl;
        delete[] points;
    }
    Point &element(int index) {
        assert(index >= 0 && index < size);
        return points[index];
    }
private:
    Point *points;          //指向动态数组首地址
    int size;               //数组大小
};
```

```
int main() {
    int count;
    cout << "Please enter the count of points: ";
    cin >> count;
    ArrayOfPoints pointsArray1(count); //创建数组对象
    pointsArray1.element(0).move(5,10);
    pointsArray1.element(1).move(15,20);

    ArrayOfPoints pointsArray2 = pointsArray1; //创建副
    本
    cout << "Copy of pointsArray1:" << endl;
    cout << "Point_0 of array2: " <<
    pointsArray2.element(0).getX() << ", "
         << pointsArray2.element(0).getY() << endl;
    cout << "Point_1 of array2: " <<
    pointsArray2.element(1).getX() << ", "
         << pointsArray2.element(1).getY() << endl;
```

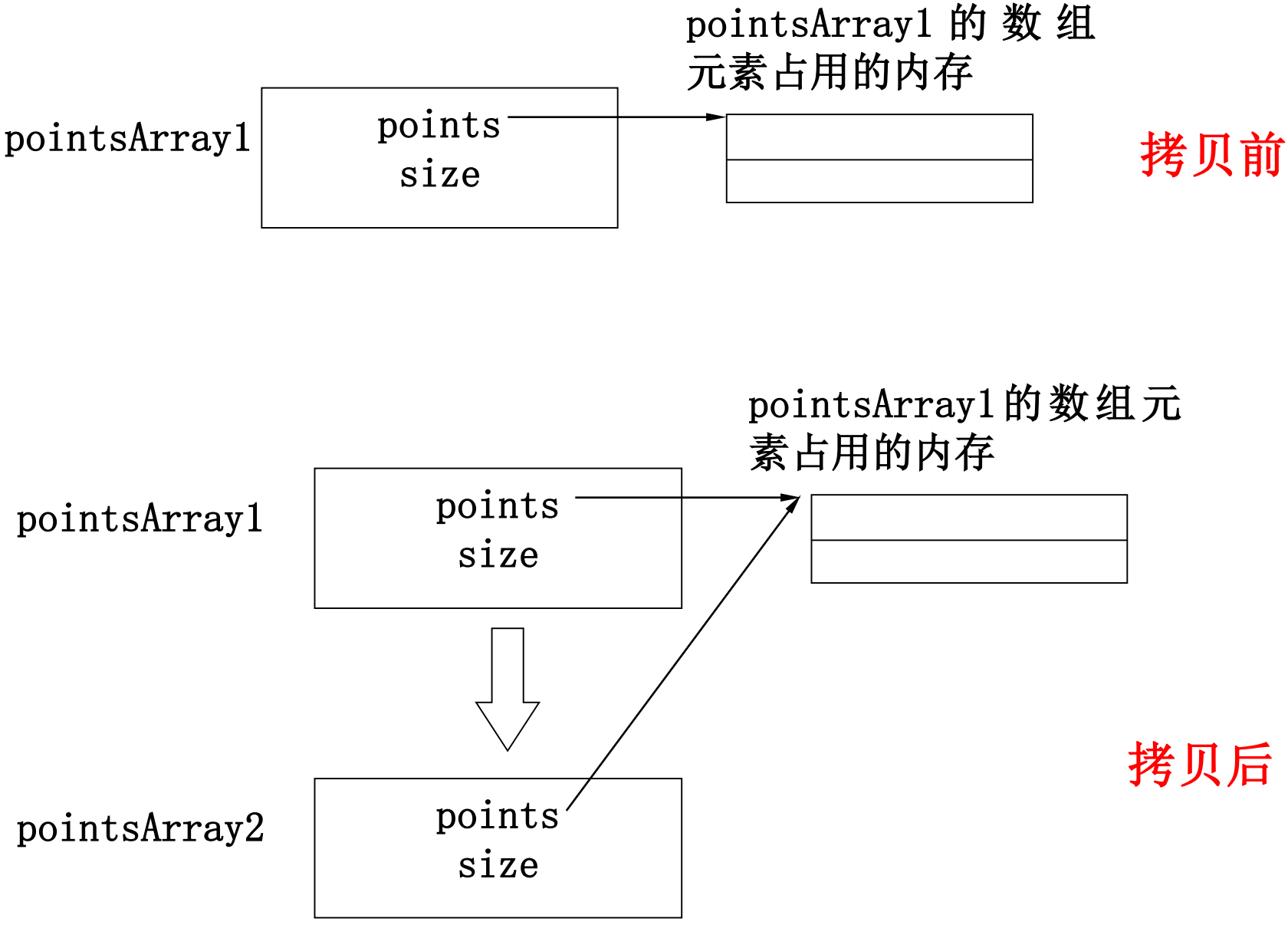
```
    pointsArray1.element(0).move(25,30);
    pointsArray1.element(1).move(35,40);
    cout << "After the moving of pointsArray1:" << endl;
    cout << "Point_0 of array2: " <<
    pointsArray2.element(0).getX() << ", "
         << pointsArray2.element(0).getY() << endl;
    cout << "Point_1 of array2: " <<
    pointsArray2.element(1).getX() << ", "
         << pointsArray2.element(1).getY() << endl;

    return 0;
}
```

运行结果如下：

```
Please enter the number of points:2
Default Constructor called.
Default Constructor called.
Copy of pointsArray1:
Point_0 of array2: 5,10
Point_1 of array2: 15,20
After the moving of pointsArray1:
Point_0 of array2: 25,30
Point_1 of array2: 35,40
Deleting...
Destructor called.
Destructor called.
Deleting...
```

接下来程序出现异常，也就是运行错误。


pointsArray1 的 数 组
元素占用的内存
pointsArray1
points
size
拷贝前
pointsArray1的数组元
素占用的内存
pointsArray1
points
size
pointsArray2
points
size
拷贝后

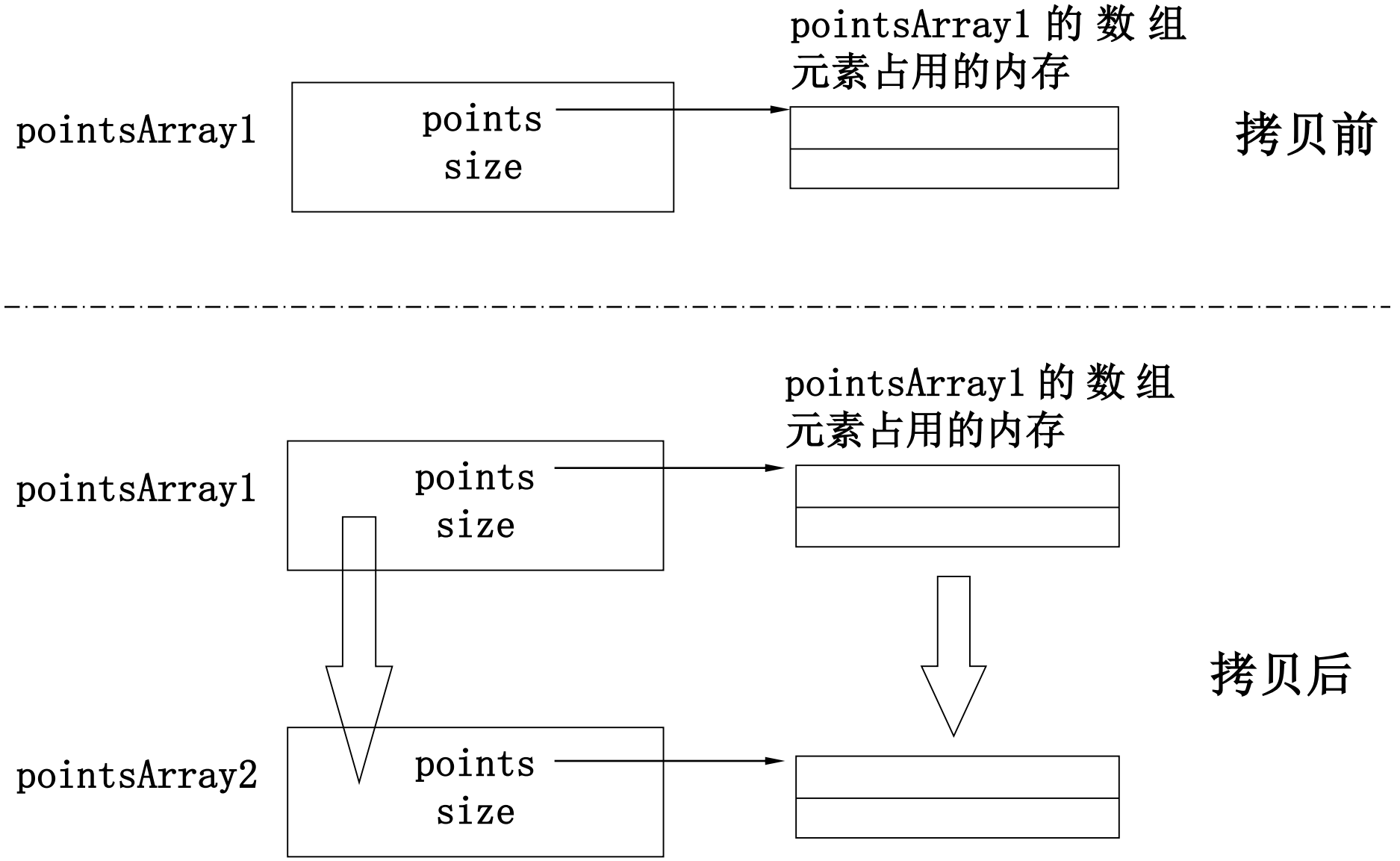
例 对象的深拷贝

```
#include <iostream>
#include <cassert>
using namespace std;
class Point {
  //类的声明同上例 ……
};
class ArrayOfPoints {
public:
   ArrayOfPoints(const ArrayOfPoints&
    pointsArray);
    //其他成员同上例
};
```

```
ArrayOfPoints::ArrayOfPoints(const ArrayOfPoints &v){
    size = v.size;
    points = new Point[size];
    for (int i = 0; i < size; i++)
        points[i] = v.points[i];
}
int main() {
    //同上例
}
```

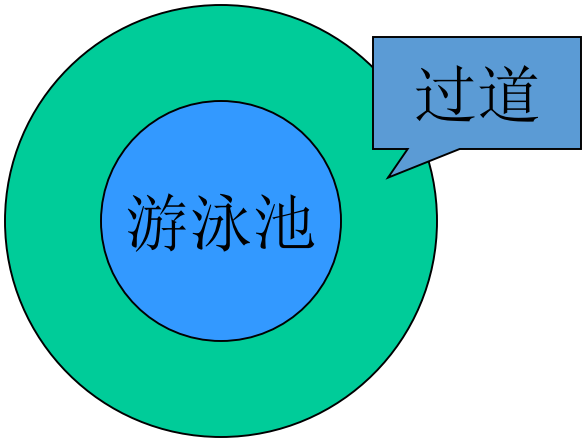
程序的运行结果如下：

```
Please enter the number of points:2
Default Constructor called.
Default Constructor called.
Default Constructor called.
Default Constructor called.
Copy of pointsArray1:
Point_0 of array2: 5, 10
Point_1 of array2: 15, 20
After the moving of pointsArray1:
Point_0 of array2: 5, 10
Point_1 of array2: 15, 20
Deleting...
Destructor called.
Destructor called.
Deleting...
Destructor called.
Destructor called.
```


pointsArray1 的 数 组
元素占用的内存
pointsArray1
points
size
拷贝前
pointsArray1 的 数 组
元素占用的内存
pointsArray1
points
size
拷贝后
pointsArray2
points
size

类的应用举例(例)

一圆形游泳池如图所示，现在需在其周围建一圆形过道，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>
using namespace std;
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
    /*函数后的修饰符const ------ 表示该成员函数的执 行不会改变类的状态, 也就是说不会修改类的数据成员。*/
    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;
}
```

```
void 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;
}
```

运行结果

```
Enter the radius of the pool: 10
Fencing Cost is ￥2858.85
Concrete Cost is ￥4335.39
```

类的组合

组合的概念

- 类中的成员数据是另一个类的对象。
- 可以在已有的抽象的基础上实现更复杂的抽象。

举例

```
class Point
{   private:
        float x,y; //点的坐标
    public:
        Point(float h,float v); //构造函数
        float GetX(void); //取X坐标
        float GetY(void); //取Y坐标
        void Draw(void); //在(x,y)处画点
};
//...函数的实现略
```

```
class Line
{
    private:
        Point  p1,p2; //线段的两个端点
    public:
        Line(Point a,Point b); //构造函数
        Void Draw(void); //画出线段
};
//...函数的实现略
```

类组合的构造函数设计

- 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。
- 声明形式：

  类名::类名(对象成员所需的形参，本类成员形参)

  ：对象1(参数)，对象2(参数)，.....

  { 本类初始化 }

类组合的构造函数调用

- 构造函数调用顺序：先调用内嵌对象的构造函数（按内嵌时的声明顺序，先声明者先构造）。然后调用本类的构造函数。（析构函数的调用顺序相反）
- 若调用默认构造函数（即无形参的），则内嵌对象的初始化也将调用相应的默认构造函数。

类的组合举例（二）

```
class Part  //部件类
{
    public:
        Part();
        Part(int i);
        ~Part();
        void Print();
    private:
        int val;
};
```

```
class Whole
{
    public:
        Whole();
        Whole(int i,int j,int k);
        ~Whole();
        void Print();
    private:
        Part one;
        Part two;
        int date;
};
```

```
Whole::Whole()
{
  date=0;
}
Whole::Whole(int i,int j,int k):
              two(i),one(j),date(k)
{}

//...其它函数的实现略
```

前向引用声明

- 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

```
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

- 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

- 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。

谢谢大家！