第 6 章　数组、指针与字符串（2）

郑 莉　清华大学

教材：C++语言程序设计（第5版） 郑莉　清华大学出版社

目录

指针数组

< 6.2.7 >

指针数组

- 数组的元素是指针型

例：Point *pa[2];

由pa[0],pa[1]两个指针组成

例6-8 利用指针数组存放矩阵

```
#include <iostream>
using namespace std;
int main() {
    int line1[] = { 1, 0, 0 };          //矩阵的第一行
    int line2[] = { 0, 1, 0 };          //矩阵的第二行
    int line3[] = { 0, 0, 1 };          //矩阵的第三行

    //定义整型指针数组并初始化
    int *pLine[3] = { line1, line2, line3 };
    cout << "Matrix test:" << endl;
   //输出矩阵
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        cout << pLine[i][j] << " ";
     cout << endl;
    }
    return 0;
}
```

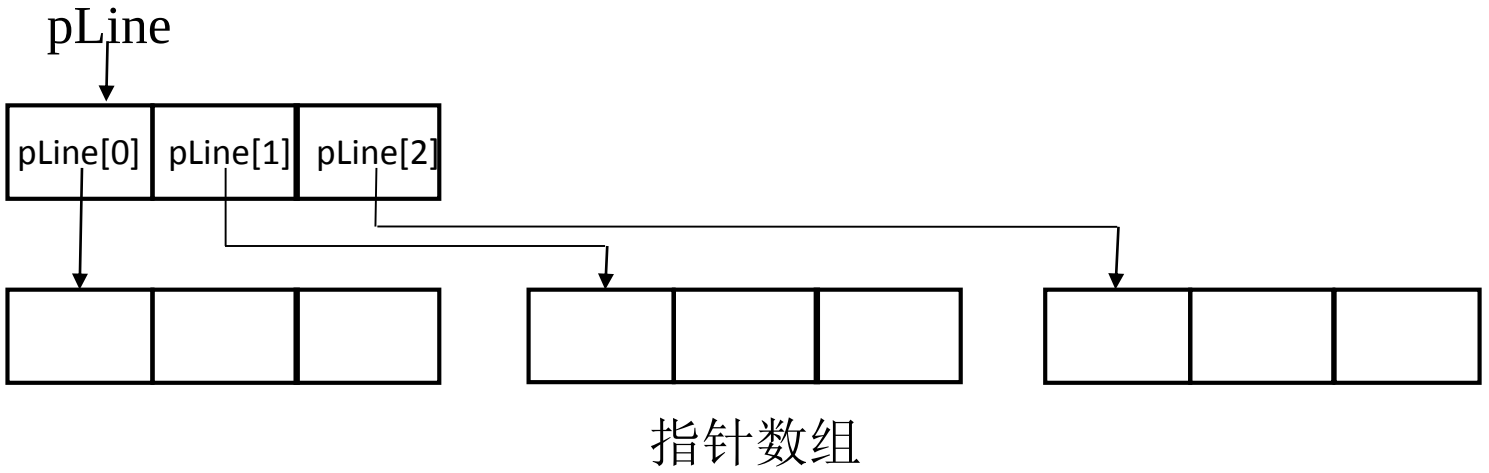
输出结果为：

```
Matrix test:
1,0,0
0,1,0
0,0,1
```

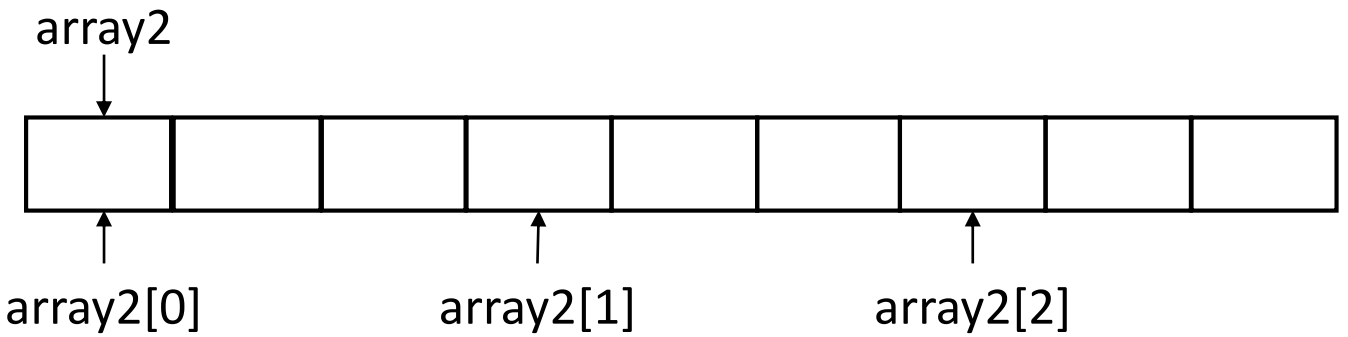
指针数组与二维数组对比

对比例6-8中的指针数组和如下二维数组

```
int array2[3][3] ={ { 1,0,0 }, { 0,1,0 }, { 0,0,1 } };
```



指针数组



二维数组

以指针作为函数参数

为什么需要用指针做参数？

- 需要数据双向传递时（引用也可以达到此效果）
  - 用指针作为函数的参数，可以使被调函数通过形参指针存取主调函数中实参指针指向的数据，实现数据的双向传递
- 需要传递一组数据，只传首地址运行效率比较高
  - 实参是数组名时形参可以是指针

例6-10

读入三个浮点数，将整数部分和小数部分分别输出

例6-10

```
#include <iostream>
using namespace std;
void splitFloat(float x, int *intPart, float *fracPart) {
   *intPart = static_cast<int>(x); //取x的整数部分
   *fracPart = x - *intPart; //取x的小数部分
}
int main() {
  cout << "Enter 3 float point numbers:" << endl;
  for(int i = 0; i < 3; i++) {
    float x, f;
    int n;
    cin >> x;
    splitFloat(x, &n, &f);          //变量地址作为实参
    cout << "Integer Part = " << n << " Fraction Part = " << f << endl;
  }
  return 0;
}
```

例: 指向常量的指针做形参

```
#include<iostream>
using namespace std;
const int N = 6;
void print(const int *p, int n);
int main() {
  int array[N];
  for (int i = 0; i < N; i++)
        cin>>array[i];
  print(array, N);
  return 0;
}
void print(const int *p, int n) {
  cout << "{ " << *p;
  for (int i = 1; i < n; i++)
        cout << ", " << *(p+i);
  cout << " }" << endl;
}
```

指针类型的函数

指针函数的定义形式

```
存储类型  数据类型  *函数名()
{ //函数体语句
}
```

注意

- 不要将非静态局部地址用作函数的返回值
  - 错误的例子：在子函数中定义局部变量后将其地址返回给主函数，就是非法地址

错误的例子

```
int main(){
    int* function();
    int* ptr= function();
    *prt=5; //危险的访问！
    return 0;
}
int* function(){
    int local=0; //非静态局部变量作用域和寿命都仅限于本函数体内
    return &local;
}//函数运行结束时，变量local被释放
```

注意

- 返回的指针要确保在主调函数中是有效、合法的地址
  - 正确的例子：

    主函数中定义的数组，在子函数中对该数组元素进行某种操作后，返回其中一个元素的地址，这就是合法有效的地址

正确的例子1

```
#include<iostream>
using namespace std;
int main(){
    int array[10]; //主函数中定义的数组
    int* search(int* a, int num);
    for(int i=0; i<10; i++)
      cin>>array[i];
    int* zeroptr= search(array, 10);  //将主函数中数组的首地址传给子函数
    return 0;
}
int* search(int* a, int num){ //指针a指向主函数中定义的数组
    for(int i=0; i<num; i++)
      if(a[i]==0)
        return &a[i]; //返回的地址指向的元素是在主函数中定义的
    return 0;
}//函数运行结束时，a[i]的地址仍有效
```

注意

- 返回的指针要确保在主调函数中是有效、合法的地址
  - 正确的例子：

    在子函数中通过动态内存分配new操作取得的内存地址返回给主函数是合法有效的，但是内存分配和释放不在同一级别，要注意不能忘记释放，避免内存泄漏

正确的例子2

```
#include<iostream>
using namespace std;
int main(){
    int* newintvar();
    int* intptr= newintvar();
    *intptr=5; //访问的是合法有效的地址
    delete intptr; //如果忘记在这里释放，会造成内存泄漏
    return 0;
}
int* newintvar (){
    int* p=new int();
    return p; //返回的地址指向的是动态分配的空间
}//函数运行结束时，p中的地址仍有效
```

指向函数的指针

函数指针的定义

- 定义形式

  存储类型　数据类型　(*函数指针名)();

- 含义
  - 函数指针指向的是程序代码存储区。

函数指针的典型用途——实现函数回调

- 通过函数指针调用的函数
  - 例如将函数的指针作为参数传递给一个函数，使得在处理相似事件的时候可以灵活的使用不同的方法。
- 调用者不关心谁是被调用者
  - 需知道存在一个具有特定原型和限制条件的被调用函数。

函数指针举例

编写一个计算函数compute，对两个整数进行各种计算。有一个形参为指向具体算法函数的指针，根据不同的实参函数，用不同的算法进行计算

编写三个函数：求两个整数的最大值、最小值、和。分别用这三个函数作为实参，测试compute函数

函数指针举例

```
#include <iostream>
using namespace std;

int compute(int a, int b, int(*func)(int, int))
{  return func(a, b);}

int max(int a, int b) // 求最大值
{  return ((a > b) ? a: b);}

int min(int a, int b) // 求最小值
{  return ((a < b) ? a: b);}

int sum(int a, int b) // 求和
{  return a + b;}
```

函数指针举例

```
int main()
{
    int a, b, res;
    cout << "请输入整数a："; cin >> a;
    cout << "请输入整数b："; cin >> b;

    res = compute(a, b, & max);
    cout << "Max of " << a << " and " << b << " is " << res << endl;
    res = compute(a, b, & min);
    cout << "Min of " << a << " and " << b << " is " << res << endl;
    res = compute(a, b, & sum);
    cout << "Sum of " << a << " and " << b << " is " << res << endl;
}
```

对象指针

对象指针

- 对象指针定义形式

  类名　*对象指针名；

  例:　Point a(5,10);

  Piont *ptr;

  ptr=&a;

- 通过指针访问对象成员

  对象指针名->成员名

  ptr->getx() 相当于 (*ptr).getx();

例6-12使用指针来访问Point类的成员

```
//6_12.cpp
#include <iostream>
using namespace std;
class Point {
public:
  Point(int x = 0, int y = 0) : x(x), y(y) { }
        int getX() const { return x; }
        int getY() const { return y; }
private:
        int x, y;
};
int main() {
   Point a(4, 5);
   Point *p1 = &a;      //定义对象指针，用a的地址初始化
   cout << p1->getX() << endl;//用指针访问对象成员
   cout << a.getX() << endl; //用对象名访问对象成员
   return 0;
}
```

this指针

- 隐含于类的每一个非静态成员函数中。
- 指出成员函数所操作的对象。
  - 当通过一个对象调用成员函数时，系统先将该对象的地址赋给this指针，然后调用成员函数，成员函数对对象的数据成员进行操作时，就隐含使用了this指针。
- 例如：Point类的getX函数中的语句：

```
return x;
```

相当于：

```
return this->x;
```

曾经出现过的错误例子

```
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;
};
```

正确的程序

```
class Fred;      //前向引用声明
class Barney {
    Fred *x;
 };
class Fred {
    Barney y;
 };
```

动态内存分配

动态申请内存操作符 new

- new 类型名T(初始化参数)
- 功能：
  - 在程序执行期间，申请用于存放T类型对象的内存空间，并依初始化参数进行初始化。
  - 基本类型初始化：如果有初始化参数，依初始化参数进行初始化；如果没有括号和初始化参数，不进行初始化，新分配的内存中内容不确定；如果有括号但初始化参数为空，初始化为0。
  - 对象类型：如果有初始化参数，以初始化参数中的值为参数调用构造函数进行初始化；如果没有括号和初始化参数或者有括号但初始化参数为空，用默认构造函数初始化。
- 结果值：成功：T类型的指针，指向新分配的内存；失败：抛出异常。

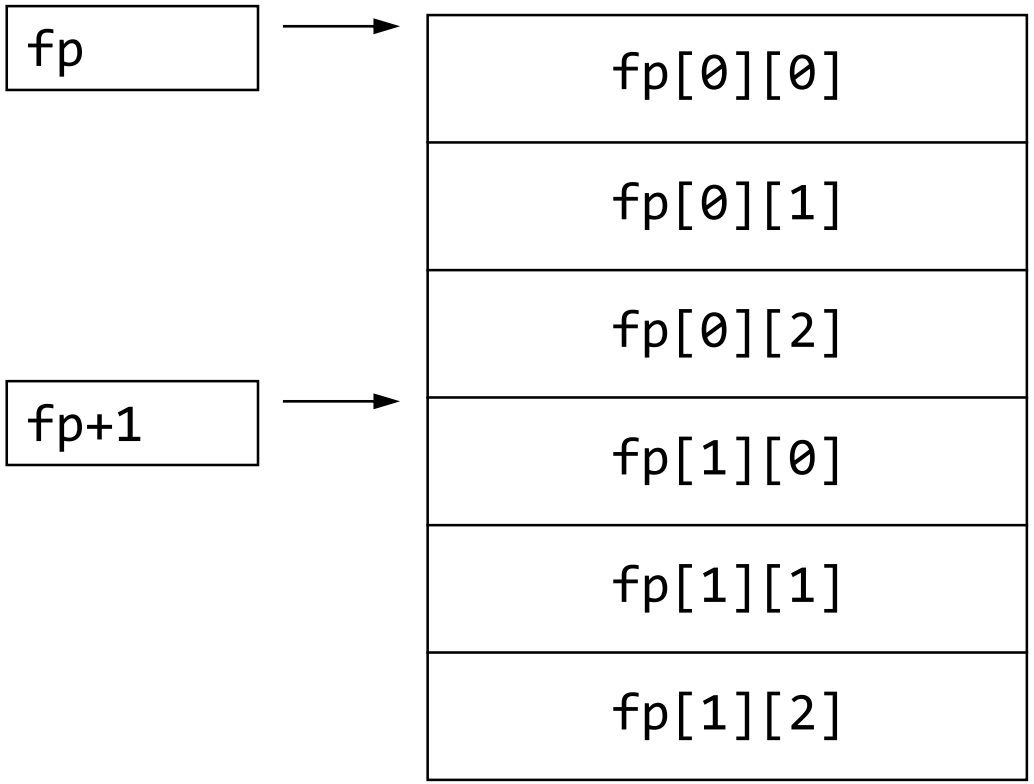
动态申请内存操作符 new （续）

- new 类型名T [表达式] [常量表达式]…… ()
- 功能：
  - 在程序执行期间，申请用于存放T类型对象数组的内存空间，可以有“()”但初始化列表必须为空。
  - 如果有“()”，对每个元素的初始化与执行“new T()”所做进行初始化的方式相同。
  - 如果没有“()”，对每个元素的初始化与执行“new T”所做进行初始化的方式相同。
- 结果值：
  - 如果内存申请成功，返回一个指向新分配内存首地址的指针。

    例如：

    ```
    double* array=new double[n]()；
    char (*fp)[3];
    fp = new char[n][3];
    ```
  - 如果失败：抛出异常。

```
char (*fp)[3];
```



释放内存操作符delete

- delete 指针p
- 功能：释放指针p所指向的内存。p必须是new操作的返回值。
- delete[] 指针p
- 功能：释放指针p所指向的数组。p必须是用new分配得到的数组首地址。

例6-16 动态创建对象举例

```
#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6-16 动态创建对象举例

```
int main() {
   cout << "Step one: " << endl;
   Point *ptr1 = new Point;   //调用默认构造函数
   delete ptr1;   //删除对象，自动调用析构函数

   cout << "Step two: " << endl;
   ptr1 = new Point(1,2);
   delete ptr1;

   return 0;
}
```

运行结果：

```
Step One:
Default Constructor called.
Destructor called.
Step Two:
Constructor called.
Destructor called.
```

例6-17 动态创建对象数组举例

```
#include<iostream>
using namespace std;
class Point {   //类的声明同例6-16，略 };
int main() {
   Point *ptr = new Point[2];      //创建对象数组
   ptr[0].move(5, 10);  //通过指针访问数组元素的成员
   ptr[1].move(15, 20); //通过指针访问数组元素的成员
   cout << "Deleting..." << endl;
   delete[] ptr;              //删除整个对象数组
   return 0;
}
```

运行结果：

```
Default Constructor called.
Default Constructor called.
Deleting...
Destructor called.
Destructor called.
```

例6-19 动态创建多维数组

```
#include <iostream>
using namespace std;
int main() {
  int (*cp)[9][8] = new int[7][9][8];
  for (int i = 0; i < 7; i++)
     for (int j = 0; j < 9; j++)
        for (int k = 0; k < 8; k++)
           *(*(*(cp + i) + j) + k) = ( i * 100 + j * 10 + k);
  for (int i = 0; i < 7; i++) {
     for (int j = 0; j < 9; j++) {
        for (int k = 0; k < 8; k++)
           cout << cp[i][j][k] << "  ";
        cout << endl;
     }
     cout << endl;
   }
   delete[] cp;
   return 0;
}
```

将动态数组封装成类

- 更加简洁，便于管理
  - 建立和删除数组的过程比较繁琐
  - 封装成类后更加简洁，便于管理
- 可以在访问数组元素前检查下标是否越界
  - 用assert来检查，assert只在调试时生效

例6-18 动态数组类

```
#include <iostream>
#include <cassert>
using namespace std;
class Point {   //类的声明同例6-16 …  };
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
};
```

例6-18 动态数组类

```
int main() {
   int count;
   cout << "Please enter the count of points: ";
   cin >> count;
   ArrayOfPoints points(count); //创建数组对象
   points.element(0).move(5, 0);   //访问数组元素的成员
   points.element(1).move(15, 20);  //访问数组元素的成员
   return 0;
}
```

思考：为什么element函数返回对象的引用？

- **返回“引用”可以用来操作封装数组对象内部的数组元素。如果返回“值”则只是返回了一个“副本”，通过“副本”是无法操作原来数组中的元素的**

运行结果：

```
Please enter the number of points:2
Default Constructor called.
Default Constructor called.
Deleting...
Destructor called.
Destructor called.
```

智能指针

- 显式管理内存在是能上有优势，但容易出错。
- C++11提供智能指针的数据类型，对垃圾回收技术提供了一些支持，实现一定程度的内存管理

C++11的智能指针

- unique_ptr ：不允许多个指针共享资源，可以用标准库中的move函数转移指针
- shared_ptr ：多个指针共享资源
- weak_ptr ：可复制shared_ptr，但其构造或者释放对资源不产生影响

vector对象

为什么需要vector？

- 封装任何类型的动态数组，自动创建和删除。
- 数组下标越界检查。
- 例6-18中封装的ArrayOfPoints也提供了类似功能，但只适用于一种类型的数组。

vector对象的定义

- vector<元素类型> 数组对象名(数组长度);
- 例：

  vector<int> arr(5)
  建立大小为5的int数组

vector对象的使用

- 对数组元素的引用
  - 与普通数组具有相同形式：
    - vector对象名 [下标表达式]
  - vector数组对象名不表示数组首地址
- 获得数组长度
  - 用size函数
    - 数组对象名.size()

例6-20 vector应用举例

```
#include <iostream>
#include <vector>
using namespace std;

//计算数组arr中元素的平均值
double average(const vector<double> &arr)
{
   double sum = 0;
   for (unsigned i = 0; i<arr.size(); i++)
         sum += arr[i];
   return sum / arr.size();
}
```

例6-20 vector应用举例

```
int main() {
   unsigned n;
   cout << "n = ";
   cin >> n;

   vector<double> arr(n);          //创建数组对象
   cout << "Please input " << n << " real numbers:" << endl;
   for (unsigned i = 0; i < n; i++)
         cin >> arr[i];

   cout << "Average = " << average(arr) << endl;
   return 0;
}
```

基于范围的for循环配合auto举例

```
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> v = {1,2,3};
    for(auto i = v.begin(); i != v.end(); ++i)
        std::cout << *i << std::endl;

for(auto e : v)
    std::cout << e << std::endl;
}
```

对象复制与移动

深层复制与浅层复制

- 浅层复制
  - 实现对象间数据元素的一一对应复制。
- 深层复制
  - 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，而是将指针所指对象进行复制。

例6-21 对象的浅层复制

```
#include <iostream>
#include <cassert>
using namespace std;
class Point {
   //类的声明同例6-16
   //……
};
class ArrayOfPoints {
   //类的声明同例6-18
   //……
};
```

例6-21 对象的浅层复制

```
int main() {
    int count;
    cout << "Please enter the count of points: ";
    cin >> count;
    ArrayOfPoints pointsArray1(count); //创建对象数组
    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;
```

例6-21 对象的浅层复制

```
    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;
}
```

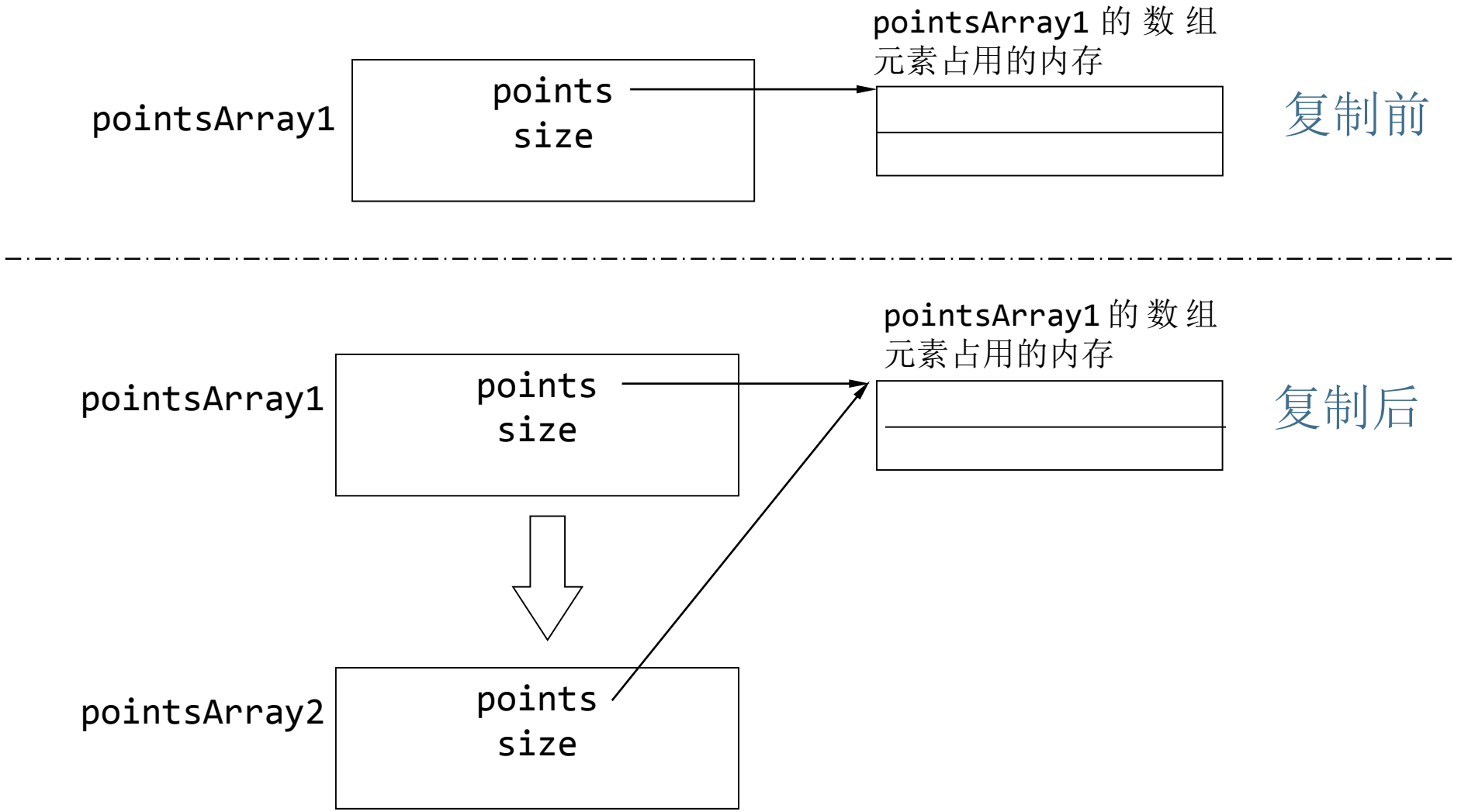
例6-21 对象的浅层复制

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...
```

接下来程序出现运行错误。

例6-21 对象的浅层复制



例6-22 对象的深层复制

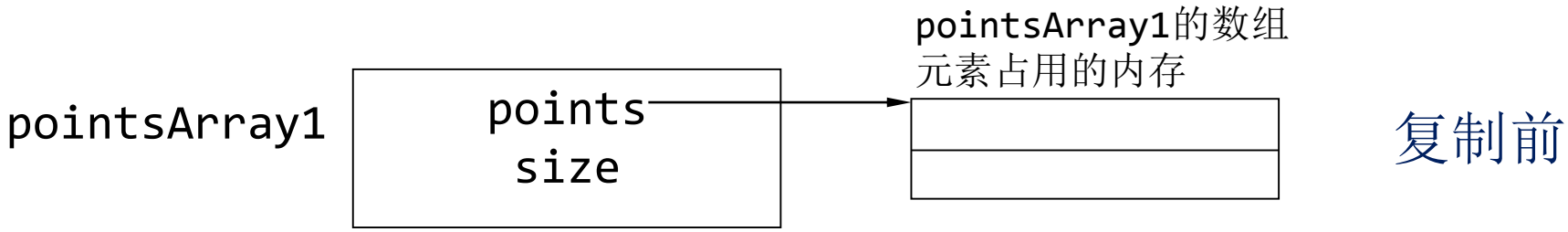
```
#include <iostream>
#include <cassert>
using namespace std;
class Point { //类的声明同例6-16
};
class ArrayOfPoints {
public:
   ArrayOfPoints(const ArrayOfPoints& pointsArray);
    //其他成员同例6-18
};
ArrayOfPoints::ArrayOfPoints(const ArrayOfPoints& v) {
    size = v.size;
    points = new Point[size];
    for (int i = 0; i < size; i++)
            points[i] = v.points[i];
}
int main() {
    //同例6-20
}
```

例6-22 对象的深层复制

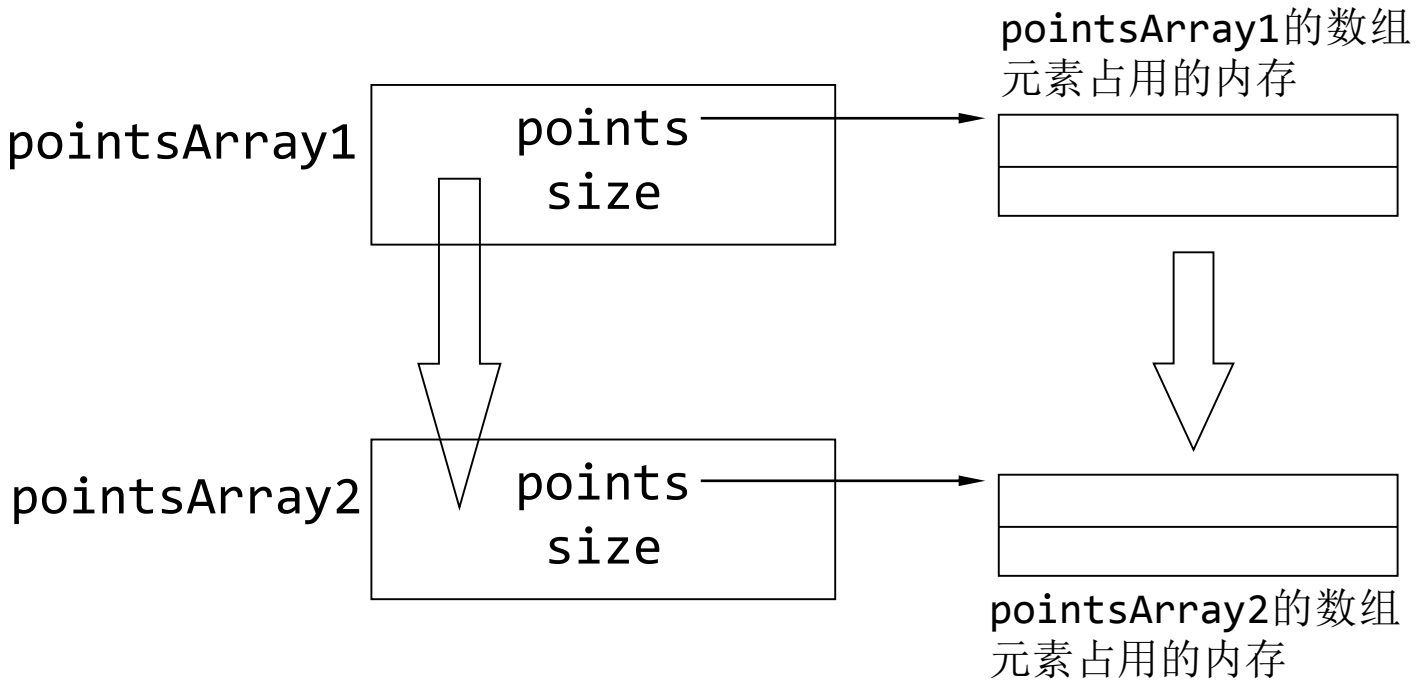
程序的运行结果如下：

```
Please enter the number of points:2
Default Constructor called.
Default Constructor called.
Default Constructor called.
Default Constructor called.
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.
```

例6-22 对象的深层复制



复制前



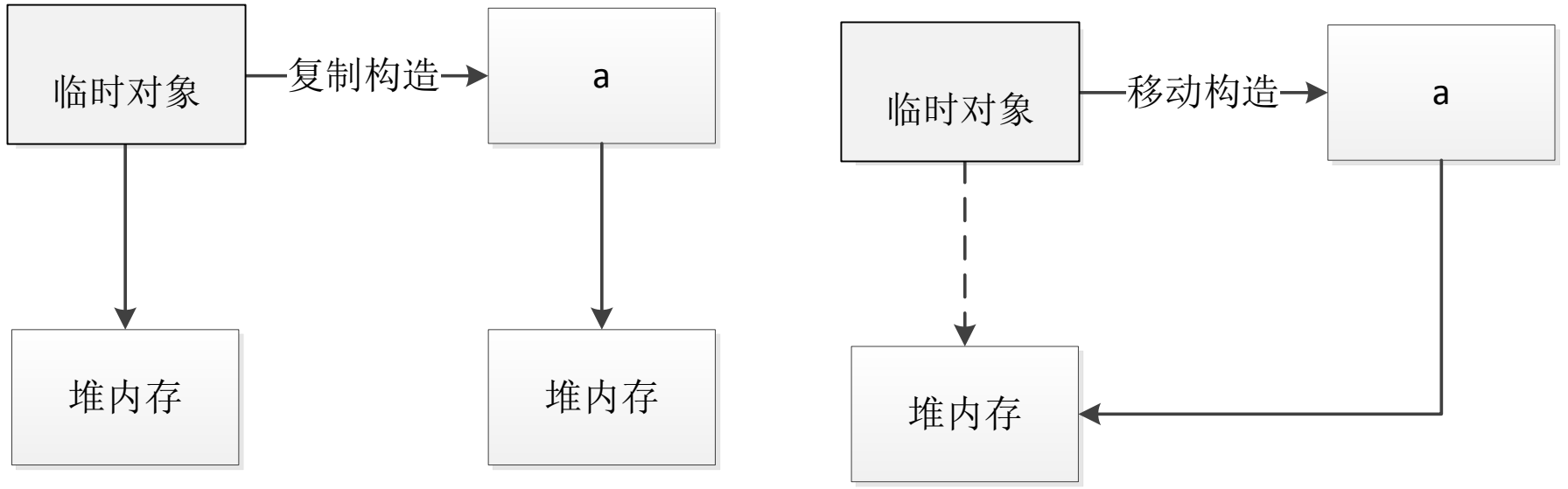
复制后

移动构造

- 什么时候该触发移动构造？
  - 有可被利用的临时对象

问题与解决

- 当临时对象在被复制后，就不再被利用了。我们完全可以把临时对象的资源直接移动，这样就避免了多余的复制操作。



移动构造

- **移动构造函数:**
  - class_name (class_name &&)

例：函数返回含有指针成员的对象

- 版本一：使用深层复制构造函数
  - 返回时构造临时对象，动态分配将临时对象返回到主调函数，然后删除临时对象。
- 版本二：使用移动构造函数
  - 将要返回的局部对象转移到主调函数，省去了构造和删除临时对象的过程。

版本一：使用复制构造

```
#include<iostream>
using namespace std;
class IntNum {
public:
        IntNum(int x = 0) : xptr(new int(x)){ //构造函数
                cout << "Calling constructor..." << endl;
        }
        IntNum(const IntNum & n) : xptr(new int(*n.xptr)){//复制构造函数
                cout << "Calling copy constructor..." << endl;
        };
        ~IntNum(){ //析构函数
                delete xptr;
                cout << "Destructing..." << endl;
        }
        int getInt() {   return *xptr;   }
private:
        int *xptr;
};
```

版本一：使用复制构造

```
//返回值为IntNum类对象
IntNum getNum() {
        IntNum a;
        return a;
}
int main() {
        cout<<getNum().getInt()<<endl;
        return 0;
}
```

运行结果：

```
Calling constructor...
Calling copy constructor...
Destructing...
0
Destructing...
```

版本二：使用移动构造

```
#include<iostream>
using namespace std;
class IntNum {
public:
        IntNum(int x = 0) : xptr(new int(x)){ //构造函数
                cout << "Calling constructor..." << endl;
        }
        IntNum(const IntNum & n) : xptr(new int(*n.xptr)){//复制构造函数
                cout << "Calling copy constructor..." << endl;
        }
        IntNum(IntNum && n): xptr( n.xptr){   //移动构造函数
                n.xptr = nullptr;
                cout << "Calling move constructor..." << endl;
        }
        ~IntNum(){ //析构函数
                delete xptr;
                cout << "Destructing..." << endl;
        }
private:
        int *xptr;
};
```

注：
- &&是右值引用
- 函数返回的临时变量是右值

版本二：使用移动构造

```
//返回值为IntNum类对象
IntNum getNum() {
  IntNum a;
  return a;
  //a是“将死之值”，a的引用是右值引用，于是调用移动构造函数，构造临时无名对象作为返回值
}
int main() {
cout << getNum().getInt() << endl; return 0;
}
```

运行结果：

```
Calling constructor...
Calling move constructor...
Destructing...
0
Destructing...
```

本章主要内容回顾

- 数组
- 指针
- 动态存储分配
- 指针与数组
- 指针与函数
- 字符串