# C++语言程序设计

## 第六章 数组 指针与字符串

# 本章主要内容

- 数组
- 指针
- 动态存储分配
- 指针与数组
- 指针与函数
- vector的基本用法
- 字符串
- 深度探索

# 数组的概念

数组

数组是具有一定顺序关系的若干相同类型变量的集合体，组成数组的变量称为该数组的元素。

数组属于构造类型。

# 一维数组的声明与引用

数组

- 一维数组的声明

  类型说明符　　数组名[ 常量表达式 ];

  数组名的构成方法与一般变量名相同。

  例如：int a[10];

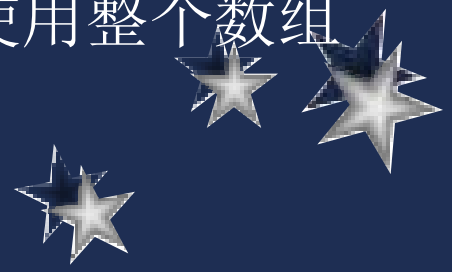
  表示 a 为整型数组，有10个元素：a[0]...a[9]

- 使用　　数组名[下标表达式]

  必须先声明，后使用。

  只能逐个使用数组元素，而不能一次使用整个数组

  例如：a[0]=a[5]+a[7]-a[2*3]



# 例6.1一维数组的声明与引用

数组

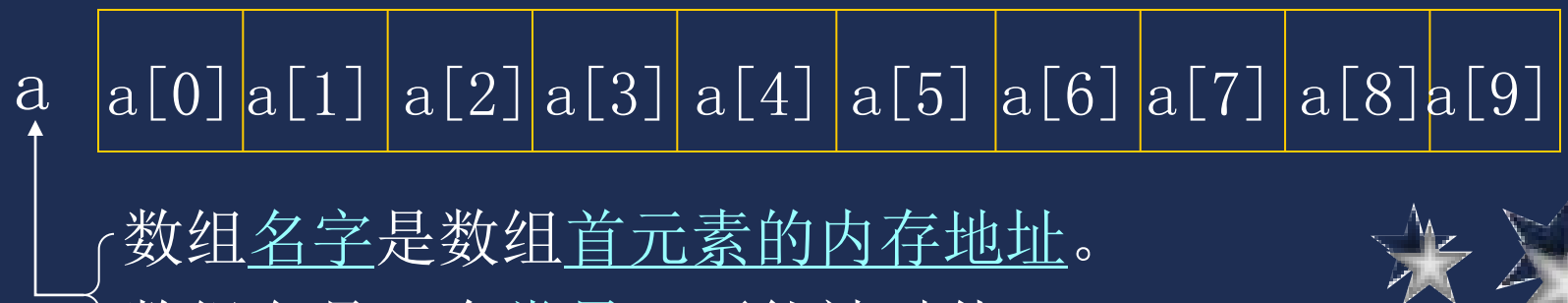
```
#include <iostream>
using namespace std;
int main() {
  int a[10], b[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 2 - 1;
    b[10 - i - 1] = a[i];
  }
  for(int i = 0; i < 10; i++) {
    cout << "a[" << i << "] = " << a[i] << "   ";
    cout << "b[" << I << "] = " << b[i] << endl;
  }
  return 0;
}
```

# 一维数组的存储顺序

数组

数组元素在内存中顺次存放，它们的地址是连续的。

例如：具有10个元素的数组 a，在内存中的存放次序如下：




数组名字是数组首元素的内存地址。

数组名是一个常量，不能被赋值。

# 一维数组的初始化

数组

可以在定义数组的同时赋给初值：

- 在声明数组时对数组元素赋以初值。

  例如：int a[10]={0,1,2,3,4,5,6,7,8,9};

- 可以只给一部分元素赋初值。

  例如：int a[10]={0,1,2,3,4};

- 在对全部数组元素赋初值时，可以不指定数组长度。

  例如：int a[]={1,2,3,4,5}

# 例：用数组来处理求Fibonacci数列问题

```
#include <iostream>
using namespace std;
int main() {
  int f[20] = {1,1};//初始化第0、1个数
  for (int i = 2; i < 20; i++) //求第2～19个数
    f[i] = f[i - 2] + f[i - 1];
  for (i=0;i<20;i++) { //输出，每行5个数
    if (i % 5 == 0) cout << endl;
    cout.width(12); //设置输出宽度为12
    cout << f[i];
  }
  return 0;
}
```

# 例：用数组来处理求Fibonacci数列问题

运行结果：

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 5 |
| 8 | 13 | 21 | 34 | 55 |
| 89 | 144 | 233 | 377 | 610 |
| 987 | 1597 | 2584 | 4181 | 6765 |

# 一维数组应用举例

数组

循环从键盘读入若干组选择题答案，计算并输出每组答案的正确率，直到输入ctrl+z为止。

每组连续输入5个答案，每个答案可以是'a'..'d'。

```
#include <iostream>
using namespace std;
int main() {
  const char KEY[ ] = {'a','c','b','a','d'};
  const int NUM_QUES = 5;
  char c;
  int ques = 0, numCorrect = 0;
  cout << "Enter the " << NUM_QUES << " question tests:" <<
   endl;
  while(cin.get(c)) {
    if(c != '\n') {
      if(c == key[ques]) {
        numCorrect++; cout << " ";
       } else
        cout<<"*";
       ques++;
    } else {
      cout << " Score "
        << static_cast<float>(numCorrect)/NUM_QUES*100<<"%";
      ques = 0;
      numCorrect = 0;
      cout << endl;
    }
  }
  return 0;
}
```

运行结果:

```
acbba
   **  Score  60%

acbad
       Score  100%

abbda
 * **  Score  40%

bdcba
*****  Score  0%
```

# 二维数组的声明及使用

数组

数据类型 标识符[常量表达式1][常量表达式2] …;

例:

int a[5][3];

表示a为整型二维数组，其中第一维有5个下标（0~4），第二维有3个下标（0~2），数组的元素个数为15，可以用于存放5行3列的整型数据表格。

# 二维数组的声明及引用

数组

- 二维数组的声明

  类型说明符 数组名[常量表达式1][常量表达式2]

  例如：float a[3][4];

  可以理解为：

  a
  - a[0]——$a_{00}$ $a_{01}$ $a_{02}$ $a_{03}$
  - a[1]——$a_{10}$ $a_{11}$ $a_{12}$ $a_{13}$
  - a[2]——$a_{20}$ $a_{21}$ $a_{22}$ $a_{23}$

- 存储顺序

  按行存放，上例中数组a的存储顺序为：

  $a_{00}$ $a_{01}$ $a_{02}$ $a_{03}$ $a_{10}$ $a_{11}$ $a_{12}$ $a_{13}$ $a_{20}$ $a_{21}$ $a_{22}$ $a_{23}$

- 使用

  例如：b[1][2]=a[2][3]/2

下标不要越界

# 二维数组的初始化

数组

- 将所有数据写在一个{}内，按顺序赋值

  例如：int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

- 分行给二维数组赋初值

  例如：int a[3][4]

  ={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

- 可以对部分元素赋初值

  例如：int a[3][4]={{1},{0,6},{0,0,11}};

# 数组作为函数参数

数组

- 数组元素作实参，与单个变量一样。
- 数组名作参数，形、实参数都应是数组名，类型要一样，传送的是数组首地址。对形参数组的改变会直接影响到实参数组。

# 例6-2 使用数组名作为函数参数

数组

- 主函数中初始化一个矩阵并将每个元素都输出，然后调用子函数，分别计算每一行的元素之和，将和直接存放在每行的第一个元素中，返回主函数之后输出各行元素的和。

```
#include <iostream>
using namespace std;
void rowSum(int a[][4], int nRow) {
  for (int i = 0; i < nRow; i++) {
     for(int j = 1; j < 4; j++)
          a[i][0] += a[i][j];
  }
}
int main() {   //主函数
  int table[3][4] = {{1, 2, 3, 4},
    {2, 3, 4, 5}, {3, 4, 5, 6}};
  //声明并初始化数组
```

```
    //输出数组元素
    for (int i = 0; i < 3; i++)  {
        for (int j = 0; j < 4; j++)
            cout << table[i][j] << "    ";
        cout << endl;
    }
    rowSum(table, 3); //调用子函数，计算各行和
    //输出计算结果
    for (int i = 0; i < 3; i++)
        cout << "Sum of row " << i << " is " <<
    table[i][0] << endl;
    return 0;
}
```

运行结果：

```
1   2   3   4
2   3   4   5
3   4   5   6
Sum of row 0 is 10
Sum of row 1 is 14
Sum of row 2 is 18
```

# 对象数组

数组

- 声明：

  类名　数组名[元素个数];

- 访问方法：

  通过下标访问

  数组名[下标].成员名

# 对象数组初始化

数组

- 数组中每一个元素对象被创建时，系统都会调用类构造函数初始化该对象。
- 通过初始化列表赋值。

  例：

```
Point a[2]={Point(1,2),Point(3,4)};
```

- 如果没有为数组元素指定显式初始值，数组元素便使用默认值初始化（调用缺省构造函数）。

# 数组元素所属类的构造函数

数组

- 不声明构造函数，则采用缺省构造函数。
- 各元素对象的初值要求为相同的值时，可以声明具有默认形参值的构造函数。
- 各元素对象的初值要求为不同的值时，需要声明带形参的构造函数。
- 当数组中每一个对象被删除时，系统都要调用一次析构函数。

# 例6-3 对象数组应用举例

数

组

```
//Point.h
#ifndef _POINT_H
#define _POINT_H

class Point {  //类的定义
public:  //外部接口
  Point();
  Point(int x, int y);
  ~Point();
  void move(int newX,int newY);
  int getX() const { return x; }
  int getY() const { return y; }
private: //私有数据成员
  int x, y;
};
#endif   //_POINT_H
```

```
//Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

Point::Point() {
   x = y = 0;
   cout << "Default Constructor called." << endl;
}

Point::Point(int x, int y) : x(x), y(y) {
   cout << "Constructor called." << endl;
}

Point::~Point() {
   cout << "Destructor called." << endl;
}

void Point::move(int newX,int newY) {
   cout << "Moving the point to (" << newX << ", " <<
   newY << ")" << endl;
   x = newX;
   y = newY;
}
```

```
//6-3.cpp
#include "Point.h"
#include <iostream>
using namespace std;

int main() {
  cout << "Entering main..." << endl;
  Point a[2];
  for(int i = 0; i < 2; i++)
      a[i].move(i + 10, i + 20);
  cout << "Exiting main..." << endl;
  return 0;
}
```

运行结果:

```
Entering main...
Default Constructor called.
Default Constructor called.
Moving the point to (10, 20)
Moving the point to (11, 21)
Exiting main...
Destructor called.
Destructor called.
```

# 关于内存地址

- 内存空间的访问方式
  - 通过变量名访问
  - 通过地址访问
- 取地址运算符：&

  例：

  ```
  int var;
  ```

  则&var 表示变量var在内存中的起始地址

# 指针变量的概念

指针

## 概念

**指针：**内存地址，用于间接访问内存单元

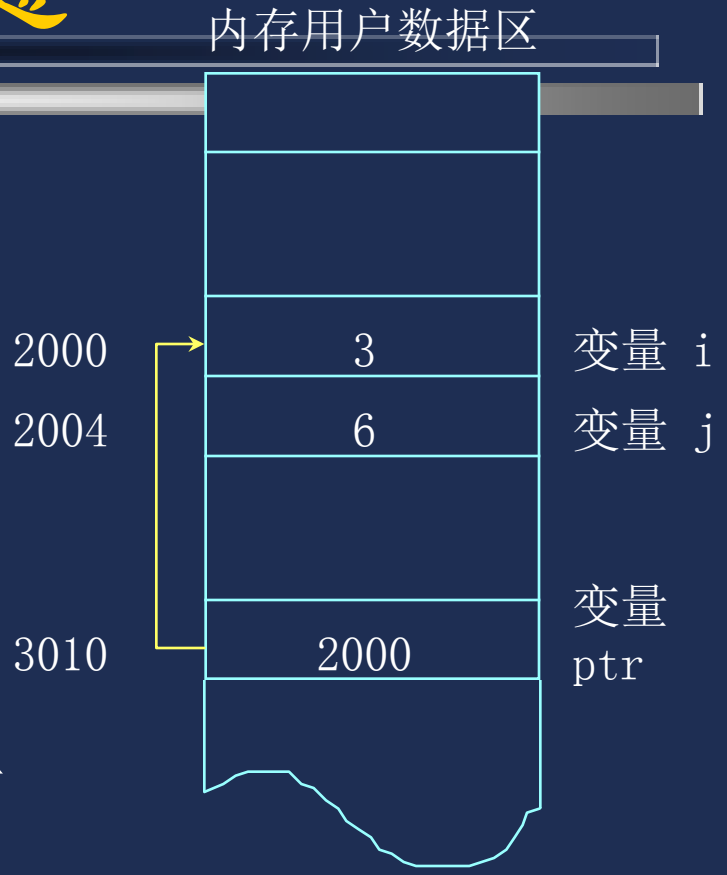
**指针变量：**用于存放地址的变量

## 声明

例：int i;

int *ptr = &i;

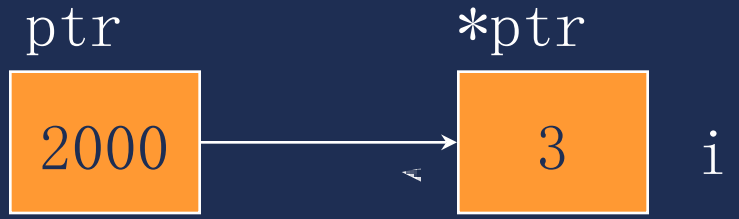
指向整型变量的指针




## 引用

例1：i = 3;

例2：*ptr = 3;




2000

# 指针变量的初始化

指针

- 语法形式

  数据类型 *指针名＝初始地址;

  例：int *pa = &a;

- 注意事项
  - 用变量地址作为初值时，该变量必须在指针初始化之前已声明过，且变量类型应与指针类型一致。
  - 可以用一个已赋初值的指针去初始化另一个指针变量。
  - 初始化为零指针：0或NULL

# 指针变量的赋值运算

指针

指针名=地址

- “地址”中存放的数据类型与指针类型必须相符。
- 向指针变量赋的值必须是地址常量或变量，不能是普通整数。但可以赋值为整数0，表示空指针。
- 指针的类型是它所指向变量的类型，而不是指针本身数据值的类型，任何一个指针本身的数据值都是unsigned long int型。
- 允许声明指向 void 类型的指针。该指针可以被赋予任何类型对象的地址。

例： void *general;

# 例6-5 指针的声明、赋值与使用

指针

```
#include <iostream>
using namespace std;
int main() {
    int i;                          //定义int型数i
    int *ptr = &i;  //取i的地址赋给ptr
    i = 10;                         //int型数赋初值
    cout << "i = " << i << endl;//输出int型数的值
    cout << "*ptr = " << *ptr << endl;//输出int型指
    针所指地址的内容
    return 0;
}
```

程序运行的结果是：

```
i = 10
*ptr = 10
```

# 例6-6 void类型指针的使用

指针

```
#include <iostream>
using namespace std;
int main() {
//!void voidObject;错，不能声明void类型的变量
   void *pv; //对，可以声明void类型的指针
   int i = 5;
   pv = &i;  //void类型指针指向整型变量
   int *pint = static_cast<int *>(pv);
   //void类型指针赋值给int类型指针
   cout << "*pint = " << *pint << endl;
   return 0;
}
```

# 指向常量的指针

指针

- 不能通过指针来改变所指对象的值，但指针本身可以改变，可以指向另外的对象。
- 例

```
int a;
const int *p1 = &a; //p1是指向常量的指针
int b;
p1 = &b;//正确，p1本身的值可以改变
*p1 = 1;//编译时出错，不能通过p1改变所指的对象
```

# 指针类型的常量

指针

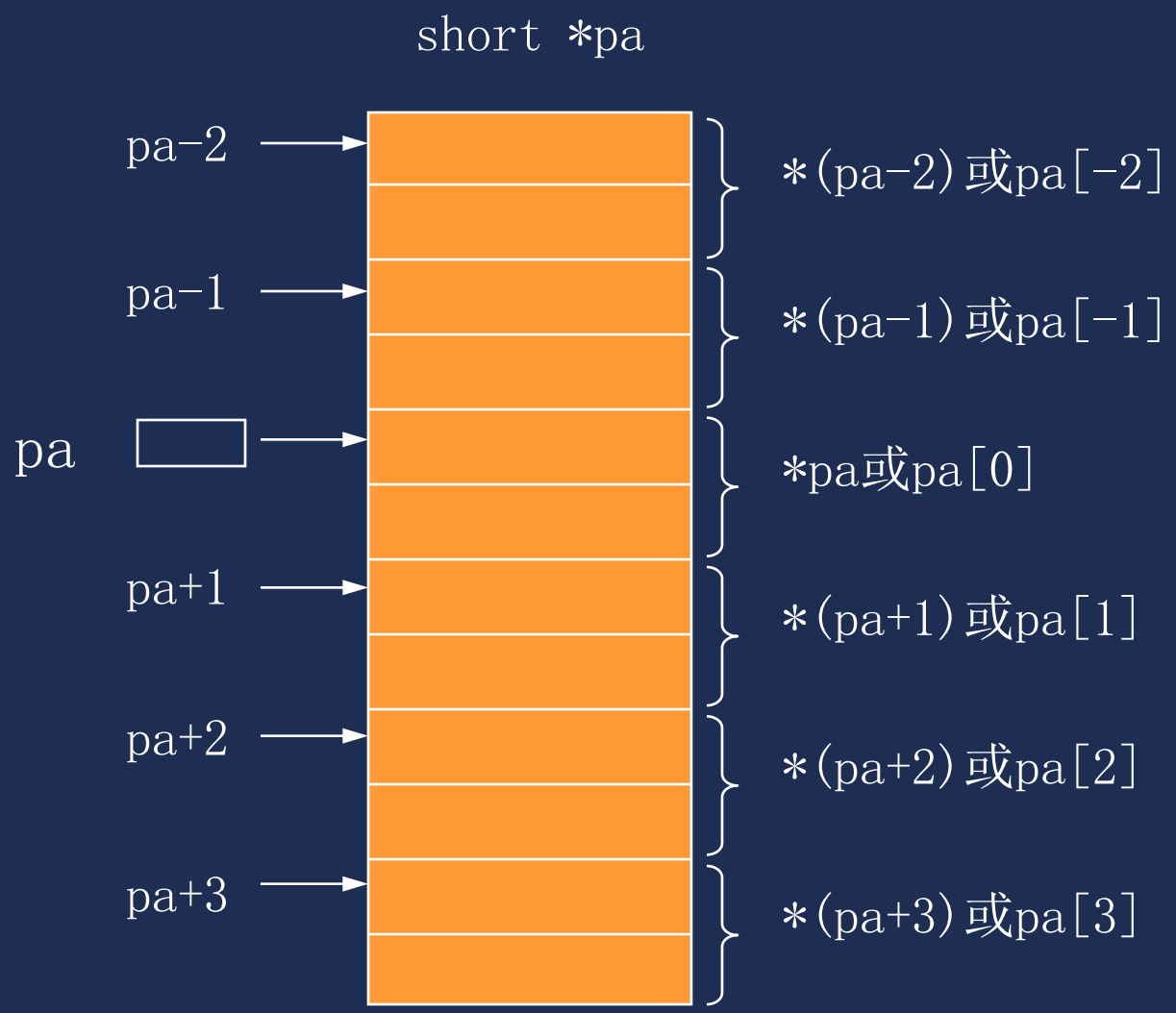
- 若声明指针常量，则指针本身的值不能被改变。
- 例：

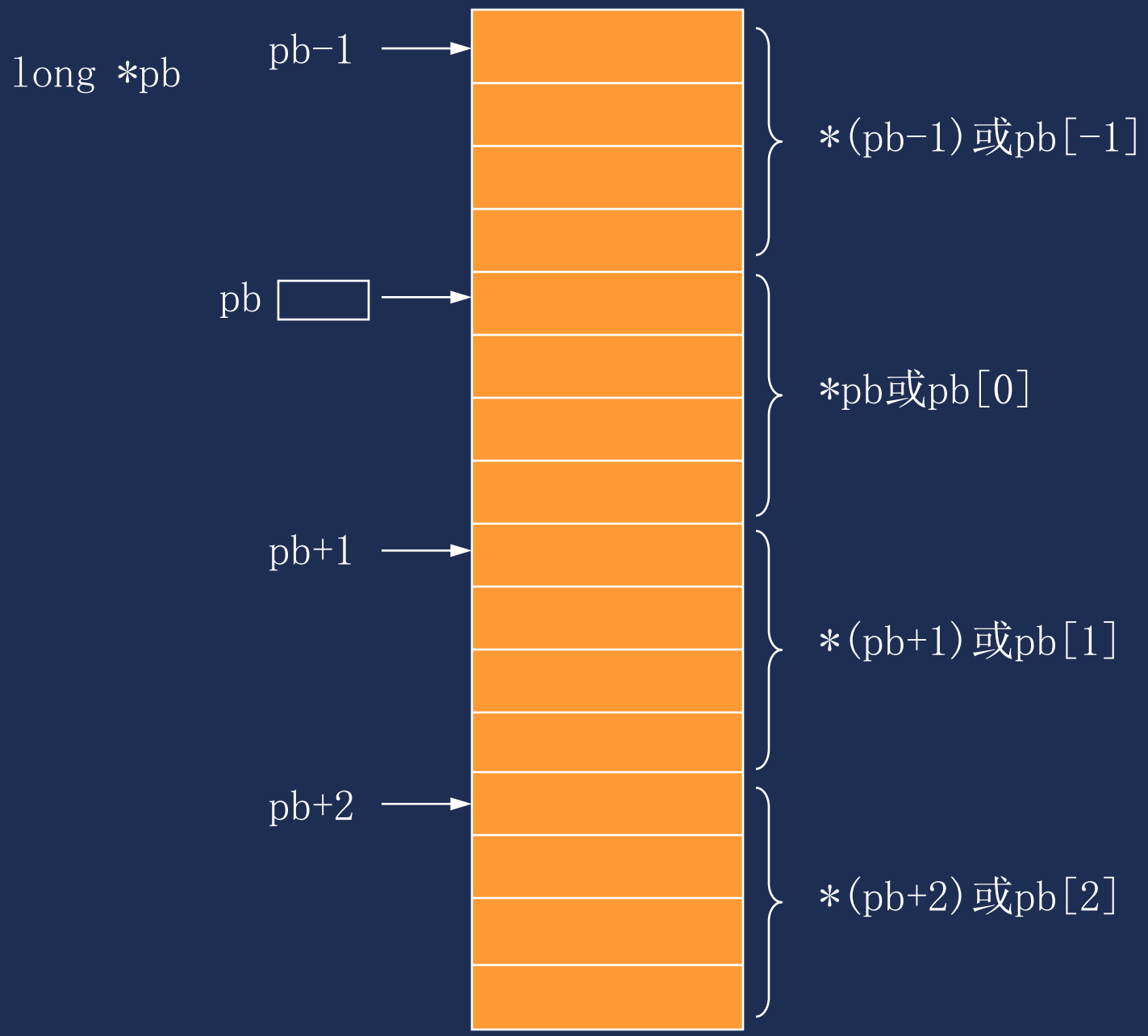
```
int a;
int * const p2 = &a;
p2 = &b;//错误，p2是指针常量，值不能改变
```

# 指针变量的算术运算

指针

- 指针与整数的加减运算
  - 指针p加上或减去n，其意义是指针当前指向位置的前方或后方第n个数据的地址。
  - 这种运算的结果值取决于指针指向的数据类型。
  - p1[n1]等价于*(p1 + n1)
- 指针加一，减一运算
  - 指向下一个或前一个数据。
  - 例如：y=*px++ 相当于 y=*(px++)
    (*和++优先级相同，自右向左运算)


short *pa
pa-2
pa-1
pa
pa+1
pa+2
pa+3
*(pa-2)或pa[-2]
*(pa-1)或pa[-1]
*pa或pa[0]
*(pa+1)或pa[1]
*(pa+2)或pa[2]
*(pa+3)或pa[3]


long *pb
pb-1
pb
pb+1
pb+2
*(pb-1)或pb[-1]
*pb或pb[0]
*(pb+1)或pb[1]
*(pb+2)或pb[2]

# 指针变量的关系运算

指针

- 关系运算
  - 指向相同类型数据的指针之间可以进行各种关系运算。
  - 指向不同数据类型的指针，以及指针与一般整数变量之间的关系运算是无意义的。
  - 指针可以和零之间进行等于或不等于的关系运算。例如：p==0或p!=0
- 赋值运算
  - 向指针变量赋的值必须是地址常量或变量，不能是普通整数。但可以赋值为整数0，表示空指针。

# 指向数组元素的指针

指针

- 声明与赋值

  例：int a[10], *pa;

  pa=&a[0]; 或 pa=a;

- 通过指针引用数组元素

  经过上述声明及赋值后：

  - *pa就是a[0]，*(pa+1)就是a[1]，... ，*(pa+i)就是a[i].
  - a[i]，*(pa+i)，*(a+i)，pa[i]都是等效的。
  - 不能写 a++，因为a是数组首地址是常量。

# 例6-7

指针

设有一个int型数组a，有10个元素。用三种方法输出各元素：

- 使用数组名和下标
- 使用数组名和指针运算
- 使用指针变量

# 使用数组名和下标

```
#include <iostream>
using namespace std;
int main() {
  int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8,
  9, 0 };
  for (int i = 0; i < 10; i++)
      cout << a[i] << "  ";
  cout << endl;
  return 0;
}
```

# 使用数组名指针运算

```
#include <iostream>
using namespace std;
int main() {
  int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8,
  9, 0 };
  for (int i = 0; i < 10; i++)
      cout << *(a+i) << "  ";
  cout << endl;
  return 0;
}
```

# 使用指针变量

```
#include <iostream>
using namespace std;
int main() {
  int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8,
  9, 0 };
  for (int *p = a; p < (a + 10); p++)
      cout << *p << "  ";
  cout << endl;
  return 0;
}
```

# 指针数组

指针

- 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 利用指针数组存放单位矩阵

指针

```
#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;
    //输出单位矩阵
    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
```

pLine[i][j]等价于*(pLine[i] + j)

# 例6-9 二维数组举例

指

针

```
#include <iostream>
using namespace std;
int main() {
  int array2[3][3]= { { 11, 12, 13 },
    { 21, 22, 23 }, { 31, 32, 33 } };
  for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++)
      cout << *(*(array2 + i) + j) << " ";
      //逐个输出二维数组第i行元素值
      cout << endl;
  }
  return 0;
}
```

*(*(array2 +i) + j)等价于array2[i][j]

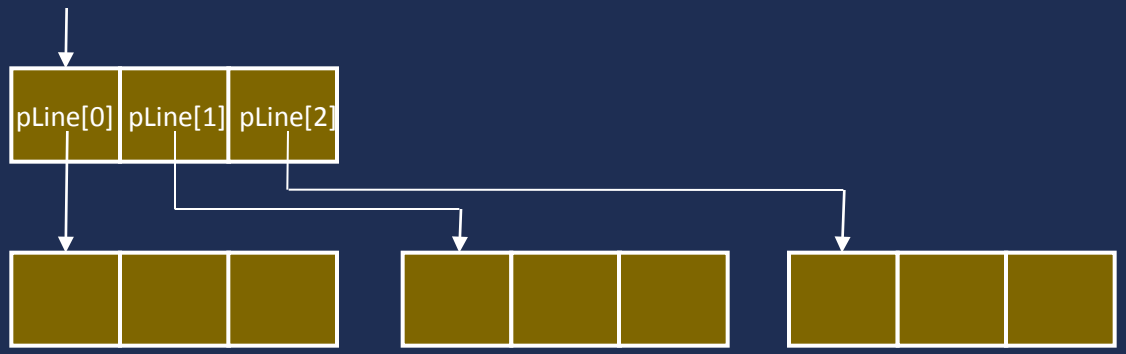
程序的输出结果为：

11 12 13

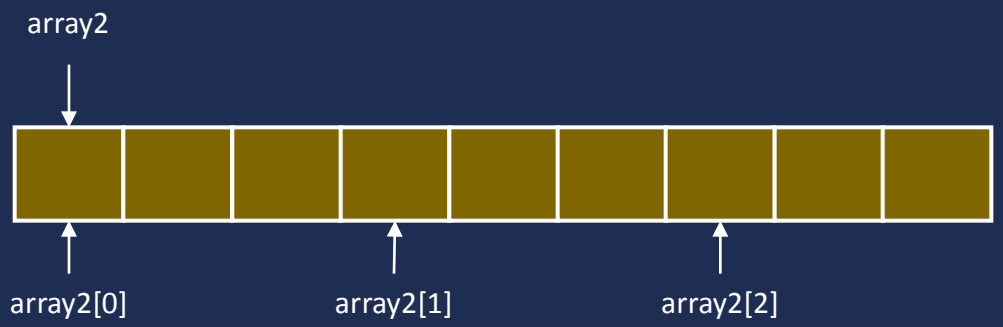
21 22 23

31 32 33

# 指针数组 VS 二维数组




(a) 指针数组




(b) 二维数组

# 以指针作为函数参数

指针与函数

- 以地址方式传递数据，可以用来返回函数处理结果。
- 实参是数组名时，形参可以是指针。
  - void f(int p[]);
  - void f(int p[3]);
  - void f(int *p);

# 例6.10



题目：读入三个浮点数，将整数部分和小数部分分别输出

```
#include <iostream>
using namespace std;
void splitFloat(float x, int *intPart, float *fracPart) {
  //取x的整数部分
  *intPart = static_cast<int>(x);
  //取x的小数部分
  *fracPart = x - *intPart;
}
```

```
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;
}
```

运行结果：

```
Enter 3 floating point numbers
4.7
Integer Part = 4  Fraction Part = 0.7
8.913
Integer Part = 8  Fraction Part = 0.913
-4.7518
Integer Part = -4  Fraction Part = -0.7518
```

# 例：输出数组元素的内容和地址



```
#include <iostream>
#include <iomanip>
using namespace std;
void arrayPtr(long *p, int n) {
  cout << "In func, address of array is "
       << p << endl;
  cout << "Accessing array using pointers" << endl;
  for (int i = 0; i < n; i++) {
    cout << "   Address for index " << i << " is "
         << p + i;
    cout << "  Value is " << *(p + i) << endl;
  }
}
```

```
int main() {
  long list[5]={50, 60, 70, 80, 90};

  cout << "In main, address of array is "
       << list << endl;
  cout << endl;

  arrayPtr(list,5);

  return 0;
}
```

运行结果：

```
In main, address of array is 0012FF50

In func, address of array is 0012FF50
Accessing array using pointers
   Address for index 0 is 0012FF50  Value is 50
   Address for index 1 is 0012FF54  Value is 60
   Address for index 2 is 0012FF58  Value is 70
   Address for index 3 is 0012FF5C  Value is 80
   Address for index 4 is 0012FF60  Value is 90
```

# 指向常量的指针做形参

指针

```
#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;
}
```

# 指针型函数

指针与函数

当函数的返回值是地址时，该函数就是指针形函数。

声明形式

```
数据类型  *函数名(形参表)
```

# 指向函数的指针



- 声明形式

  数据类型　(*函数指针名)(形参表);

- 含义：
  - 数据指针指向数据存储区，而函数指针指向的是程序代码存储区。

# 指向函数的指针

指针与函数

- 赋值
  - 函数指针名 = 函数名;
  - 函数指针名 = &函数名;
  - 函数指针名 = 0; // NULL
- 使用函数指针调用函数
  - 函数指针名(实参表);
  - (*函数指针名)(实参表);

# 例6-11函数指针

指针与函数

```
#include <iostream>
using namespace std;
void printStuff(float) {
   cout << "This is the print stuff function."
        << endl;
}
void printMessage(float data) {
   cout << "The data to be listed is "
        << data << endl;
}
void printFloat(float data) {
   cout << "The data to be printed is "
        << data << endl;
}
```

```
const float PI = 3.14159f;
const float TWO_PI = PI * 2.0f;

int main() { //主函数
   void (*functionPointer)(float); //函数指针
   printStuff(PI);
   functionPointer = printStuff;
   functionPointer(PI);           //函数指针调用
   functionPointer = printMessage;
   functionPointer(TWO_PI);       //函数指针调用
   functionPointer(13.0);         //函数指针调用
   functionPointer = &printFloat;
   (*functionPointer)(PI);        //函数指针调用
   printFloat(PI);
   return 0;
}
```

运行结果：

```
This is the print stuff function.
This is the print stuff function.
The data to be listed is 6.28318
The data to be listed is 13
The data to be printed is 3.14159
The data to be printed is 3.14159
```

# 对象指针的一般概念

指针

- 声明形式

  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();

# 对象指针应用举例

指针

```
int main() {
    Point a(5, 10);
    Point *ptr;
    ptr = &a;
    int x;
    x = ptr->getX();
    cout << x << endl;
    return 0;
}
```

# 曾经出现过的错误例子

指针

```
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

指针

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

# this指针

指针

- 隐含于每一个类的成员函数中的特殊指针。
  - this是一个指针常量。
  - 对于常成员函数，是指向常量的指针常量。
- 明确地指出了成员函数当前所操作的数据所属的对象。
  - 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。

# this指针

指针

例如：Point类的getX函数中的语句：

```
return x;
```

相当于：

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

或

```
return (*this).x;
```

# 指向类的非静态成员的指针

指针

- 通过指向成员的指针只能访问公有成员
- 声明指向成员的指针
  - 声明指向公有数据成员的指针

    类型说明符　类名::*指针名;
  - 声明指向公有函数成员的指针

    类型说明符　(类名::*指针名)(参数表);

# 指向类的非静态成员的指针

指针

- 指向数据成员的指针
  - 说明指针应该指向哪个成员

    指针名 = &类名::数据成员名;
  - 通过对象名（或对象指针）与成员指针结合来访问数据成员

    对象名.*类成员指针名

    或：

    对象指针名->*类成员指针名

# 指向类的非静态成员的指针

指针

- 指向函数成员的指针
  - 初始化

    指针名=&类名::函数成员名;
  - 通过对象名（或对象指针）与成员指针结合来访问函数成员

    (对象名.*类成员指针名)(参数表)

    或:

    (对象指针名->*类成员指针名)(参数表)

和指向普通函数的指针不同，’&’和’*’不能省略

# 指向类的非静态成员的指针

指针

例6-13 访问对象的公有成员函数的不同方式

```
int main() {      //主函数
  Point a(4,5);  //声明对象A
  Point *p1 = &a; //声明对象指针并初始化
   //声明成员函数指针并初始化
  int (Point::*funcPtr)() const = &Point::getX;
   //(1)使用成员函数指针访问成员函数
  cout << (a.*funcPtr)() << endl;
   //(2)使用成员函数指针和对象指针访问成员函数
  cout << (p1->*funcPtr)() << endl;
   //(3)使用对象名访问成员函数
  cout << a.getX() << endl;
   //(4)使用对象指针访问成员函数
  cout << p1->getX() << endl;

  return 0;
}
```

# 指向类的静态成员的指针

指针

- 对类的静态成员的访问不依赖于对象
- 可以用普通的指针来指向和访问静态成员
- 例6-14
  - 通过指针访问类的静态数据成员
- 例6-15
  - 通过指针访问类的静态函数成员

# 例6-14通过指针访问类的静态数据成员

指针

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

class Point {       //Point类定义
public:     //外部接口
   Point(int x = 0, int y = 0) : x(x), y(y) {
       count++;
   }
   Point(const Point &p) : x(p.x), y(p.y) {
       count++;
   }
   ~Point() {  count--; }
   int getX() const { return x; }
   int getY() const { return y; }
   static int count;
private:    //私有数据成员
   int x, y;
};
int Point::count = 0;
```

```
int main() {      //主函数实现
  //定义一个int型指针，指向类的静态成员
   int *ptr = &Point::count;

   Point a(4, 5); //定义对象a
   cout << "Point A: " << a.getX() << ", " <<
   a.getY();
   cout << " Object count = " << *ptr << endl;

   Point b(a);     //定义对象b
   cout << "Point B: " << b.getX() << ", " <<
   b.getY();
   cout << " Object count = " << *ptr << endl;

   return 0;
}
```

# 例6-15通过指针访问类的静态函数成员

指

针

```
#include <iostream>
using namespace std;
class Point {    //Point类定义
public: //外部接口
    Point(int x = 0, int y = 0) : x(x), y(y) { count++; }
    Point(const Point &p) : x(p.x), y(p.y) { count++; }
    ~Point() {  count--; }
    int getX() const { return x; }
    int getY() const { return y; }
    static void showCount() {
      cout << "  Object count = " << count << endl;
    }
private://私有数据成员
    int x, y;
    static int count;
};
int Point::count = 0;
```

```
int main() { //主函数实现
    //定义一个指向函数的指针，指向类的静态成员函数
    void (*funcPtr)() = Point::showCount;

    Point a(4, 5);    //定义对象A
    cout << "Point A: " << a.getX() << ", " << a.getY();
    funcPtr();//输出对象个数，直接通过指针访问静态函数成员

    Point b(a);       //定义对象B
    cout << "Point B: " << b.getX() << ", " << b.getY();
    funcPtr();//输出对象个数，直接通过指针访问静态函数成员

    return 0;
}
```

# 动态申请内存操作符 new

动态存储分配

new 类型名T（初始化参数列表）

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，并依初值列表赋以初值。

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。

# 基本类型变量初始化

动态存储分配

分配内存后，用初值来初始化：

```
int *point = new int(2);
```

不初始化：

```
int *point = new int;
```

用0初始化：

```
int *point = new int();
```

# 类对象初始化

动态存储分配

分配内存后，调用构造函数初始化：

T *point = new T(初值列表);

有用户定义的默认构造函数时，new T和new T()效果相同，都调用默认构造函数

若用户未定义默认构造函数：

new T 调用系统生成的默认构造函数

new T() 执行默认构造函数，并为基本数据类型和指针类型的成员用0赋初值，该过程是递归的

# 释放内存操作符delete

动态存储分配

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。

如果被删除的是对象，该对象的析构函数将被调用。

delete零指针是安全的：

```
int *ip = 0;
delete ip;
```

最好在delete指针后，把其值置为0

# 例6-16 动态创建对象举例

动态存储分配

```
#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;
};
```

```
int main() {
  cout << "Step one: " << endl;
  Point *ptr1 = new Point;//调用缺省构造函数
  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.
```

# 申请和释放动态数组

- 分配：new 类型名T［ 数组长度 ］
  - 数组长度可以是任何正整数值表达式，在运行时计算
  - 方括号后可加小括号”()”，但小括号内不能带任何参数。是否加”()”的区别同”new T”和”new T()”初始化时的区别

```
int *p = new int[10]();
```

- 释放：delete[] 指针名p
  - 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6-17动态创建对象数组举例

动态存储分配

```
#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.
```

# 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
};
```

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

运行结果如下：

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

# 动态创建多维数组

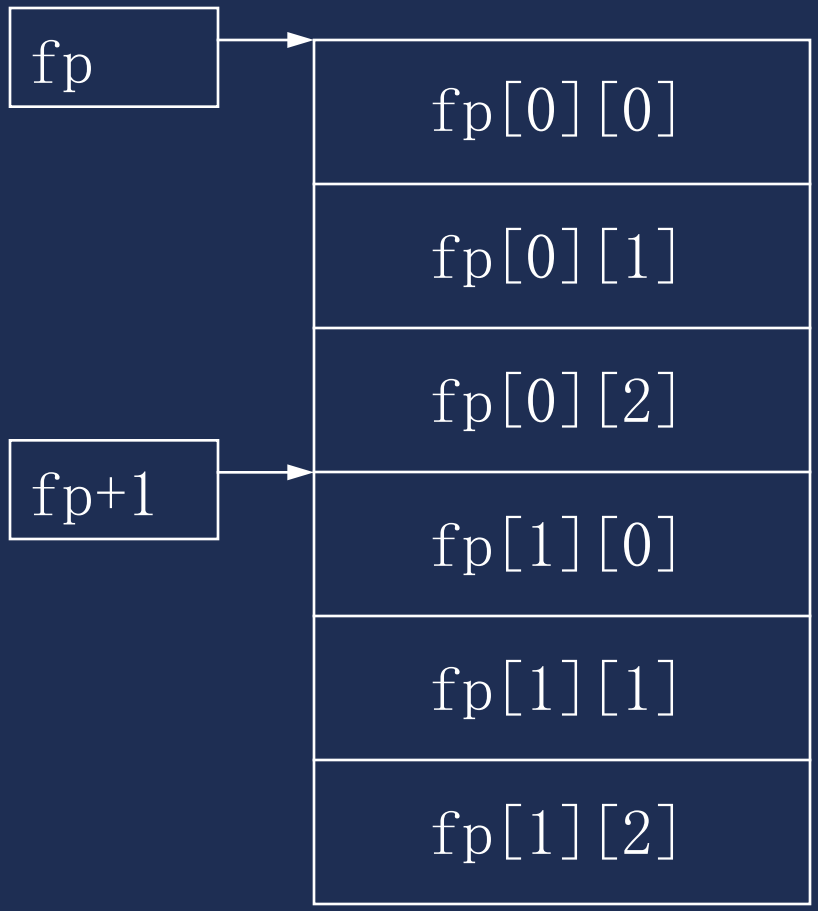
动态存储分配

new 类型名T[第1维长度][第2维长度]…;

如果内存申请成功，new运算返回一个指向新分配内存首地址的指针，是一个T类型的数组，数组元素的个数为除最左边一维外各维下标表达式的乘积。例如：

```
char (*fp)[3];
fp = new char[2][3];
```

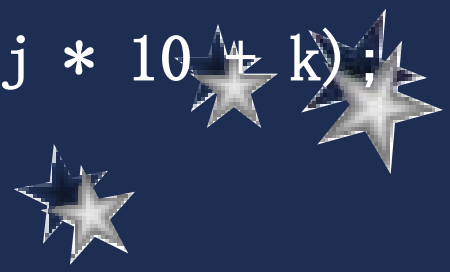
```
char (*fp)[3];
```



# 例6-19动态创建多维数组

动态存储分配

```
#include <iostream>
using namespace std;
int main() {
  float (*cp)[9][8] = new float[8][9][8];
  for (int i = 0; i < 8; i++)
    for (int j = 0; j < 9; j++)
      for (int k = 0; k < 8; k++)
        //以指针形式数组元素
        *(*(*(cp + i) + j) + k) =
        static_cast<float>(i * 100 + j * 10 + k);
```



```
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 9; j++) {
            for (int k = 0; k < 8; k++)
            //将指针cp作为数组名使用，通过数组名和
    下标访问数组元素
                cout << cp[i][j][k] << "  ";
            cout << endl;
        }
        cout << endl;
    }
    delete[] cp;
    return 0;
}
```

# 用vector创建动态数组

- 为什么需要vector？
  - 将动态数组封装，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动态数组对象的定义
  - 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数组对象的使用

- 对数组元素的引用
  - 与普通数组具有相同形式：
    - 数组对象名 [ 下标表达式 ]
  - 但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();
}
```

```
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;
}
```

# 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。
  - 隐含的拷贝构造函数完成浅拷贝
- 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6-21对象的浅拷贝

浅拷贝与深拷贝

```
#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
    //……
};
```

```
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;
```

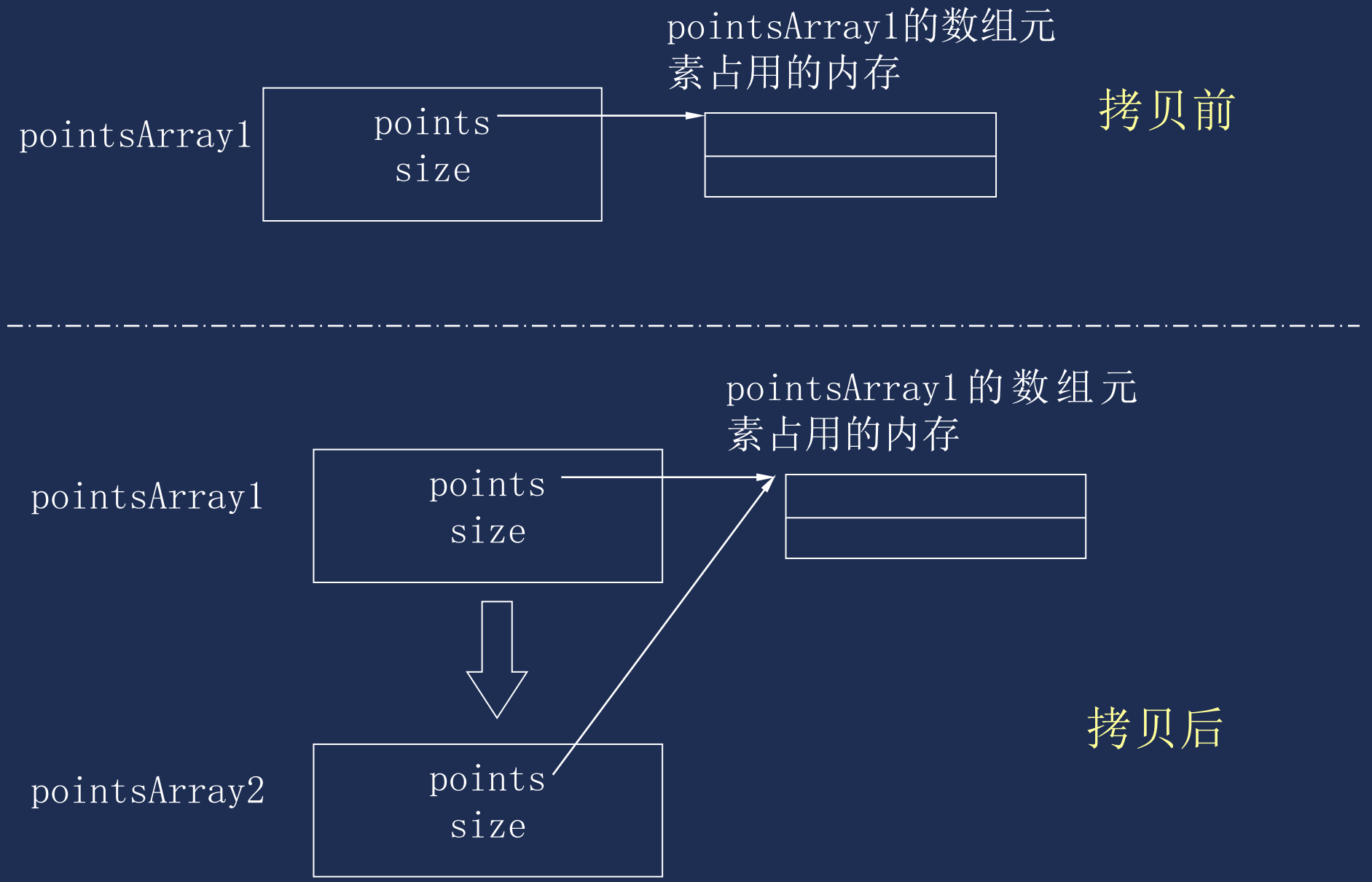
```
    pointsArray1.element(0).move(25, 30);
    pointsArray1.element(1).move(35, 40);
    cout << "After the moving of pointsArray1:" <<
    endl;
    cout << "Point_0 of array2: " <<
    pointsArray2.element(0).getX() << ", "
        << pointsArray2.element(0).getY() << endl;
    cout << "Point_1 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6-22对象的深拷贝

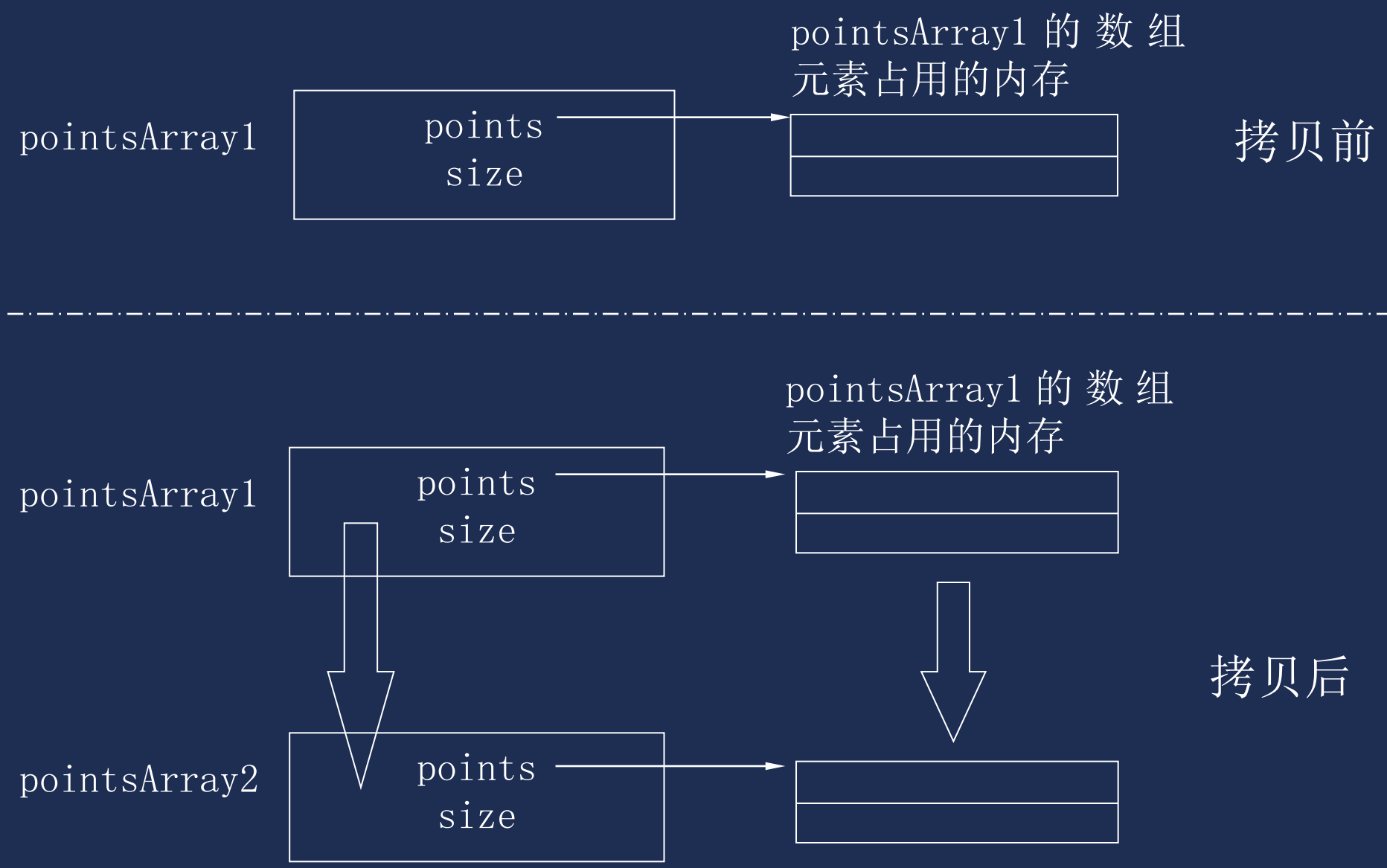
浅拷贝与深拷贝

```
#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
}
```

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

# 用字符数组存储和处理字符串

字符串

- 字符串常量（例："program"）
  - 各字符连续、顺序存放，每个字符占一个字节，以'\0'结尾，相当于一个隐含创建的字符常量数组
  - "program"出现在表达式中，表示这一char数组的首地址
  - 首地址可以赋给char常量指针：
    - `const char *STRING1 = "program";`
- 字符串变量
  - 可以显式创建字符数组来表示字符串变量，例如，以下三条语句具有等价的作用：

```
char str[8] = { 'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0' };
char str[8] = "program";
char str[] = "program";
```

| p | r | o | g | r | a | m | \0 |
|---|---|---|---|---|---|---|---|

# 用字符数组表示字符串的缺点

字符串

- 用字符数组表示字符串的缺点
  - 执行连接、拷贝、比较等操作，都需要显式调用库函数，很麻烦
  - 当字符串长度很不确定时，需要用new动态创建字符数组，最后要用delete释放，很繁琐
  - 字符串实际长度大于为它分配的空间时，会产生数组下标越界的错误
- 解决方法
  - 使用字符串类string表示字符串
  - string实际上是对字符数组操作的封装

# string的用法(1)

字符串

- 常用构造函数
  - string(); //缺省构造函数，建立一个长度为0的串
  - string(const char *s); //用指针s所指向的字符串常量初始化string类的对象
  - string(const string& rhs); //拷贝构造函数
- 例：
  - string s1; //建立一个空字符串
  - string s2 = “abc”; //用常量建立一个初值为”abc”的字符串
  - string s3 = s2; //执行拷贝构造函数，用s2的值作为s3的初值
- 返回串的长度：
  - unsigned int length() const;

# string的用法(2)

字符串

- 常用操作符
  - s + t　　将串s和t连接成一个新串
  - s = t　　用t更新s
  - s == t　　判断s与t是否相等
  - s != t　　判断s与t是否不等
  - s < t　　判断s是否小于t（按字典顺序比较）
  - s <= t　　判断s是否小于或等于t （按字典顺序比较）
  - s > t　　判断s是否大于t （按字典顺序比较）
  - s >= t　　判断s是否大于或等于t （按字典顺序比较）
  - s[i]　　访问串中下标为i的字符
- 例：
  - string s1 = “abc”, s2 = “def”;
  - string s3 = s1 + s2; //结果是”abcdef”
  - bool s4 = (s1 < s2); //结果是true
  - char s5 = s2[1];　　//结果是’e’

# 例6.23 string类应用举例

字符串

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

//根据value的值输出true或false，title为提示文字
inline void test(const char *title, bool value) {
    cout << title << " returns " << (value ? "true" : "false") << endl;
}
```

```
int main() {
  string s1 = "DEF";
  cout << "s1 is " << s1 << endl;
  string s2;
  cout << "Please enter s2: ";
  cin >> s2;
  cout << "length of s2: " << s2.length() <<
  endl;

  //比较运算符的测试
  test("s1 <= \"ABC\"", s1 <= "ABC");
  test("\"DEF\" <= s1", "DEF" <= s1);
  //连接运算符的测试
  s2 += s1;
  cout << "s2 = s2 + s1: " << s2 << endl;
  cout << "length of s2: " << s2.length() <<
  endl;
  return 0;
}
```

# 用getline输入整行字符串



- 输入整行字符串
  - 用cin的>>操作符输入字符串，会以空格作为分隔符，空格后的内容会在下一回输入时被读取
  - 用string头文件中的getline可以输入整行字符串，例如：
    - `getline(cin, s2);`
- 以其它字符作为分隔符输入字符串
  - 输入字符串时，可以使用其它分隔符作为字符串结束的标志（例如逗号、分号）
  - 把分隔符作为getline的第3个参数即可，例如：
    - `getline(cin, s2, ',');`

# 例6.24 用getline输入字符串

字符串

```
include <iostream>
#include <string>
using namespace std;
int main() {
  for (int i = 0; i < 2; i++)  {
     string city, state;
     getline(cin, city, ',');
     getline(cin, state);
     cout << "City:" << city << "  State:" << state << endl;
  }
  return 0;
}
```

```
Beijing,China
City: Beijing   State: China
San Francisco,the United States
City: San Francisco   State: the United States
```

# 指针与引用

深度探索

- 引用本身占用的内存空间中，存储的就是被引用变量的地址
- 引用的功能相当于指针常量
  - 普通指针可以多次被赋值
  - 引用只能在初始化时指定被引用的对象
- 只有常引用，没有引用常量
  - 不能用T & const作为引用类型

# 指针常量与引用的比较

深度探索

| 操作 | T类型的指针常量 | 对T类型的引用 |
| --- | --- | --- |
| 定义并用v初始化 | T * const p = &v; | T &r = v; |
| 取v的值 | *p | r |
| 访问成员m | p->m | r.m |
| 读取v的地址 | p | &r |

- p可以再被取地址，而&r则不行
- 引用本身的地址是不可以获得的
- 引用实现的功能，用指针都可以实现

# 指针与引用的对应关系

深度探索

```
//使用指针常量
void swap(int * const pa,
    int * const pb) {
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
int main() {
    int a, b;
    ……
    swap(&a, &b);
    ……
    return 0;
}
```

```
//使用引用
void swap(int &ra, int &rb) {
    int temp = ra;
    ra = rb;
    rb = temp;
}
int main() {
    int a, b;
    ……
    swap(a, b);
    ……
    return 0;
}
```

# 指针与引用的联系

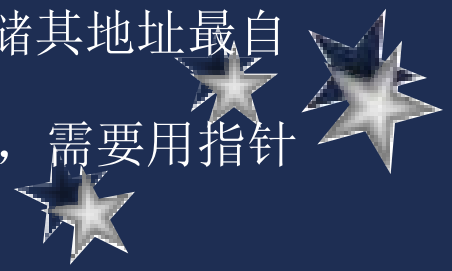
深度探索

- 引用在底层通过指针来实现
  - 一个引用变量，通过存储被引用对象的地址，来标识它所引用的对象
- 引用是对指针的包装，比指针更高级
  - 指针是C语言就有的底层概念，使用起来很灵活，但用不好容易出错
  - 引用隐藏了指针的“地址”概念，不能直接对地址操作，比指针更安全

# 引用与指针的选择

深度探索

- 什么时候用引用？
  - 如无需直接对地址进行操作，指针一般都可用引用代替
  - 用更多的引用代替指针，更简洁、安全
- 什么时候用指针？
  - 引用的功能没有指针强大，有时不得不用指针：
    - 引用一经初始化，无法更改被引用对象，如有这种需求，必须用指针；
    - 没有空引用，但有空指针，如果空指针有存在的必要，必须用指针；
    - 函数指针；
    - 用new动态创建的对象或数组，用指针存储其地址最自然；
    - 函数调用时，以数组形式传递大量数据时，需要用指针作为参数。



# 指针的地址安全性问题

深度探索

- 地址安全性问题
  - 通过指针，访问了不该访问的地址，就会出问题
  - 典型问题：数组下标越界、多次delete
  - 问题的严重性：有时会在不知不觉中产生错误，错误源很难定位，因此程序调试起来很麻烦
- 解决方案
  - 指针只有赋了初值才能使用（这一点普通变量也应遵循）
  - 指针的算术运算，一定要限制在通过指向数组中某个元素的指针，得到指向同一个数组中另一个元素的指针
  - 尽量使用封装的数组（如vector），而不直接对指针进行操作

# 指针的类型安全性问题(1)

深度探索

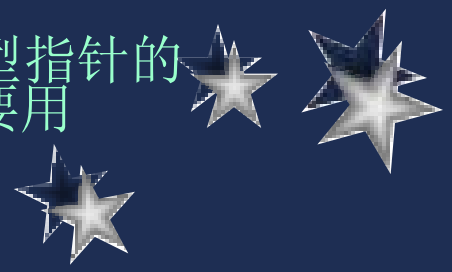
- 基本类型数据的转换是基于内容的：
  - static_cast：比较安全的，基于内容的类型转换
  - 例：

```
int i = 2;
float x = static_cast<float>(i);
```

- 目标类型不同的指针间的转换，会使一种类型数据的二进制序列被当作另一种类型的数据：
  - reinterpret_cast：可以将一种类型的指针转换为另一种类型

```
int i = 2;
float *p = reinterpret_cast<float *>(&i);
```

- 结论
  - 从一种具体类型指针转换为另一种具体类型指针的reinterpret_cast很不安全，一般情况下不要用



# 指针的类型安全性问题(2)

深度探索

- void指针与具体类型指针的转换：
  - 具体类型指针可以隐含地转换为void指针：

```
int i = 2;
void *vp = &i;
```

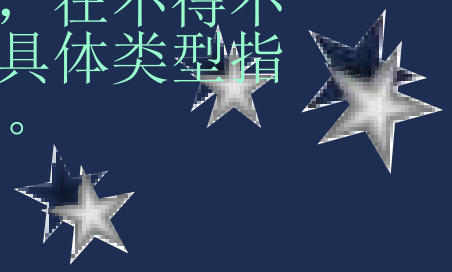
  - void指针转换为具体类型指针需要用static_cast，例如：

```
int *p = static_cast<int *>(vp);
```

  - 但这样就出问题了：

```
float *p2 = static_cast<float *>(vp);
```

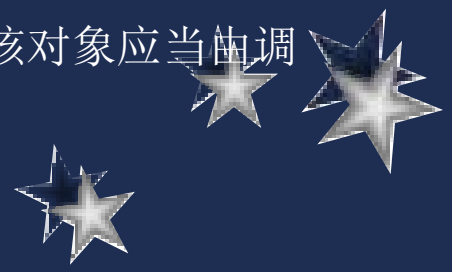
- 结论
  - void指针也容易带来不安全，尽量不用，在不得不用的时候，一定要在将void指针转换为具体类型指针时，确保指针被转换为它最初的类型。



# 堆对象的管理

深度探索

- 堆对象必须用delete删除
  - 避免“内存泄漏”
  - 原则很简单，但在复杂的程序中，一个堆对象常常要被多个不同的类、模块访问，该在哪里删除，常常引起混乱
- 如何有条理地管理堆对象
  - 明确每个堆对象的归属
  - 最理想的情况：在一个类的成员函数中创建的堆对象，也在这个类的成员函数中删除
    - 把对象的建立和删除变成了一个类的局部问题
  - 如需在不同类之间转移堆对象的归属，一定要在注释中注明，作为类的对外接口约定
    - 例如在一个函数内创建堆对象并将其返回，则该对象应当由调用该函数的类负责删除



# const_cast介绍

深度探索

- const_cast的用法
  - 用于将常指针、常引用转换为不带“const”的相关类型
  - 例：

```
void foo(const int *cp) {
  int *p = const_cast<int *>(cp);
  (*p)++;
}
```

  - 这可以通过常指针修改指针发对象的值，但这是不安全的用法，因为破坏了接口中const的约定
- const_cast不是安全的转换，但有时可以安全地使用

# const_cast的安全使用(1)

深度探索

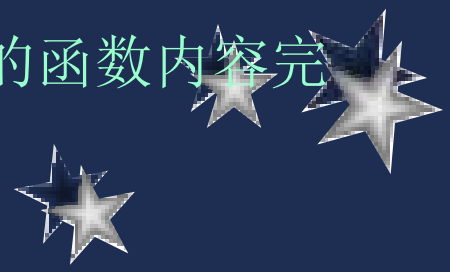
- 对例6-18的改进
  - 例6-18通过下列函数访问数组元素

```
Point &element(int index);
```

  - 问题：由于不是常成员函数，无法使用常对象访问数组元素
  - 解决方法：重载element函数：

```
const Point &element(int index) const {
  assert(index >= 0 && index < size);
  return points[index];
}
```

- 新问题
  - 代码的重复：这个element函数与原先的函数内容完全相同，两份重复的代码，不易维护



# const_cast的安全使用(2)

深度探索

- 新问题的解决
  - 修改原先的element函数：

```
Point &element(int index) {
  return const_cast<Point &>(
    static_cast<const ArrayOfPoints *>(this)
      ->element(index));
}
```

  - 执行过程：调用常成员函数element()，再将其返回结果中的const用const_cast去除
  - 将this用static_cast转换为常指针，是安全的转换
  - 该函数本身不是常成员函数，确保将最终的结果以普通引用形式返回是安全的
- 思考：如果保留该函数，而修改常成员函数element，使常成员函数element调用该函数，是否合适？

# 小结与复习建议

- 主要内容
  - 数组、指针、动态存储分配、指针与数组、指针与函数、字符串
- 达到的目标
  - 理解数组、指针的概念，掌握定义和使用方法，掌握动态存储分配技术，会使用string类。