对象数组、指针

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

数组的概念

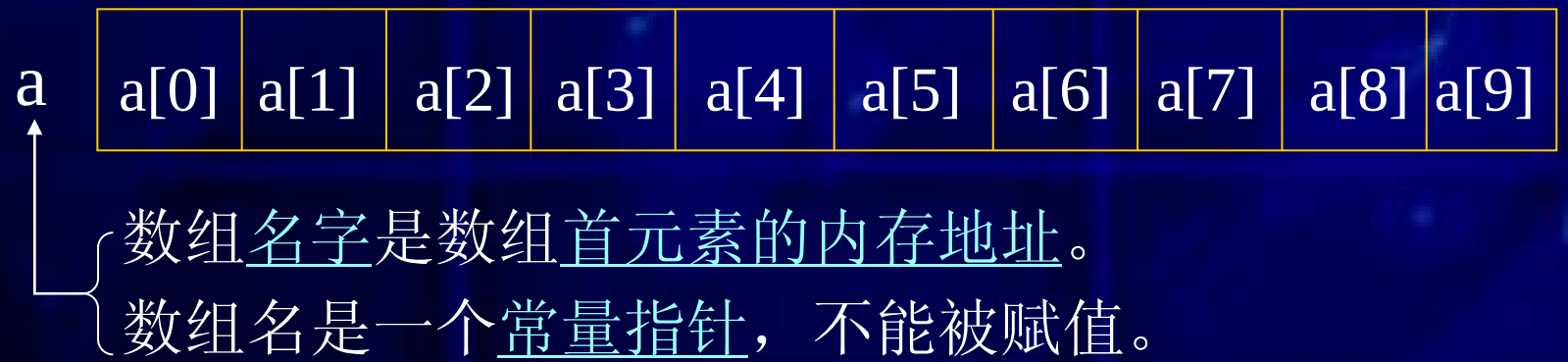
- 数组是具有一定顺序关系的若干相同类型变量的集合体，组成数组的变量称为该数组的元素。
- 数组属于构造类型。
- 一维数组的声明

  类型说明符　数组名[常量表达式];
- 引用
  - 必须先声明，后使用。
  - 只能逐个引用数组元素，而不能一次引用整个数组

    数组名 [整型表带式]

一维数组的存储顺序

- 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。

数组名是一个常量指针，不能被赋值。

一维数组的初始化

可以在编译阶段使数组得到初值：

- 在声明数组时对数组元素赋以初值。

  例如： static 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： static 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： static int a[]={1,2,3,4,5}

- 注意： 不存在与数组初始化相对应的数组赋值

```
char v[3];
V={'c','a','d'}// error
```

多维数组的声明及引用

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，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

- **二维数组的声明**

  类型说明符 数组名[常量表达式][常量表达式];

  例如：float a[3][4];　　由 3 个一维数组构成

  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}

- **引用**

  例如：b[1][2]=a[2][3]

二维数组的初始化

- 将所有数据写在一个{}内，按顺序赋值

```
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

```
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

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

数组作为函数参数

- 数组元素作实参，与单个变量做参数一样。
- 数组名作参数，实际上是指针做参数
  - 形、实参数都是数组名，类型要一样，传送的是数组首地址
  - 形参是指针，实参是数组名，类型相同，传送的也是数组首地址
  - 对形参数组的改变会直接影响到实参数组

对象数组

- 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

对象数组初始化与删除

- 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。

```
Point A[2]={Point(1,2), Point(3,4)};  // 调用 2 次构造函数
```

- 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（调用默认构造函数）

```
Point A[2];  // 这时 Point 类必须定义了默认构造函数
```

- 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

指针变量

- 数据存贮在内存空间中，对数据的访问方式：
  - 通过变量名访问（直接访问）
    - 数据存贮的地址由编译程序分配，固定的
    - 变量标识符与其地址对应
  - 通过地址访问（间接访问）
    - 指针变量，用于记录要访问的数据的存贮地址
    - 访问的数据存贮地址由程序指定，可变
- 地址运算符：&表达式
  - 取表达式的内存地址，表达式必须是一个左值

&var　　取变量 var 的地址

&(Object.ab)　取成员变量的地址

&(a[8])　　取数组元素的地址

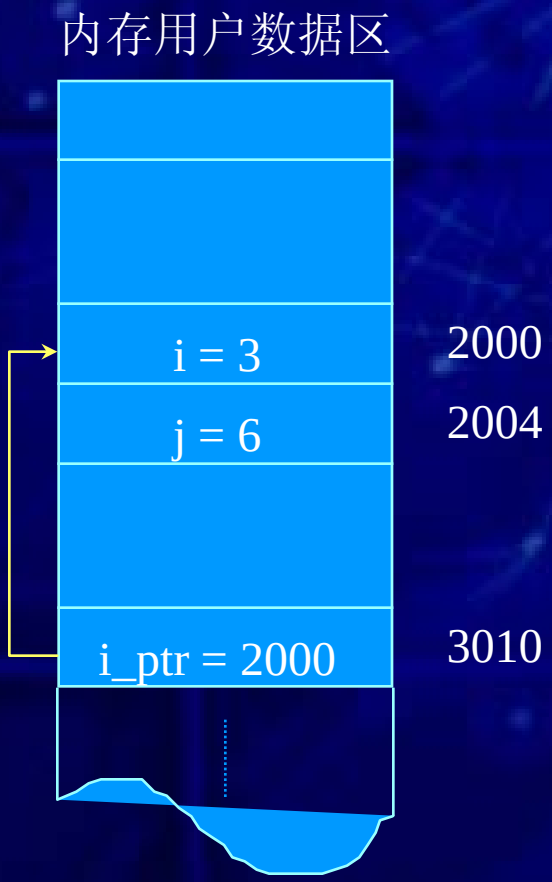
指针变量

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

指针变量是有类型的



声明

例：
```
int i;
int *i_ptr = &i;
```

指向整型变量的指针

引用

```
i = 3;
*i_ptr = 3;
```

void 类型指针

```
void *pv;      // 用于记录一个地址，类型未知（不关心）
int i;
pv = &i;       // 记录 i 的地址
*pv = 0;       // error
pv = pv + 1;   // error

int *pint = (int*)pv;
// void指针的值（地址）可以赋值给任何类型的指针变量
// 但需要类型强制转换
```

指向常量的指针

```
const   Type   * ptr;
```

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

```
const  char *name1 = “John’;  //指向常量的指针
char s[]="abc";
name1   = s;          //正确，name1本身的值可以改变
*name1  =‘1’;         //错误，试图修改常量
name1[2]=‘3’;         //错误，试图修改常量
```

指针类型的常量

Type * const ptr;

- 声明指针是常量，指针的值（地址值）不能被改变。

例：

```
char  str_1[] = “abcdef”;
char str_2[] = “ghijkl”;
char  * const  s = str_1;
s[1] = ‘A’;
s = str_2;                    //错误，指针常量值不能改变

const char * const cp = str_1;  // 指针和指向对象都是常量
```

指针变量的算术运算

- 指针与整数的加减运算
  - 指针 p 加上或减去 n，等于是指针当前指向位置的前方或后方第 n 个数据对象的地址。
  - 这种运算的结果值取决于指针指向对象的数据类型
- 指针加一，减一运算
  - 指向后一个或前一个数据对象的地址

```
px++        px--
```

注：* 和 ++、-- 优先级相同，自右向左运算

```
*px++  ==  *(px++)
```

指针变量的关系运算

- 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。

数组与指针

- 数组名是指向数组第一个数组元素的常量指针
- 类型是数组元素的类型

```
int a[10];
int  *pa = a;                  // 或 pa = &a  或  pa=&a[0]
```

通过指针或数组名引用数组元素是等效的

```
a[0] , pa[0] , *pa, *a    引用的是同一数组元素
a[m],  pa[m],  *(pa+m),  *(a+m) 引用的是同一数组元素
...
```

不能写 a++, 因为a是数组首地址是常量。

指针数组

- 数组的元素是指针型

```
Point *pa[2]; //由两个指向Point类型的指针组成的数组
```

- 多维数组

```
int pi[2][10];        // 由两个一维数组构成
```

pi[0]、pi[1] 是一个指向一维数组首地址指针（常量）

```
pi[0] == &(pi[0][0]);
pi[1] == &(pi[1][0]);
```

指向函数的指针

- 声明形式

  返回数据类型 (*函数指针名)(参数表);

- 调用

  函数指针名(参数表)

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
- 每一个函数名是一个常量函数指针

例：函数指针

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

void print_float(float data_to_print);
void (*function_pointer)(float);

int main()
{
   float pi = (float)3.14159;
   function_pointer = print_float;
   function_pointer(pi);  // 等于 print_float(pi)
      ...
}
```

对象指针

- 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

  - 例

```
ptr->getx() ;   // 相当于 (*ptr).getx();
```

this指针

- 隐含于每一个类的成员函数中的特殊指针。
- 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，该对象的地址赋给 this 指针，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类的构造函数体中的语句：

    X = xx　　相当于　　this->X = xx

  - 调用对象的成员函数（除了静态成员函数外）时，都隐含地将 this 对象指针作为成员函数的一个参数

指向类成员（非静态）的指针

- 通过指向成员的指针来访问类的公有成员
- 指向成员变量的指针
  - 声明　　类型说明符　类名::*指针名;
  - 赋值/初始化　　指针名 = &类名::数据成员名;
  - 访问数据成员　　对象名.* 类成员指针名
    对象指针名->*类成员指针名
- 指向成员函数的指针
  - 声明　　类型说明符　(类名::*指针名)(参数表);
  - 赋值/初始化　　指针名 = 类名::函数成员名;
  - 访问成员函数　　(对象名.* 类成员指针名)(参数表)
    (对象指针名->*类成员指针名)(参数表)

指向类成员（非静态）的指针

```
class CA
{
public:
   int x;
   int y;

   void f(int i);
   void g(int i);
   ...
};
void CA::f(int i)
{
    x = i;
}
void CA::g(int i)
{
    y = i;
}


int main()
{
   int   CA::* pData       = &CA::x;
   void (CA::*pFunc)(int)  = CA::f;

   CA a, b;

   a.*pData = 1;
   b.*pData = 2;
   cout << a.x << " " << b.x << endl;

   pData = &CA::y;
   a.*pData = 3;
   b.*pData = 4;
   cout << a.y << " " << b.y << endl;

   (a.*pFunc)(5);
   (b.*pFunc)(6);
   cout << a.x << " " << b.x << endl;

   pFunc = CA::g;
   (a.*pFunc)(7);
   (b.*pFunc)(8);
   cout << a.y << " " << b.y << endl;
     ...
}
```

指向类的静态成员的指针

- 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

```
class Point
{
  public:
    static int GetC();
    // ... 其他外部接口

    static int countP; //静态数据说明
    // ... 其他公共数据

  private:  //私有成员
    ...
};

static int GetC()
{
   return countP;
}
//...其他外部接口实现

int Point::countP = 0; //静态数据定义
```

```
int main()
{
   int *count = &Point::countP;
   (*count) ++;
   cout << *count <<endl;

   int (*pGetC)() = Point::GetC;
   cout << pGetC() <<endl;

   Point A(3,4);
   cout << pGetC() <<endl;

   ...
}
```