程序设计及算法语言

第五章 数组与指针

戚隆宁
Email: longn_qi@seu.edu.cn
Tel: 13813839703

本章目录

5.1 数组

- 应用背景
  - 处理大量同类数据的需要
    - 计算结果为大量同类数据：分类/分布统计、字符串
    - 输入数据长时间依赖：数据匹配、排列组合
- 概念
  - 复合类型（compound type），描述“同类型对象的有限序列”

数组的相关概念

- 数组元素（array element）
  - 构成数组的对象（子对象）
  - 存储地址连续
- 数组下标（array subscript）

  - 数组元素的访问索引（位置标识），整数
  - 连续有界，下界从**0**开始
    - 例：数组第4个元素的下标为3
  - 上界为数组元素个数减1

数组的定义与声明

- 定义语法

[修饰符] <数组元素类型> <数组名> <数组元素个数> [初值表];

```
static long data [10];
double x[5], y[6];
char table [3] = { ‘a’ ,’ b’ ,’ c’ };
```

数组的定义与声明

- 定义语法要求
  - 数组名：符合一般变量名规范
  - 数组元素类型：支持所有确定大小的类型（基本类型、结构体类型、类类型、数组类型等）
  - 数组元素个数：正整数，常数或常变量（编译时必须确定大小）
    - 有初值表时可省略，元素个数根据初值表确定。

```
long data [] = { 1, 2, 3 };
```

数组的定义与声明

- 初值表（数组的初始化）
  - 完全初值

```
char table [3] = {'a','b','c'};
```

  - 部分初值

其余初值为默认值0

```
long data [10] = { 1, 2, 3 };
```

  - 越界检查

```
long data [2] = { 1, 2, 3 }; × 越界！
```

数组的定义与声明

- 外部声明语法

<外部声明修饰符><数组元素类型> <数组名> <元素个数>;

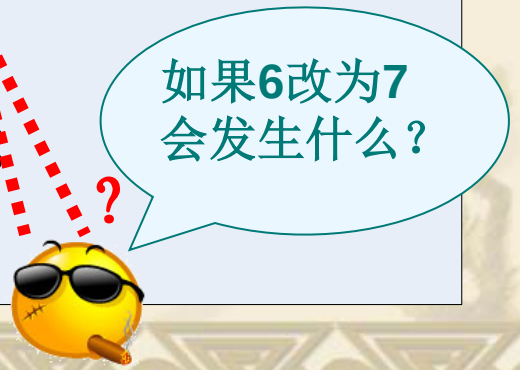
  - 允许声明时数组元素个数不确定

```
extern long data [10];
extern char table [];
```

数组的操作

- 访问数组元素
  - 按下标访问
  - 无越界检查

```
int i ;
int data [6] = { 1 };
data[1] =  2;
data[2] = data[0] + data[1];
for( i = 3; i < 6; i++ )
{
        data[i] = 1;
}
```



数组的操作

- 除sizeof运算外，不支持数组整体访问
  - 数组名表示数组首地址（第一个元素所在位置），而非数组整体。支持比较、偏移和输出

```
int data [6] = { 0 };
int copy [6] = data;
data = 1;
copy = data;
```

以上除第一句外全部错误!
以下全部正确

```
int size = sizeof( data );
if ( copy < data )
{
    int offset = copy - data;
}
cout << data << endl;
```

数组的操作

- 作为函数参数传递
  - 传递数组首地址
  - 被调函数无法直接判断数组大小

```
void func( int data [6] )
{
        int size = sizeof( data );
}

void main( )
{
        int  data  [3] = { 0 };
        func( data );
        int size = sizeof( data );
}
```

5.2 字符数组

- 字符串常量
  - 字符串标识符：“ ”
  - 连续字符
  - 存储结束标志：‘\0’（ASCII码值0）
  - 只读

”Hello”

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’

仅有结束标志的是空字符串！

字符数组

- 字符数组的定义和初始化

```
char str [] = “Hello”;
```

```
char str [6] = { ‘H’, ’e’, ’l’, ’l’, ’o’, ’\0’ };
```

- 字符数组的运算

```
cout << str [1];
cout << str;
cout << sizeof(str);
```

字符数组的库函数

- C字符串库函数<cstring>或<string.h>
  - 复制：strcpy, strdup
  - 比较：strcmp
  - 连接：strcat
  - 转换：strlwr, strupr, strrev
  - 查找：strchr, strspn, strcspn, strstr, strtok
  - 统计：strlen（字符串长度）

字符数组的输入输出

- 字符数组的输入输出（<iostream>）
  - 输入：cin、get(char *, int, char='\n')、getline(char *, int, char='\n')
  - 输出：cout

```
char str [6];
cin >> str; // 以空白字符作为分隔符
cout << str;
cin.getline( str, 5, '\t'); // 以指定字符（默认为换行符）作为分隔符（分隔符读取但不存储）
cin.get( str, 5, 'e'); // 以指定字符（默认为换行符）作为分隔符（分隔符不读取不存储）
```

多维数组

- 问题的引入
  - 多维度描述：多层次索引或分类
    - 2排3座
    - 3号楼1层2室
    - 中国——江苏省——南京市——玄武区
- 多维数组：数组元素类型是数组
  - 下标的层次数（数组的维数）
    - 多维数组的存储（映射到1维）：按高维（级别从左至右降低）顺序存储

多维数组的初始化

- 初值表
  - 线性初值表
  - 嵌套初值表

```
int data [2][2] = { 1 , 2, 3, 4 };
或  int data [2][2] = {{ 1 , 2 },
                       { 3, 4 }};
```

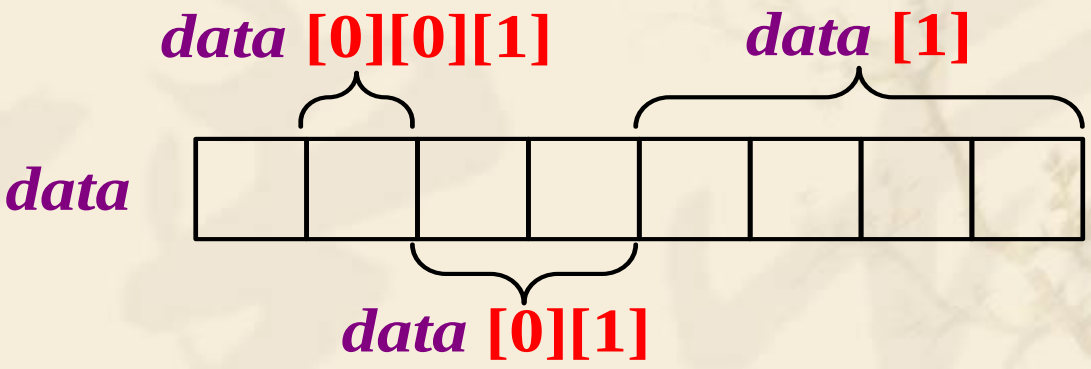
- 仅第一维大小可根据初值表大小确定

```
int data [][2] = {{ 1 }, { 3 }};
```

多维数组的操作

- sizeof运算
- 分层（分维）访问

```
int data [2][2][2];
```



多维数组的操作

- 作为函数参数传递
  - 除第一维外，其它维的边界必须参数匹配

```
void func( int data [][3] )
{
        int size = sizeof( data );
}

void main( )
{
        int  data  [3][3] = { 0 };
        func( data );
}
```

数组的基本特征

- 同类有序
- 分层有界

返回目录

5.3 数组的应用

- 统计
- 字符串处理
- 矩阵计算
- 排列组合（八皇后问题，回溯法）
- 排序、查找

排序与查找

- 查找（search）的概念
  - 在数据集合中寻找满足特征条件的数据对象
    - 查找源
    - 查找关键字
    - 查找结果
- 查找的分类
  - 数据集合的有序性：有序、无序
  - 特征条件的复杂性：单一条件、组合条件
  - 特征条件的匹配性：单对象、多对象

常用查找方法

- 基于有序表（顺序表）的查找
  - 顺序查找
  - 对半查找
  - 直接查找
  - 分类查找

顺序表

- 顺序表（sequential list）的概念
  - 数据结构特征
    - 元素线性排列（最多只有一个前驱和一个后继）
    - 通过下标直接随机访问
    - 访问元素的时间开销与表的长度无关
- 顺序表的实现方式
  - 数据表：数组（存放关键字或关键字索引）
  - 表头和表尾位置
  - 表长 = 表尾位置 - 表头位置 + 1

常用查找方法

- 基于有序表（顺序表）的查找
  - 顺序查找
  - 对半查找
  - 直接查找
  - 分类查找

常用查找方法

- 基于有序表（顺序表）的查找
  - 顺序查找
  - 对半查找
  - 直接查找
  - 分类查找

对半查找

对半查找

对半查找

```
int BinarySearch( int data[], int key, int low, int high)
{
    int mid = -1;
    if( low == high )  return data[low] == key ? low : mid;
    else if( low < high )
    {
        mid = (low+high)/2;
        if( data[mid] < key ) mid = BinarySearch( data, key, mid+1, high);
        else mid = BinarySearch( data, key, low, mid);
    }
    return mid;
}
```

对半查找

```
int BinarySearch( int data[], int key, int low, int high)
{
    int mid = -1;
    if( low <= high )
    {
        mid = (low+high)/2;
        if( data[mid] < key ) mid = BinarySearch( data, key, mid+1, high);
        else if( data[mid] > key ) mid = BinarySearch( data, key, low, mid-1);
    }
    return mid;
}
```

常用查找方法

- 基于有序表（顺序表）的查找
  - 顺序查找
  - 对半查找
  - 直接查找
    - 散列表（稀疏数据集合）
  - 分类查找

常用查找方法

- 基于有序表（顺序表）的查找
  - 顺序查找
  - 对半查找
  - 直接查找
  - 分类查找

5.4 指针与地址

- 地址的概念
  - 内存地址（线性空间）
    - 数据：变量地址（取地址运算&）
    - 代码：函数地址、文字常量地址
- 问题的引入：可变的地址引用
- 指针（pointer）的概念
  - 复合类型，描述“地址”
    - 指向性的地址
    - 类型化的地址

指针的定义与声明

- 定义语法

[修饰符] <指向类型> <指针标识符> [修饰符] <指针名> [初值表];

```
static char * table  = 0;
const char * str;
char * const s = table;
long * p, * q;
```

指针的定义与声明

- 定义语法要求
  - 指针名：符合一般变量名规范
  - 指向类型：任意已声明的类型
  - 初值（指针的初始化）
    - 已定义变量的地址
    - 已定义函数地址
    - 文字常量地址
    - 空地址：0
    - 已定义的指针

指针的初始化

```
long a;
long * p = &a;
long * q = p;
char * str = “Hi”;
char * table = 0;
```

- 指向变量地址，变量类型必须匹配
- 指向已定义指针，指针类型必须匹配
- 指向文字常量地址，必须是字符型指针
- 指向空地址（NULL），可以是任意指针

指针的初始化

- 字符指针

```
const char * p = “Hello”;

char str [] = “Hello”;
char * p = str;
```

注意空字符串与空指针的区别！

指针的初始化

- 函数指针
  - 参数列表必须匹配
  - 函数返回类型必须匹配

```
int func( int a, int b );
int ( * p )( int, int ) = func;
typedef int ( * FUNC )( int, int );
FUNC p = func;
```

指针的初始化

- 数组指针
  - 参见5.5节
- 多级指针
  - 参见5.6节
- 通过类型强制转换，可以指向任意地址

```
long a;
char * str = (char *)&a;
```

指针的定义与声明

- 外部声明语法

<外部声明修饰符> <指向类型> <指针标识符> <指针名>;

```
extern  char *  str;
```

指针的操作

- 赋值操作
- 间接引用操作(*)
  - 空指针不可间接引用

```
char a = ‘A’ , b;
char * p = &a, * q;
q = &b;
*q = ‘a’ ;
*p = *q + 1;
q = p;
*q = b + 1;
p = “Hi”;
*p = a;
p = 0;
```

指针的操作

- sizeof运算
- 比较运算
  - 指向类型必须相同
- 差值运算
  - 指向类型必须相同
- 偏移运算

```
long a = 1, b = 2;
long * p = &a, * q = &b;
int size = sizeof( p );
if ( p < q && p != 0 )
{
    int offset = q - p;
}
p = p + 2;
```

指针的操作

- 作为函数参数
- 作为函数返回
  - 不可返回指向局部变量的指针

```
int * func( int * p )
{
        int size = sizeof( p );
        * p = 3;
        return &size;
}
void main( )
{
        int  data = 0, * p;
        p = func( &data );
}
```

指针的操作

- 作为函数执行
  - 仅适用于函数指针

```
int add( int a, int b );
int mul( int a, int b );
int ( * p )( int, int ) ;
void main()
{
        int result;
        p = add;
        result = (* p)( 1, 2 );
        p = mul;
        result = (* p)( result, 4 );
}
```

指针的操作

- const修饰符的影响
  - 指针标识符分界
    - 之前，修饰指向类型，指向内容不可修改
    - 之后，修饰指针变量本身，指针本身不可修改
  - 指向类型兼容性（自动转换）
    - const修饰兼容无修饰，反之不可
    - 多用于函数参数传递

间接访问权限转换，从严不从宽！

const修饰符的影响

```
const char c = 'A';
char d = 'B';
const char *p = &c;
char * const q = &d;
char * s = "OK";
```

```
*p = 'C';
*q = 'D';
*s = 'P';
s = p;
p = q;
q = s;
s = q;
q = p;
p = s;
*s = 'E';
```

返回目录

5.5 数组与指针的关系

- 指针的下标操作

```
char * p = “Hello” , c;
c = p[4];     // 等同于 c = *(p + 4);
```

- 数组的间接访问操作

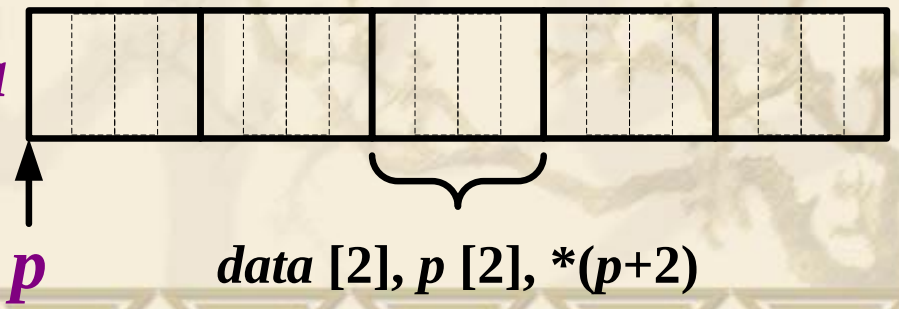
```
char data [] = “Hello” , c;
c = * ( data + 4 ); // 等同于 c = data[4];
```

数组名与指针

- 用指针访问数组
  - 数组名 ≈ 指向数组第一个元素的指针常变量
    - 除sizeof运算外，其他完全相同

```
        long data [5];
        long * const p = & data [0];
或      long * const p = data;
```

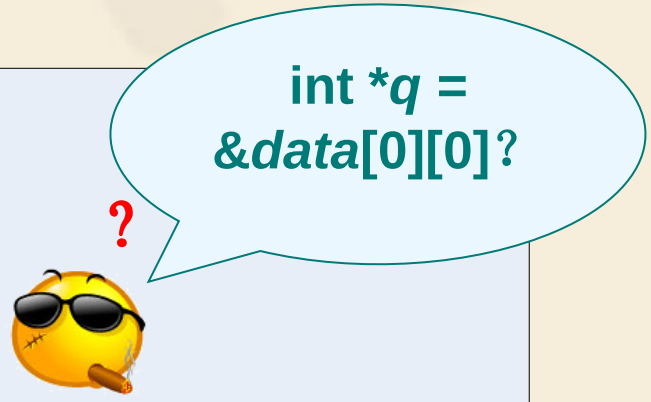
  - 访问任意数组元素
    - 下标、偏移



数组指针

- 用指针访问多维数组

```
int data [2][3];

int ( * q ) [3] = data;

typedef int ARRAY [3];
ARRAY * q = data;
typedef int ( * PARRAY ) [3];
PARRAY q = data;
```



指针数组

- 用数组存储指针

```
char * table [2]  = { “Yes”, “No” };

int add( int a, int b );
int mul( int a, int b );
int ( * func [2])( int, int ) = { add, mul };
```

指针与数组的复杂组合

- 判别原则
  - 优先级：括号 > 数组和函数 > 指针

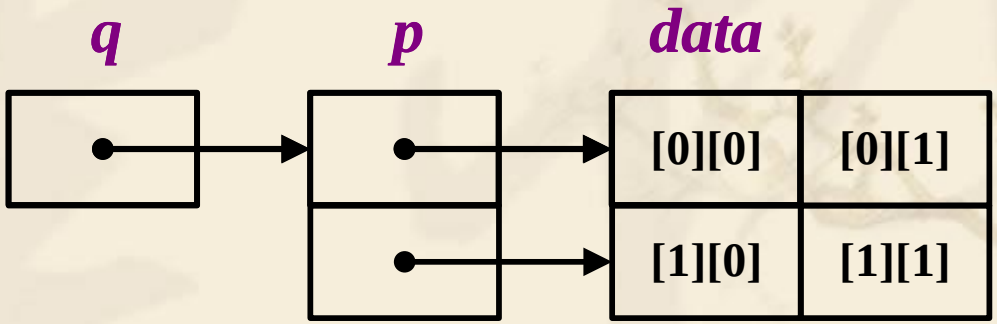
```
int * ( * ( * p )( int * ( * )( int * )))( int * );

int ( * ( * p [2] )( int ( * ) [3] ))[4];
```

返回目录

5.6 多维数组与多级指针

- 多级指针
  - 指向类型是指针的指针

```
long data [2][2];
long * p [2] = { data[0], data [1] };
long * * q = &p[0];
```



多维数组与多级指针

- 多级指针的操作

间接访问：	****q**
下标操作：	***q*[1][0]**
混合操作：	***(*q*[0] + 1)**
	(*(*q* + 1))[1]

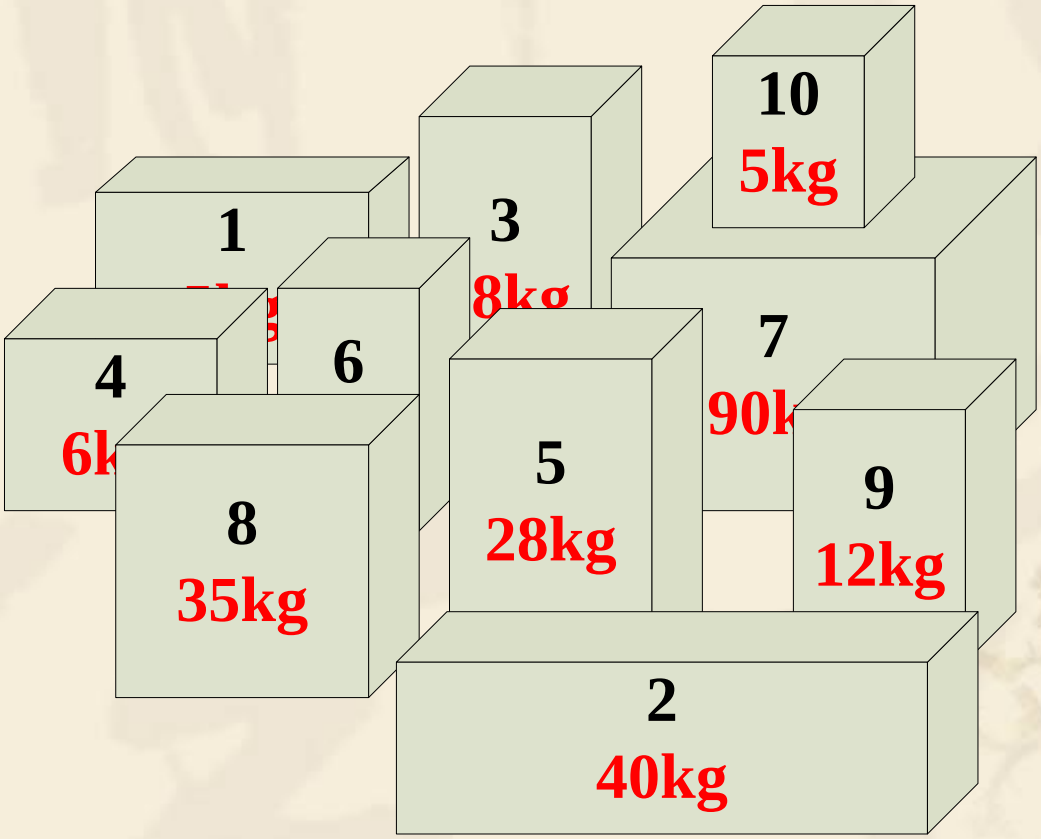
多维数组与多级指针

- 比较

	多维数组	多级指针	数组指针
各维边界	固定	不定	第一维不定
数据空间	连续	可不连续	连续
索引空间	无	需要	无

返回目录

索引查找与指针数组



索引查找

- 特点
  - 构成：索引表（多个）、数据表（唯一）
  - 按关键字建立索引表
  - 通过索引表查找数据项
  - 关键字排序不影响数据表
- 实现方式
  - 指针数组

返回目录

string字符串

- string类
  - C++标准类库（namespace std, <string>）
  - 优点
    - 兼容：支持cstring
    - 便捷：支持赋值、比较和连接运算
    - 安全：增强参数检查，支持自动扩展，有越界访问检查

string类的构造

- 默认构造：空字符串
- C字符串构造
- 拷贝构造

```
string a;
string b( “Hello” );
string c = “Hello” ;
string d = c;
```

string类的输入输出

- 输入
  - cin
  - cin.getline
  - cin.get
- 输出
  - cout

```
string str;
cin >> str;
cout << str;
cin.getline( str, 5, '\t');
cin.get( str, 5,  'e');
```

string类的运算

- 下标
- 赋值
- 比较
- 连接

```
string a;
string b( "Hello" );
string c = b;
b[0] = 'h';
a = "World";
if ( a != c )
{
        b = c + a;
        c += a;
}
```

string类可以自动扩展，但下标运算不会！

string类的常用操作

- 定位：at, substr
- 查找：find
- 插入：insert
- 删除：erase
- 统计：length
- 空字符串判断：empty

```
string a = "Hi, boy!", b;
int pos;

pos = a.find( "boy" );
a.erase( pos, 3 );
a.insert( pos, "girl" );
b = a.substr( 4, 3 );
if( !b.empty() )
{
        cout << a.at(4) << endl;
        cout << a.length();
}
```

string类与cstring

- C字符串 ← string类（显示强制转换）
- string类 ← C字符串（隐式自动转换）

```
string str = "Hello";
const char * p;
char * q = "OK";

p = str.c_str();
str = q;
```

返回目录

结构体

- 结构体（structure）的概念
  - 程序员自定义的用于组合数据的复合数据类型
  - 被组合的数据称为结构体的成员（structure member），按照定义时的顺序在内存中排列。

结构体类型的定义

- 定义语法

struct 结构体类型名 { 成员列表 };

例：

```
struct WordSta    // 单词统计
{
   string      s;      // 单词字符串
   int         n;      // 出现次数
};
struct WordSta x;
WordSta a, b;
```

结构类型变量的初始化

- 初始化语法

struct 结构类型名 变量名 = { 初始化列表 };

例：

WordSta a = { “Hello”, 1 };

结构类型变量的使用

- 按整体使用：整体复制

例：

```
WordSta a = { “hello”, 1 }, b;
b = a;
```

- 按成员使用：通过成员引用符（.或->）访问

例：

```
WordSta a, *p = &a;
a.s = “the”;
a.n = 10;
cout << p->s << ‘ ‘ << p->n << endl;
```