10 数组和指针

内容提要

- 数组
- 数组和指针
- 指针操作
- 保护数组内容
- 多维数组
- 变长数组
- 复合文字
- 关键概念

数组

1 数组

- 数组：一系列类型相同的元素构成的数据
- 数组声明：数组元素的数目、元素的类型
  - `float candy[365]; /* 内含365个float类型元素的数组 */`
  - `char code[12]; /*内含12个char类型元素的数组*/`
  - `int states[50]; /*内含50个int类型元素的数组 */`
  - 方括号（[ ]）表明candy、code和states都是数组，方括号中的数字表明数组中的元素个数
- 访问数组中的元素，通过下标（也称为索引）
  - 方括号下标方式：`states[4]`
  - 下标为整数，从0 开始计数

1.1 初始化数组

- 单个值的变量，标量变量（scalar variable）
- 数组初始化
  - 以逗号分隔的值列表（用花括号括起来）
  - `int powers[5]={1, 2, 4, 6, 8};`
- 对数组使用const
  - 声明并初始化只读数组
    - 在运行过程中不能修改该数组中的内容
  - `const int states[4]={1, 5, 8, 3};`
- 程序清单10.1 day_mon1.c

```c
/* day_mon1.c -- prints the days for each month */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %d has %2d days.\n", index +1,
               days[index]);

    return 0;
}
```

1.1 初始化数组

- 数组长度 初始值列表长度
  - 无数组长度，数组长度 = 初始值列表长度
  - 数组长度 > 初始值列表长度，初始化补全
  - 数组长度 < 初始值列表长度，编译错误
- 程序清单10.3 somedata.c
- 程序清单10.4 day_mon2.c

```c
/* some_data.c -- partially initialized array */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int some_data[SIZE] = {1492, 1066};
    int i;

    printf("%2s%14s\n",
           "i", "some_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, some_data[i]);

    return 0;
}
```

1.2 指定初始化器（C99）

- C99增加了一个新特性：
  - 指定初始化器（designated initializer）。利用该特性可以初始化指定的数组元素
- 程序清单10.5 designate.c

```
1.  // designate.c -- use designated initializers
2.  #include <stdio.h>
3.  #define MONTHS 12
4.  int main(void)
5.  {
6.      int days[MONTHS] = {31,28, [4] = 31,30,31, [1] =
    29};
7.      int i;
8.
9.      for (i = 0; i < MONTHS; i++)
10.         printf("%2d  %d\n", i + 1, days[i]);
11.
12.     return 0;
13. }
```

1.3 给数组元素赋值

- 声明数组后，可以借助数组下标（或索引）给数组元素赋值

- 不允许把数组作为一个单元赋给另一个数组
  - 实质上数组名代表的常量指针，不能再接受赋值，且指针形式的赋值，是浅拷贝！
  - 除初始化以外，不允许使用花括号列表的形式赋值

```c
#include <stdio.h>
int main(void)
{
    /* 给数组的元素赋值 */
    #define SIZE 50
    int counter, evens[SIZE];
    for (counter = 0; counter < SIZE; counter++)
            evens[counter] = 2 * counter;

    /* 一些无效的数组赋值 */
    #define SIZE 5
    int oxen[SIZE] = {5,3,2,8}; /* 初始化没问题 */
    int yaks[SIZE];
    yaks = oxen; /* 不允许 */
    yaks[SIZE] = oxen[SIZE]; /* 数组下标越界
}
```

1.4 数组边界

- 使用数组时，要防止数组下标超出边界
  - 确保下标是有效的值，0到长度-1之间的整数
- 程序清单10.6 bounds.c
- 编译器不检查数组下标
  - 使用越界下标的结果未定义
  - 意味着程序可运行，但结果不可预知或异常中止
- C程序员的原则：不检查边界，程序运行更快
  - 安全起见，编译器必须在运行时添加额外代码检查数组的每个下标值，但这会降低程序的运行速度

```
1.  #define SIZE 4
2.  int main(void)
3.  {
4.      int value1 = 44;
5.      int arr[SIZE];
6.      int value2 = 88;
7.      printf("v1 = %d, v2 = %d\n", value1, value2);
8.      for (int i = -1; i <= SIZE; i++)
9.          arr[i] = 2 * i + 1;
10.
11.     for (int i = -1; i < 7; i++)
12.         printf("%2d  %d\n", i , arr[i]);
13.     printf("v1 = %d, v2 = %d\n", value1, value2);
14.
15.     printf("address of arr[-1]: %p\n", &arr[-1]);
16.     printf("address of arr[4]:  %p\n", &arr[4]);
17.     printf("address of value1:  %p\n", &value1);
18.     printf("address of value2:  %p\n", &value2);
19.     return 0;
20. }
```

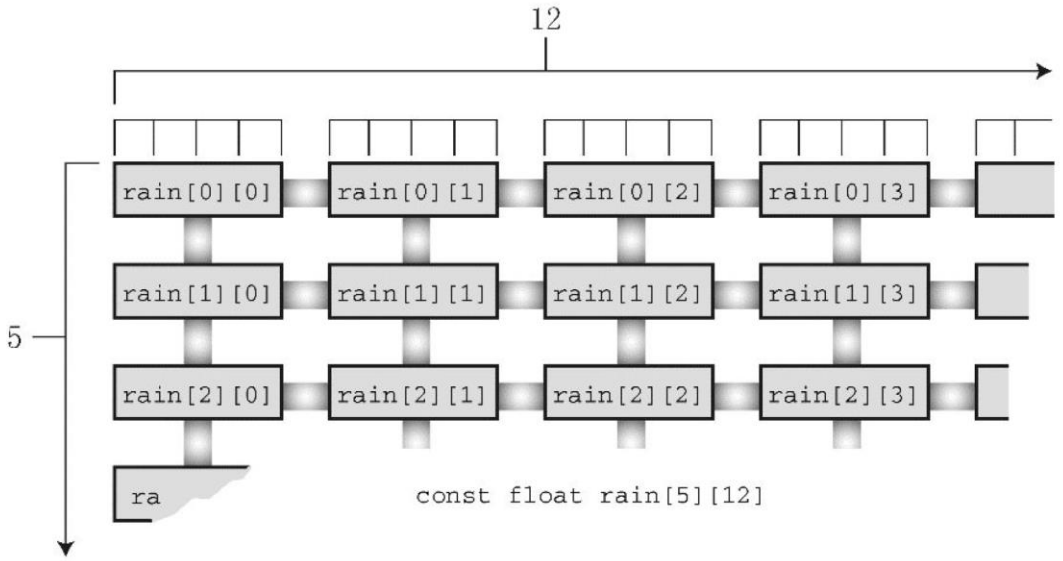
1.5 指定数组的大小（不推荐！！）

- C99标准允许创建了一种新型数组，称为变长数组（variable-length array）或简称VLA
  - **C11放弃了这一创新的举措**，把VLA设定为可选，而不是语言必备的特性

多维数组

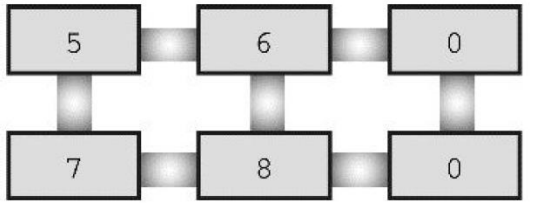
2 多维数组

- 一维数组排成一行，二维数组排成矩阵，三维数组堆成体空间
- float rain[5][12];
  - 表示二维数组m,有5行12列;
  - 5个元素的数组，每个元素为长12的float数组
  - 使用： rain[4][6] = 11.0f;
- 理解该声明的一种方法是，先查看中间部分
- float rain[5][12]; // rain是一个内含5个元素的数组
  - 说明数组rain有5个元素，至于每个元素的情况，要查看声明的其余部分
  - rain的每个元素是一个内含12个float类型值的数组

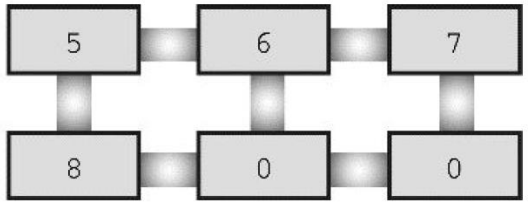


2.1 初始化二维数组

- 初始化二维数组是建立在初始化一维数组的基础上




```
int sq[2][3] = {{5,6},{7,8}};
```




```
int sq[2][3] = {5,6,7, 8};
```

2.2 其他多维数组

- `int box[10][20][30];`
- 可以把一维数组想象成一行数据，把二维数组想象成数据表，把三维数组想象成一叠数据表
- 还有一种理解box的方法是，把box看作数组的数组。也就是说，box内含10个元素，每个元素是内含20个元素的数组，这20个数组元素中的每个元素是内含30个元素的数组。或者，可以简单地根据所需的下标值去理解数组。
- 通常，处理三维数组要使用3重嵌套循环，处理四维数组要使用4重嵌套循环。对于其他多维数组，以此类推

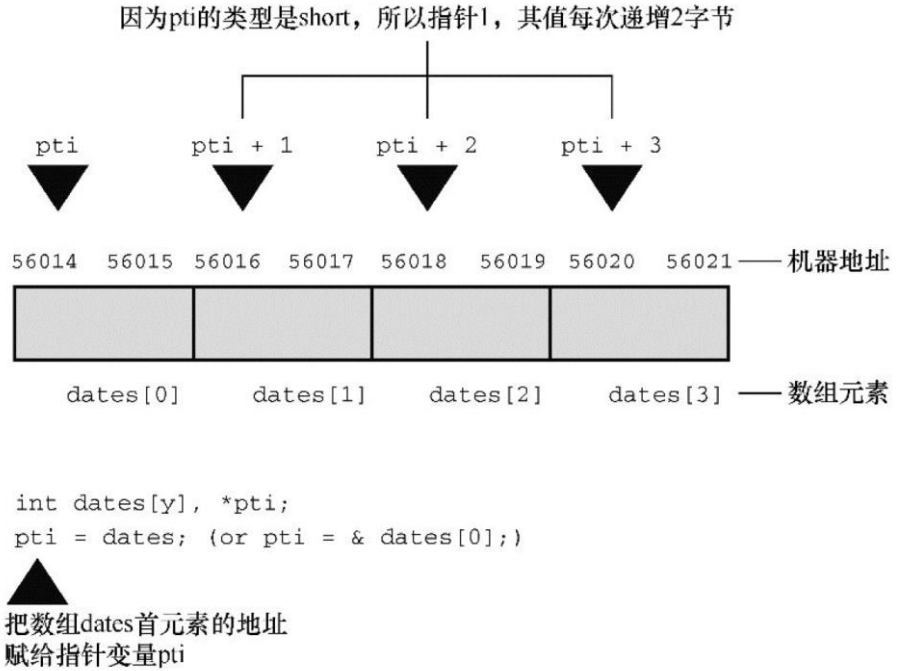
指针和数组

3 指针和数组

- 概述
  - 指针提供一种以符号形式使用地址的方法
  - 计算机硬件指令非常依赖地址，指针把程序员想要传达的指令以更接近机器的方式表达
    - 因此，使用指针的程序更有效率
- 指针
  - 指针的数值为它所指向的对象的地址
  - &用于取得变量的地址
  - 指针前用“ *“运算符，得到指针所指向对象的数值
- 数组和指针
  - 数组名是数组首元素的地址【指向数组的指针常量】
  - `flizny == &flizny[0]; // 数组名是该数组首元素的地址`

指针和数组

- 程序清单10.8 pnt_add.c
- 系统中地址按字节编址
  - 指针加1指增加一个存储单元
  - 对数组而言，加1后的地址是下一个元素的地址，而不是下一个字节的地址




```c
#include <stdio.h>
#define SIZE 4
int main(void){
    short dates [SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;

    pti = dates; // assign array address to pointer
    ptf = bills;
    printf("%23s %15s\n", "short", "double");
    for (index = 0; index < SIZE; index ++)
        printf("pointers + %d: %10p %10p\n",
               index, pti + index, ptf + index);

    return 0;
}
```

指针操作

- 指针的值是它所指向对象的地址，在指针前面使用*运算符可以得到该指针所指向对象的值
- 指针加1，指针的值递增它所指向类型的大小（以字节为单位）
- dates + 2 == &dates[2] // 相同的地址
- *(dates + 2) == dates[2] // 相同的值
- 可以使用指针标识数组的元素和获得元素的值
  - 从本质上看，同一个对象有两种表示法
  - 实际上，C语言标准在描述数组表示法时确实借助了指针

- 定义ar[n]的意思是*(ar + n)
  - 可以认为*(ar + n)的意思是“到内存的ar位置，然后移动n个单元，检索存储在那里的值”
  - 不要混淆 *(dates+2)和*dates+2。间接运算符（*）的优先级高于+，*dates+2相当于(*dates)+2：

指针操作

- 程序清单10.9 day_mon3.c
- days是数组首元素的地址
  - days + index是元素days[index]的地址
  - *(days + index)是该元素的值，相当于days[index]
  - 编译器编译这两种写法生成的代码相同
- for循环依次引用数组中的每个元素，并打印各元素的内容

```
1.  #include <stdio.h>
2.  #define MONTHS 12

3.  int main(void)
4.  {
5.      int days[MONTHS] =
    {31,28,31,30,31,30,31,31,30,31,30,31};
6.      int index;
7.
8.      for (index = 0; index < MONTHS; index++)
9.          printf("Month %2d has %d days.\n", index +1,
10.                 *(days + index)); // days[index]
11.
12.     return 0;
13. }
```

函数、数组和指针

4 函数、数组和指针

- 声明数组参量
  - 如果实参是数组名，形参必须是相配的指针
  - 4种等价的函数原型：
    - int sum(int *ar, int n);
      - int sum(int *, int) ;
    - int sum(int ar[], int n);
      - int sum(int [ ], int) ;
- 只有在函数原型或函数定义头中，才可以
  - 用int ar[]代替int * ar
  - 只需要参数类型，而不需要参数名
- 程序清单10.10 sum_arr1.c

```
1.  #define SIZE 10
2.  int sum(int ar[], int n);
3.  int main(void){
4.      int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
5.      long answer;
6.      answer = sum(marbles, SIZE);
7.      printf("The total number is %ld.\n", answer);
8.      printf("The size is %zd bytes.\n", sizeof marbles);
9.
10.     return 0;
11. }

12. int sum(int ar[], int n) {
13.     int total = 0;
14.     for(int i = 0; i < n; i++) total += ar[i];
15.     printf("ar is %zd bytes.\n", sizeof ar);
16.     return total;
17. }
```

4.1 使用指针形参

- 指针参数
  - 使用数组的函数需要知道何时开始何时结束，可以使用两个指针分别指明数组的开始和结束的地址
  - int sump(int * start, int * end);
- 程序清单10.11 sum_arr2.c
- 程序清单10.12 order.c

```c
#define SIZE 10
int sump(int * start, int * end);
int main(void){
    int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
    long answer;
    answer = sump(marbles, marbles + SIZE);
    printf("The total number is %ld.\n", answer);
    return 0;
}

/* use pointer arithmetic   */
int sump(int * start, int * end){
    int total = 0;
    while (start < end)    {
        total += *start; // add value to total
        start++; // advance pointer to next element
    }
    return total;
}
```

4.2 指针表示法和数组表示法

- ar[i]和*(ar+i)这两个表达式等价
  - 无论ar是数组名还是指针变量，这两个表达式都没问题。
  - 只有当ar是指针变量时，才能使用ar++这样的表达式

- 指针表示法（尤其与递增运算符一起使用时）更接近机器语言，一些编译器在编译时能生成效率更高的代码
  - 许多程序员认为主要任务是确保代码正确、逻辑清晰，代码优化应留给编译器

指针操作

5 指针操作

```c
// ptr_ops.c -- pointer operations
#include <stdio.h>
int main(void){
    int urn[5] = {100,200,300,400,500};
    int * ptr1, * ptr2, *ptr3;
    ptr1 = urn; // assign an address to a pointer
    ptr2 = &urn[2];     // ditto
    // dereference a pointer and take the address
    printf("pointer value, …, pointer address:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);
    // pointer addition
    ptr3 = ptr1 + 4;
    printf("\nadding an int to a pointer:\n");
    printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
           ptr1 + 4, *(ptr1 + 3));
    ptr1++; // increment a pointer
    printf("\nvalues after ptr1++:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);
    ptr2--;     // decrement a pointer
    printf("\nvalues after --ptr2:\n");
    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
           ptr2, *ptr2, &ptr2);
    --ptr1;   // restore to original value
    ++ptr2;   // restore to original value
    printf("\nPointers reset to original values:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
    // subtract one pointer from another
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 =
%td\n", ptr2, ptr1, ptr2 - ptr1);
    // subtract an integer from a pointer
    printf("p3 = %p, p3 - 2 = %p\n", ptr3, ptr3 - 2);
    return 0;
}
```

5 指针操作

- 赋值
  - 可以把地址赋给指针
- **解引用**
  - *运算符给出指针指向地址上存储的值
- **取址**
  - 和所有变量一样，指针变量也有自己的地址和值。对指针而言，&运算符给出指针本身的地址
- 指针与整数相加
  - 指针与整数相加，或整数与指针相加。整数和指针所指向类型的大小（以字节为单位）相乘，然后把结果与初始地址相加
- 递增指针
  - 递增指向数组元素的指针可以让该指针移动至数组的下一个元素
- 指针减去一个整数
  - 使用-运算符从一个指针中减去一个整数。指针必须是第1个运算对象，整数是第2个运算对象。该整数将乘以指针指向类型的大小（以字节为单位），然后用初始地址减去乘积
- 递减指针
- 指针求差
  - 计算两个指针的差值
- 比较
  - 使用关系运算符比较两个指针的值，前提是两个指针都指向相同类型的对象
- 千万不要解引用未初始化的指针
  - `int * pt; // 未初始化的指针`
  - `*pt = 5; // 严重的错误`

保护数组内容

6 保护数组内容

- 如果函数的意图不是修改数组中的数据内容，那么在函数原型和函数定义中声明形式参数时应使用关键字const
  - 函数处理数组的时候把它当成不变，函数体里不能修改数组元素值
  - `int sum(const int ar [ ], int n)`

- 程序清单10.14　arf.c程序

```
1.  #define SIZE 5
2.  // displays array contents
3.  void show_array(const double ar[], int n){
4.     for (int i = 0; i < n; i++) printf("%8.3f", ar[i]);
5.  }
6.  // multiplies each array
7.  void mult_array(double ar[], int n, double mult){
8.        for (int i = 0; i < n; i++) ar[i] *= mult;
9.  }
10. int main(void){
11.     double dip[SIZE] = {20.0, 17.6, 8.2, 15.3, 22.2};
12.     printf("The original dip array:\n");
13.     show_array(dip, SIZE);
14.     mult_array(dip, SIZE, 2.5);
15.     printf("Array after calling mult_array():\n");
16.     show_array(dip, SIZE);
17.     return 0;
18. }
```

6.2 const的其他内容

- const的其他内容
  - 可以把常量或非常量数据的地址赋给常量的指针；只有非常量数据的指针能赋给普通指针
  - `const int num[3] = {1,3,5};`
  - `int year[6] = {1,8,4,11,13,16};`
  - `const int *pc = num ; //合法`
  - `pc = year ; //合法`
  - `int *pnc = year; //合法`
  - `pnc = num; //非法`

指针与多维数组

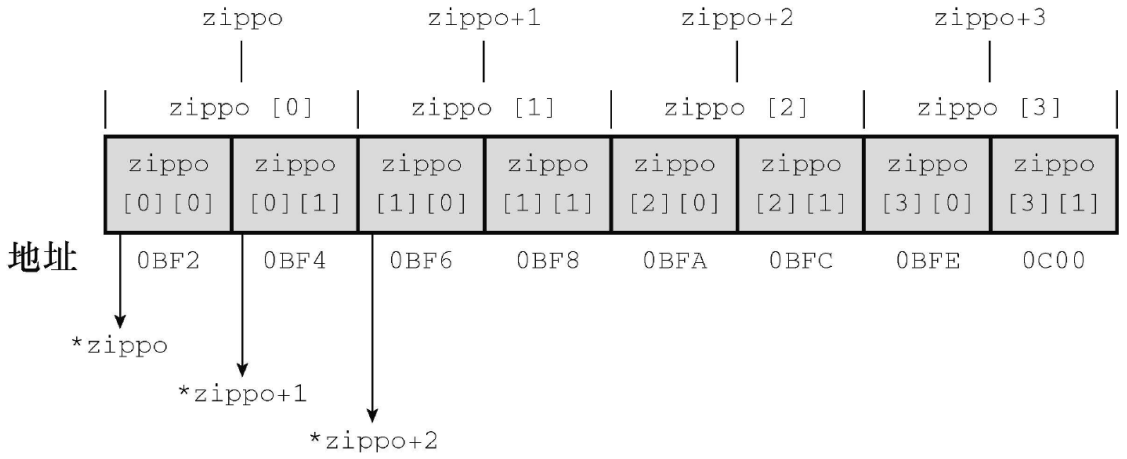
7 指针与多维数组

- int zippo[4][2]; //
  - zippo为长度为4的数组， zippo[0]是一个int[2]的数组
  - zippo是二维数组的首地址： zippo = &zippo[0]
  - *zippo代表（zippo[0]）
- 指针表示多维数组m
  - zippo[m][n]==*(*(zippo + m) +n)

zippo	←二维数组首元素的地址（每个元素都是内含两个 int 类型元素的一维数组）
zippo+2	←二维数组的第 3 个元素（即一维数组）的地址
*(zippo+2)	←二维数组的第 3 个元素（即一维数组）的首元素（一个 int 类型的值）地址
*(zippo+2) + 1	←二维数组的第 3 个元素（即一维数组）的第 2 个元素（也是一个 int 类型的值）地址
((zippo+2) + 1)	←二维数组的第 3 个一维数组元素的第 2 个 int 类型元素的值，即数组的第 3 行第 2 列的值（zippo[2][1]）

多维数组

- int zippo[4][2]; /* 内含int数组的数组 */
- 程序清单10.15 zippo1.c
- 因为zippo是数组首元素的地址，所以zippo的值和&zippo[0]的值相同。而zippo[0]本身是一个内含两个整数的数组，所以zippo[0]的值和它首元素（一个整数）的地址（即&zippo[0][0]的值）相同。
- 给指针或地址加1，其值会增加对应类型大小的数值
- 解引用一个指针（在指针前使用*运算符）或在数组名后使用带下标的[]运算符，得到引用对象代表的值




```c
/* zippo1.c --  zippo info */
#include <stdio.h>
int main(void){
    int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
    printf("z = %p, z + 1 = %p\n", zippo, zippo + 1);
    printf("z[0] = %p, z[0] + 1 = %p\n", zippo[0],
zippo[0] + 1);
    printf("*z = %p, *z+1 = %p\n", *zippo, *zippo + 1);
    printf("z[0][0] = %d\n", zippo[0][0]);
    printf("*z[0] = %d\n", *zippo[0]);
    printf("**z = %d\n", **zippo);
    printf("z[2][1] = %d\n", zippo[2][1]);
    printf("*(*(zi+2) + 1) = %d\n", *(*(zippo+2) + 1));

    return 0;
}
```

多维数组

- 声明二维数组的指针
  - int(* pz)[2]; // pz指向一个内含两个int类型值的数组
- 程序清单10.16 zippo2.c

```
1.  /* zippo2.c --  zippo info via a pointer variable */
2.  #include <stdio.h>
3.  int main(void){
4.      int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
5.      int (*pz)[2];
6.      pz = zippo;
7.
8.      printf("pz = %p, pz + 1 = %p\n", pz, pz + 1);
9.      printf("pz[0] = %p, pz[0] + 1 = %p\n", pz[0],
    pz[0] + 1);
10.     printf("*pz = %p, *pz + 1 = %p\n", *pz, *pz + 1);
11.     printf("pz[0][0] = %d\n", pz[0][0]);
12.     printf("*pz[0] = %d\n", *pz[0]);
13.     printf("**pz = %d\n", **pz);
14.     printf("pz[2][1] = %d\n", pz[2][1]);
15.     printf("*(*(pz+2) + 1) = %d\n", *(*(pz+2) + 1));
16.
17.     return 0;
18. }
```

7.2 指针的兼容性

- 指针之间的赋值比数值类型之间的赋值要严格
- 无效的赋值表达式语句中涉及的两个指针都是指向不同的类型

7.3 函数和多维数组

- `void somefunction( int (* pt)[4] );`
- 当且仅当pt是一个函数的形式参数时
  - `void somefunction(int pt[][4]);`

- 实际上只能定义出一维数组
  - 二维数组只能看作是指向一个具体一维数组类型的一维数组

```
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int[][COLS], int);      // 省略形参名，没问题
int sum2d(int (*ar)[COLS], int rows); // 另一种语法
int sum2(int ar[][], int rows);       // 错误的声明
```

array2d.c

```c
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int ); //omit names
int sum2d(int (*ar)[COLS], int rows);//another

int main(void){
    int junk[ROWS][COLS] = {{2,4,6,8}, {3,5,7,9},
{12,10,8,6} };
    sum_rows(junk, ROWS);
    sum_cols(junk, ROWS);
    printf("Sum of … = %d\n", sum2d(junk, ROWS));
    return 0;
}
void sum_rows(int ar[][COLS], int rows){
    for (int r = 0; r < rows; r++){
        int tot = 0;
        for (int c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("row %d: sum = %d\n", r, tot);
    }
}

void sum_cols(int ar[][COLS], int rows){
    for (int c = 0; c < COLS; c++){
        int tot = 0;
        for (int r = 0; r < rows; r++)
            tot += ar[r][c];
        printf("col %d: sum = %d\n", c, tot);
    }
}
int sum2d(int ar[][COLS], int rows){
    int tot = 0;
    for (int r = 0; r < rows; r++)
        for (int c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

变长数组（不推荐！！）

8 变长数组（不推荐！！）

- 声明变长数组不可以初始化
  - `int a=4;  int b=5;`
  - `double sales[a][b];//一个变长数组`
- 变长数组名实质上是指针，具有变长数组参量的函数实际上直接使用原数组，但可以改变作为参数的数组
- 注意！变长数组不能改变大小
- 变长数组中的“变”不是指可以修改已创建数组的大小。一旦创建了变长数组，它的大小则保持不变。这里的“变”指的是：在创建数组时，可以使用变量指定数组的维度

复合字面量

9 复合字面量【不推荐】

- 复合文字像是在数组的初始化列表面前加上用圆括号括起来的类型名。

```
(int [2]){10, 20} // 复合字面量
(int []){50, 20, 90} // 内含3个元素的复合字面量
int * pt1;
pt1 = (int [2]) {10, 20};
```

- 程序清单10.19 flc.c

```
int sum(const int ar[], int n){
    int i, total = 0;
    for( i = 0; i < n; i++) total += ar[i];
    return total;
}
int sum2d(const int ar[][COLS], int rows){
    int r, c, tot = 0;
    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

```
#include <stdio.h>
#define COLS 4
int sum2d(const int ar[][COLS], int rows);
int sum(const int ar[], int n);
int main(void){
    int total1, total2, total3;
    int * pt1;
    int (*pt2)[COLS];
    pt1 = (int [2]) {10, 20};
    pt2 = (int [2][COLS]) { {1,2,3,-9}, {4,5,6,-8} };
    total1 = sum(pt1, 2);
    total2 = sum2d(pt2, 2);
    total3 = sum((int []){4,4,4,5,5,5}, 6);
    printf("total1 = %d\n", total1);
    printf("total2 = %d\n", total2);
    printf("total3 = %d\n", total3);
    return 0;
}
```

关键概念

关键概念

- 数组用于存储相同类型的数据。C把数组看作是派生类型，数组是建立在其他类型的基础上
- 把数组名作为实际参数时，传递给函数的不是整个数组，而是数组的地址（因此，函数对应的形式参数是指针）
  - 为了处理数组，函数必须知道从何处开始读取数据和要处理多少个数组元素
- 数组和指针的关系密切，同一个操作可以用数组表示法或指针表示法
  - 它们之间的关系允许你在处理数组的函数中使用数组表示法，即使函数的形式参数是一个指针，而不是数组