A powerful index.

C程序设计

T10

数组和指针

厦门大学信息学院软件工程系

黄炜 副教授

主要内容

- 数组
  - 一维数组的声明与使用
  - 数组下标越界
  - 多维数组的声明与使用
- 指针
  - 指针的声明、赋值与使用
  - 指针的操作、指针和数组之间的关系
  - 指针的应用场景

1. 数组声明和使用

数组的声明

- 格式

```
<类型> <数组名>[<数组长度>];
```

  - 声明语句的方括号是**修饰符**；而表达式的方括号是**操作符**。
  - 数组长度应为无符号**整型常数**，可以为0。

```
int arr[100];
double score[50], final_score[50];
```

  - 如果声明时在类型前加注const，数组**元素**不可更改。
- 数组是有序的元素序列。
- 数组也应该**先声明，再赋值（初始化），最后使用**

数组的初始化

- 声明时初始化：用复合文字（{ }）初始化
  - 复合文字中如指定下标，则以指定的下标为准
  - 复合文字中未指定下标，则下标为左边相邻元素下标加1
  - 复合文字中第一个元素下标未指定时，下标为0
  - 复合文字中未指定的元素赋值为0
  - 如果数组长度不指定，以复合文字的长度为准

```
int powers[8] = {1,2,4,6,8,16,32,64}; /* ANSI C and later */
int powers[8] = {1,2,[4]=4,6,[1]=16,32}; /* C99 */
int powers[8] = {}; /* C99 */
int powers[] = {1,2,4,6,8,16,32,64}; /* ANSI C and later */
```

数组的初始化

- 声明时后另行初始化
  - 不得使用复合文字
  - 使用循环进行初始化（不一定要初始化为0）

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
    arr[i] = 0;
```

  - 使用内存操作函数进行初始化为0（注意内存格式）

```
memset(arr, 0, sizeof(arr));
```

数组的元素访问

- 格式

```
<数组名> [<下标>]
```

  - 表达式的方括号是**操作符**；而声明语句的方括号是**修饰符**。
  - 其中：数组下标应为**整型表达式**，否则会有编译错误
    - 可以为负数，甚至超过其边界（程序员应避免此做法）
    - 数组越界将产生运行错误
  - 例如：

```
arr[3]=0;
```

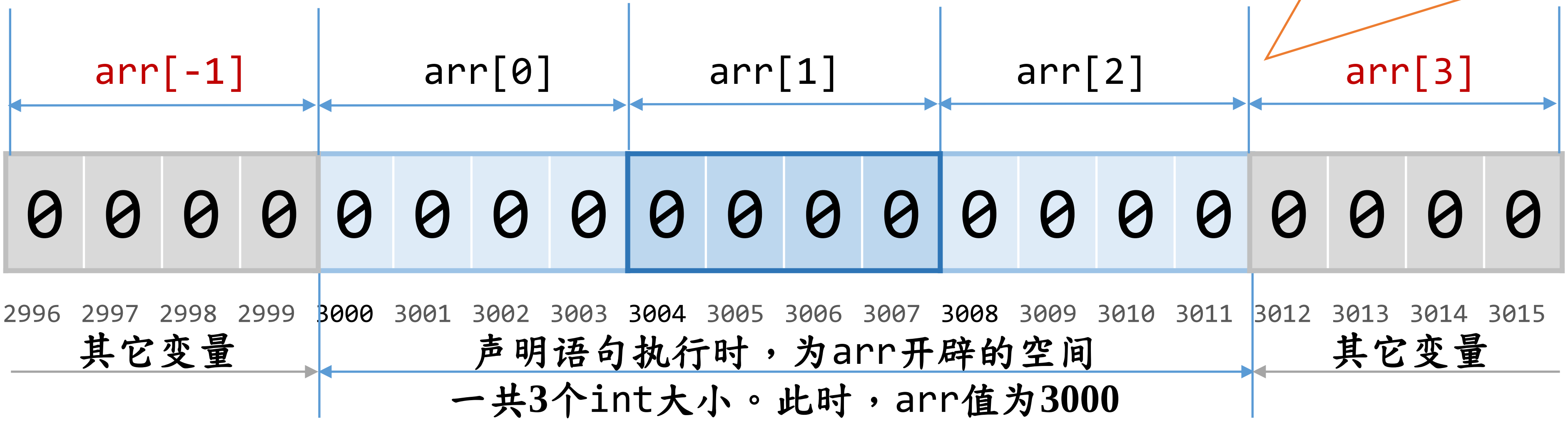
  - 表示：数组首地址起前进下标所指步数，步长为元素宽度
  - 数组应先声明，再初始化或赋值，才可以访问

数组的元素

```
int arr[3];
arr[1]=0;
```

- 表达式使用 **操作符** [] 访问具体元素
  - 数组名的值为数组声明时开辟的内存空间首地址
  - arr[index]是以arr为基准，index为步数，元素类型的长度为步长，所指内存区域的值

如果该区域是程序不可访问的，将产生**运行错误**；否则不会产生错误。



arr[-1]				arr[0]				arr[1]				arr[2]				arr[3]			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007	3008	3009	3010	3011	3012	3013	3014	3015

其它变量

声明语句执行时，为arr开辟的空间一共3个int大小。此时，arr值为3000

其它变量

```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;
}
```

数组的声明

for循环访问数组，一般以0开始，以“<数组长度”结束，下标增量

数组的引用

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
（此处省略数行）
Month 12 has 31 days.
```

```
/* no_data.c -- uninitialized array */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int no_data[SIZE];  /* uninitialized array */
    int i;

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

    return 0;
}
```

数组的声明不等于初始化

数组的元素不经初始化不可访问，否则结果不可靠

```
 i    no_data[i]
 0    -858993460
 1    -858993460
 2    -858993460
 3    -858993460
```

```c
/* day_mon2.c -- letting the compiler count elements */
#include <stdio.h>
int main(void)
{
    const int days[] = {31,28,31,30,31,30,31,31,30,31};
    int index;

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

    return 0;
}
```

有经验的程序员将不应修改元素的数组标记为const避免误改

在数组声明范围内数组长度为sizeof(a) / sizeof(a[0])

声明index，应按物理意义；此处不可以写成：for (index = 1; index <= LENGTH; index++)

```
Month  1 has 31 days.
Month  2 has 28 days.
（此处省略数行）
Month 10 has 31 days.
```

```c
// designate.c -- use designated initializers
#include <stdio.h>
#define MONTHS 12
int main(void)
{
    int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
    int i;

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

    return 0;
}
```

```
 1  31
 2  29
 3  0
（此处省略数行）
12  0
```

```c
// bounds.c -- exceed the bounds of an array
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int value1 = 44;
    int arr[SIZE];
    int value2 = 88;
    int i;
    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;
    for (i = -1; i < 7; i++)
        printf("%2d  %d\n", i , arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    printf("address of arr[-1]: %p\n", &arr[-1]);
    printf("address of arr[4]:  %p\n", &arr[4]);
    printf("address of value1:  %p\n", &value1);
    printf("address of value2:  %p\n", &value2);
    return 0;
}
```

程序员不应越界访问数组，不应利用越界对数组上下的变量赋值

Visual Studio	Linux
value1 = 44, value2 = 88 -1 -1 0 1 1 3 2 5 3 7 4 9 5 -858993460 6 44 value1 = 44, value2 = 88 address of arr[-1]: 002CFBC4 address of arr[4]: 002CFBD8 address of value1: 002CFBEC address of value2: 002CFBB0	value1 = 44, value2 = 88 -1 -1 0 1 1 3 2 5 3 7 4 9 5 5 6 0 value1 = 9, value2 = -1 address of arr[-1]: 0028FED4 address of arr[4]: 0028FEE8 address of value1: 0028FEE8 address of value2: 0028FED4

结果分析

- 实际结果视编译器而定
  - Visual Studio

value2					a[-1]	a[0]	a[1]	a[2]	a[3]	a[4]					value1
88					-1	1	3	5	7	9	未知				44
B0	B4	B8	BC	C0	C4	C8	CC	D0	D4	D8	DC	E0	E4	E8	EC

002CFB+

  - Linux

									value2 a[-1]	a[0]	a[1]	a[2]	a[3]	value1 a[4]	
									-1	1	3	5	7	9	
B0	B4	B8	BC	C0	C4	C8	CC	D0	D4	D8	DC	E0	E4	E8	EC

0028FE+

断言

- 断言
  - 包含assert.h文件，只在DEBUG模式下有效

```
#ifdef NDEBUG           /* required by ANSI standard */
# define assert(__e) ((void)0)
#else
# define assert(__e) ((__e) ? (void)0 : __assert_func (__FILE__,
__LINE__, __ASSERT_FUNC, #__e))
```

  - 作用：当断言的表达式为假时，程序异常中止
- 利用断言检查数组下标越界的错误
  - 数组下标越界往往是运行错误的原因
  - 声明过大的数组也可能是运行或编译错误的原因

利用断言检查数组下标越界错误

- 原有程序

```c
#include <stdio.h>
#define ARR_LENGTH 10

int main()
{
    int a[ARR_LENGTH];

    for (int i = 0; i < ARR_LENGTH; i++)
        a[i] = i * 2;
    for (int i = 0; i < ARR_LENGTH; i += 2)
        a[i] = a[i + 1] - a[i + 2];
    for (int i = 0; i < ARR_LENGTH; i++)
        printf("a[%d]=%d\n", i, a[i]);

    return 0;
}
```

这里下标i受到for的限制，不可能越界，不需要判断

这里下标不是i而是i的表达式，未直观地受到for的限制，可能越界，需要判断

利用断言检查数组下标越界错误

- 修改程序

```
#include <stdio.h>
#include <assert.h>
#define ARR_LENGTH 10
int main()
{
    int a[ARR_LENGTH];
    for (int i = 0; i < ARR_LENGTH; i++)
        a[i] = i * 2;
    for (int i = 0; i < ARR_LENGTH; i += 2)
    {
        assert(i + 1 >= 0 && i + 1 < ARR_LENGTH);
        assert(i + 2 >= 0 && i + 2 < ARR_LENGTH);
        a[i] = a[i + 1] - a[i + 2];
    }
    for (int i = 0; i < ARR_LENGTH; i++)
        printf("a[%d]=%d\n", i, a[i]);
    return 0;
}
```

有经验的程序员制备大量测试数据，运行程序，如果在这里中止，说明该组数据下，数组下标越界。

2. 多维数组

多维数组的声明

- 格式

```
<类型> <数组名>[<数组长度3>][<数组长度2>][<数组长度1>];
```

  - 数组长度应为无符号**整型常数**，可以为0。
  - 元素个数为各维度的乘积
  - 高维数组可以视为数组的数组，高维在前，低维在后
  - 例：

```
int matrix[3][5];
```

这是3个int m[5]堆起来的大数组

- 声明时初始化

```
int powers[2][8] = {{1,2,4,6,8,16,32,64},
                    {1,2,4,6,8,16,32,64}};
```

多维数组的访问

- **格式**

```
<数组名>[<下标3>][<下标2>][<下标1>];
```

  - 其中：数组下标应为**整型表达式**，否则会有编译错误
    - 可以为负数，甚至超过其边界（程序员应避免此做法）
  - 例如：

```
matrix[2][4]=6;
```

- 多维数组的求值应先计算偏移量再求值
  - 只要偏移量所指向内存区域相同，其值也相同
    - 但应该书写含义明确的下标形式

```
int m[3][5];
```

这是m[1][4]，也是m[0][9], m[2][-1], m[-1][14], m[3][-6]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
/* rain.c  -- finds yearly totals, yearly average, and monthly
 average for several years of rainfall data */
#include <stdio.h>
#define MONTHS 12    // number of months in a year
#define YEARS   5    // number of years of data
int main(void)
{
    // initializing rainfall data for 2010 - 2014
    const float rain[YEARS][MONTHS] =
    {
        {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
        {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
        {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
        {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
    };
    int year, month;
    float subtot, total;
```

```c
printf(" YEAR    RAINFALL  (inches)\n");
for (year = 0, total = 0; year < YEARS; year++)
{             // for each year, sum rainfall for each month
    for (month = 0, subtot = 0; month < MONTHS; month++)
        subtot += rain[year][month];
    printf("%5d %15.1f\n", 2010 + year, subtot);
    total += subtot; // total for all years
}
printf("\nThe yearly average is %.1f inches.\n\n",
       total/YEARS);
printf("MONTHLY AVERAGES:\n\n");
printf(" Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct ");
printf(" Nov  Dec\n");

for (month = 0; month < MONTHS; month++)
{             // for each month, sum rainfall over years
    for (year = 0, subtot =0; year < YEARS; year++)
        subtot += rain[year][month];
        printf("%4.1f ", subtot/YEARS);
    }
    printf("\n");

    return 0;
}
```

```
 YEAR    RAINFALL  (inches)
 2010            32.4
 2011            37.9
 2012            49.8
 2013            44.0
 2014            32.9

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
 7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
```

数组的局限性

- 例题
  - 比较2个字符串的首字母，将较大的字符串首置为大写

```
if (a[0] >= b[0])
    a[0] = a[0] - 'a' + 'A';
else
    b[0] = b[0] - 'a' + 'A';
```

  - 将字符串首置为大写字母，与字符串名无关

```
if (a[0] >= b[0])
    p = a;
else
    p = b;
p[0] = p[0] - 'a' + 'A';
```

如果存在一种类型，作为能存储数组名的变量，那就更高内聚低耦合了。这样的类型是指针。

3. 内存的组织

内存中的数据

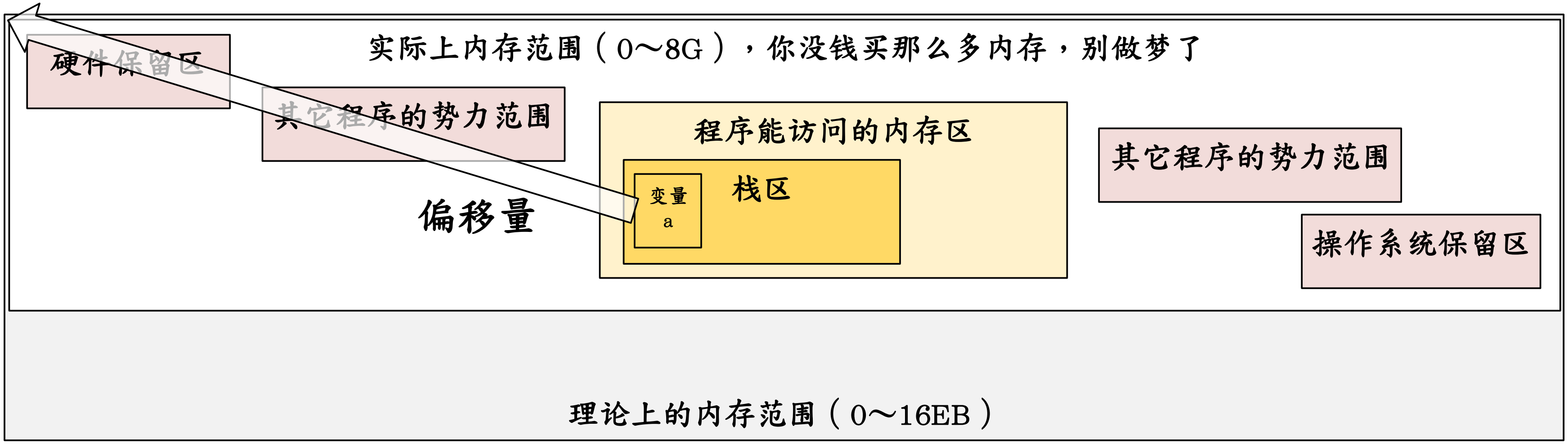
- 计算机程序中的变量在运行时存储于内存中
- 内存数据的单位
  - 内存中数据以电平（0、1）的形式存储，称为位（bit）
  - 内存中数据的最小单位是8位，称为字节（Byte）
    - 因为2^3=8位才能表示不少于26个英文字母的情况
- 内存地址：内存每个字节的编号，称为内存地址
  - 内存地址的最小值是0，最大值由CPU架构、操作系统和程序类型决定，并受计算机实际内存容量限制

硬件保留区	某进程占用	有空	某进程占用	主引导记录等

内存地址

- 内存地址的最小值为0，最大值由应用程序架构决定

CPU架构	操作系统	应用程序	内存范围	备注
x86	32位	32位	$0\sim2^{32}-1$B	内存范围不应超过实际内存大小
x64	32位	32位	$0\sim2^{32}-1$B	
	64位	32位	$0\sim2^{32}-1$B	通过WoW或运行时库
		64位	$0\sim2^{64}-1$B	



实际上内存范围（0～8G），你没钱买那么多内存，别做梦了

硬件保留区

其它程序的势力范围

程序能访问的内存区

栈区

变量
a

偏移量

其它程序的势力范围

操作系统保留区

理论上的内存范围（0～16EB）

声明变量时的内存分配

- 执行声明语句时

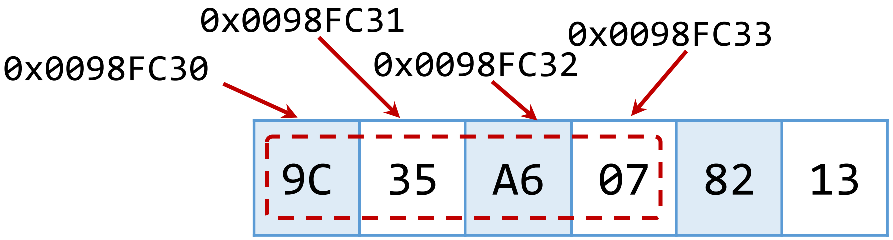
```
int a;
```

  - 内存的栈区中开辟一个空间
  - 从而变量a具有
    - 内存地址（偏移量）
    - 长度
    - 值（取决于长度和位的组织格式）

4. 指针的声明、赋值和使用

指针的声明

- 指针为无符号整数，其物理意义为指向内存地址
  - 指针的值为内存地址的偏移量
- 指针含义
  - 内存中一段区域的首地址
  - 格式为“类型名”的类型、长度为“类型名”的长度
- 指针是变量，具有地址和值

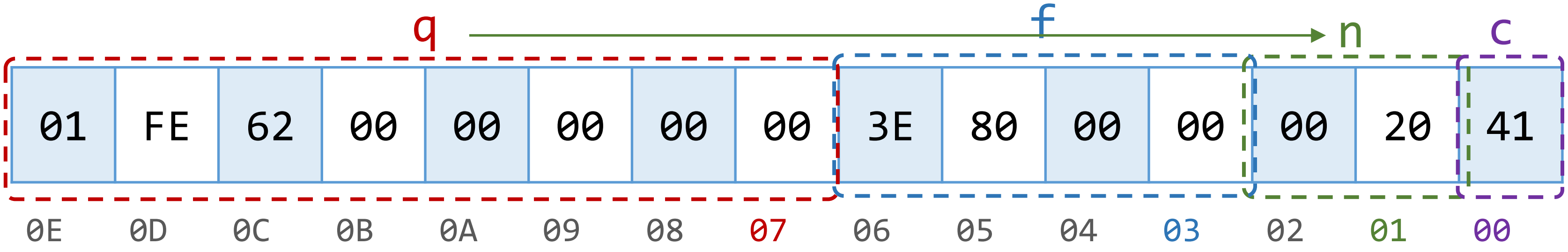


指针的声明

- 指针的声明
  - 格式：`<类型> *<指针名>;`
    - 例如：`int *p, *q, r;`
  - 声明语句中使用修饰符 * 标记一个变量为指针
    - 这一点表达式中的操作符 * 含义不同。
    - 列表形式声明语句，每个指针前应单独书写修饰符。
  - 指针声明时应指明其指向内存地址的数据类型
- 指针的存储大小为4或8字节（由程序架构决定）

声明指针变量时的内存分配

- 指针也是数据类型，指针变量也有所在地址和值



q → n

f

c

0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
01	FE	62	00	00	00	00	00	3E	80	00	00	00	20	41

变量名	内存地址	值	长度
q	0x0062FE07	0x0062FE01	8B
f	0x0062FE03	0.25	4B
n	0x0062FE01	32	2B
c	0x0062FE00	'A'	1B

```
char c = 'A';
short n = 32;
float f = 0.25;
short *q = &n;
```

指针的赋值

- 指针应先声明，再赋值，最后使用
- 指针的赋值形式
  - 指针类型常数

```
int *p = NULL;
```

    - 通常以 NULL（即：(void *)0）表示指针无定义
  - 同类型的变量取地址

```
int q = 0, *p = &q;
```

  - 值为同类型的指针表达式

```
int *p = &q, *r = p + 1;
```

指针基本操作

- 间接引用（indirection / dereference）操作符 *
  - 操作符后应为指针类型变量或常量，例如：*p
  - 查找p值指向内存区域存储的值，值由p的数据类型决定
- 取地址（address）操作符 &
  - 操作符后应为变量或const常量，例如：&p
  - 查找变量p在声明时开辟的内存区域首地址，类型为指针
  - *(*p)有意义但&(&p)没有意义

```c
/* loccheck.c -- checks to see where variables are stored  */
#include <stdio.h>
void mikado(int);                        /* declare function  */
int main(void)
{
    int pooh = 2, bah = 5;               /* local to main()   */
    printf("In main(), pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("In main(), bah = %d and &bah = %p\n", bah, &bah);
    mikado(pooh);
    return 0;
}
void mikado(int bah)
{
    int pooh = 10;                       /* local to mikado() */
    printf("In mikado(), pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("In mikado(), bah = %d and &bah = %p\n", bah, &bah);
}
```

```
In main(), pooh = 2 and &pooh = 00C4F7B0
In main(), bah = 5 and &bah = 00C4F7A4
In mikado(), pooh = 10 and &pooh = 00C4F6C0
In mikado(), bah = 2 and &bah = 00C4F6D0
```

指针的修饰符与操作符的区别

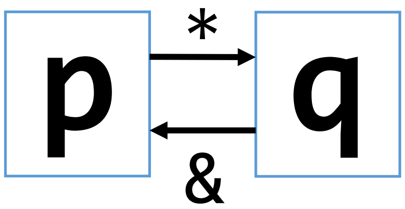
- 例题
  - 执行 `int q = 1, *p = &q;` 后，*p 的值为多少？

```c
#include <stdio.h>

int main() {
    int q = 1, * p = &q;
    printf("&p=%p\n", &p);
    printf(" p=%p\n", p);
    printf("*p=%d\n", *p);
    printf("&q=%p\n", &q);
    printf(" q=%d\n", q);
    return 0;
}
```

这里*是修饰符，和表达式中的*不同，因此运行后p的值为&q的值。
不能认为*p的值为&q的值。

```
&p=0xffffcbc8
 p=0xffffcbc4
*p=1
&q=0xffffcbc4
 q=1
```



指针的指向类型

- 指向数据的指针
  - 指向基本数据类型的指针

```
char q = 'A'; char *p = &q;
```

  - 指向数组的指针

```
int a[5] = {0}; int *p = a;
```

  - 指向指针的指针

```
int a = 0; int *p = &a;
int **pp = &p;
```

- 指向代码的指针
  - 指向函数的指针

```
double fabs (double _X);
double (*q) (double) = fabs;
```

5. 指针的操作

指针操作

- 指针可以执行以下操作
  - 类型变量增/减时的步长为所指向类型的大小

操作	常量	变量	示例
赋值	×	√	ptr=&var
求值或取值	√	√	*ptr
取指针地址	√	√	&var
加上或减去一个整数	√	√	ptr+num
自增或自减	×	√	ptr++
求差值	√	√	ptr1-ptr2
比较大小	√	√	ptr1<=ptr2

```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 of a pointer
    printf("pointer value, dereferenced pointer, 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("\nsubtracting one pointer from another:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n",
           ptr2, ptr1, ptr2 - ptr1);
    // subtract an integer from a pointer
    printf("\nsubtracting an int from a pointer:\n");
    printf("ptr3 = %p, ptr3 - 2 = %p\n",
           ptr3,  ptr3 - 2);

    return 0;
}
```

```
pointer value, dereferenced pointer, pointer address:
ptr1 = 0xbf804b6c, *ptr1 =100, &ptr1 = 0xbf804b60

adding an int to a pointer:
ptr1 + 4 = 0xbf804b7c, *(ptr4 + 3) = 400

values after ptr1++:
ptr1 = 0xbf804b70, *ptr1 =200, &ptr1 = 0xbf804b60

values after --ptr2:
ptr2 = 0xbf804b70, *ptr2 = 200, &ptr2 = 0xbf804b64

Pointers reset to original values:
ptr1 = 0xbf804b6c, ptr2 = 0xbf804b74

subtracting one pointer from another:
ptr2 = 0xbf804b74, ptr1 = 0xbf804b6c, ptr2 - ptr1 = 2

subtracting an int from a pointer:
ptr3 = 0xbf804b7c, ptr3 - 2 = 0xbf804b74
```

对只读参量标记 const

- 对传指针却不做修改的参量应标记const
  - 示例：只读的数组（如有赋值则报错）

```
void show_array(const double ar[], int n);
```

```
error C2166: l-value specifies const object
```

  - 但不保证绝不赋值（假设通过传入的其它参数修改其值）

- 不修改指针应对指针本身标记const

```
void show_array(double * const ar, int n);
```

- 两者都不修改应都标记const

```
void show_array(const double * const ar, int n);
```

```c
/* arf.c -- array functions */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
    double dip[SIZE] = {20.0, 17.66, 8.2, 15.3, 22.22};

    printf("The original dip array:\n");
    show_array(dip, SIZE);
    mult_array(dip, SIZE, 2.5);
    printf("The dip array after calling mult_array():\n");
    show_array(dip, SIZE);

    return 0;
}
```

```
/* displays array contents */
void show_array(const double ar[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%8.3f ", ar[i]);
    putchar('\n');
}

/* multiplies each array member by the same multiplier */
void mult_array(double ar[], int n, double mult)
{
    int i;
    for (i = 0; i < n; i++)
        ar[i] *= mult;
}
```

```
The original dip array:
  20.000   17.660    8.200   15.300   22.220
The dip array after calling mult_array():
  50.000   44.150   20.500   38.250   55.550
```

指针和多维数组

- 指针可以指向任何类型，包括多维数组
- 函数传入多维数组时
  - 应在数组参量中指出除最高维以外的维度
    - 否则下标无意义
  - 并用参数指出其最高维的限度
    - 用于防止越界

在函数中通过指针使用多维数组

- 一定要正确定义指向多维数组的指针（参见前页）
  - 长度不定的部分，通过设置参量传入
  - 长度确定的部分，通过设置常量传入

函数形式参量声明	说明
`int sum2(int ar[][], int rows);`	错误，低维长度未指定
`int sum2(int ar[][4], int rows);`	正确
`int sum2(int ar[3][4], int rows);`	正确，但高维长度（3）被忽略

  - 建议通过typedef实现

```
typedef int arr4[4]; // arr4 array of 4 int
typedef arr4 arr3x4[3]; // arr3x4 array of 3 arr4
int sum2(arr3x4 ar, int rows); // same as next declaration
```

在函数中通过指针使用多维数组

- 函数中使用数组应正确区分指针和数组

```
int sum4d(int ar[][12][20][30], int rows);
```

```
int sum4d(int (*ar)[12][20][30], int rows); // ar a pointer
```

```c
// array2d.c -- functions for 2d arrays
#include <stdio.h>
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int );    // ok to omit names
int sum2d(int (*ar)[COLS], int rows); // another syntax
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 all elements = %d\n", sum2d(junk, ROWS));
    return 0;
}
```

```c
void sum_rows(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;
    for (r = 0; r < rows; r++)
    {
        tot = 0;
        for (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)
{
    int r;
    int c;
    int tot;
    for (c = 0; c < COLS; c++)
    {
        tot = 0;
        for (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 r;
    int c;
    int tot = 0;
    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

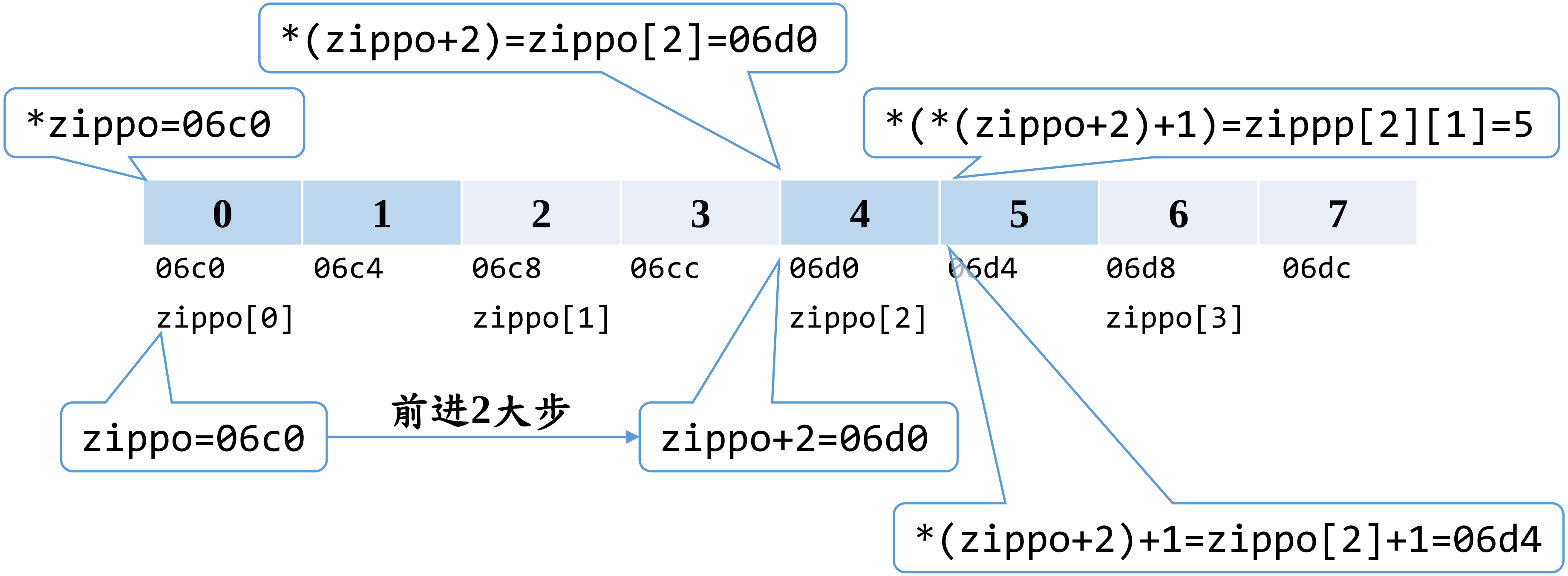
```
row 0: sum = 20
row 1: sum = 24
row 2: sum = 36
col 0: sum = 17
col 1: sum = 19
col 2: sum = 21
col 3: sum = 23
Sum of all elements = 80
```

指针和多维数组

- 多维数组

```
int zippo[4][2];  /* an array of arrays of ints */
```

  – 说明zippo是二维数组



*(zippo+2)=zippo[2]=06d0

*zippo=06c0

((zippo+2)+1)=zippp[2][1]=5

0	1	2	3	4	5	6	7
06c0	06c4	06c8	06cc	06d0	06d4	06d8	06dc
zippo[0]		zippo[1]		zippo[2]		zippo[3]	

zippo=06c0

前进2大步

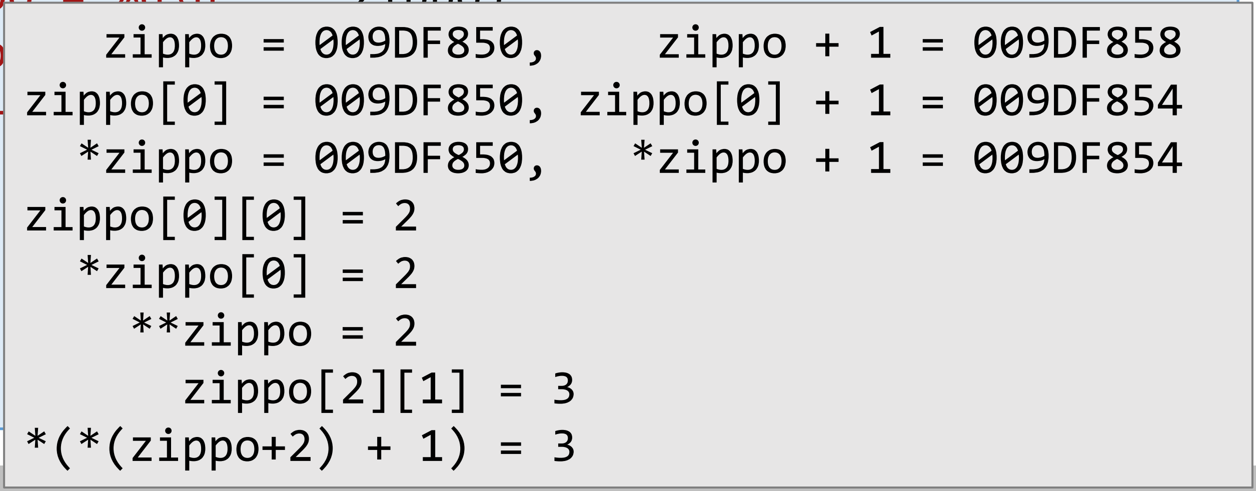
zippo+2=06d0

*(zippo+2)+1=zippo[2]+1=06d4

```c
/* zippo1.c --  zippo info */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };

    printf("   zippo = %p,    zippo + 1 = %p\n", zippo, zippo + 1);
    printf("zippo[0] = %p, zippo[0] + 1 = %p\n", zippo[0],
zippo[0] + 1);
    printf("  *zippo = %p,   *zippo + 1 = %p\n", *zippo, *zippo
+ 1);
    printf("zippo[0][0] = %d\n", zippo[0][0]);
    printf("  *zippo[0] = %d\n", *zippo[0]);
    printf("    **zippo = %d\n", **zippo);
    printf("      zipp
    printf("*(*(zippo+

    return 0;
}
```



```
   zippo = 009DF850,    zippo + 1 = 009DF858
zippo[0] = 009DF850, zippo[0] + 1 = 009DF854
  *zippo = 009DF850,   *zippo + 1 = 009DF854
zippo[0][0] = 2
  *zippo[0] = 2
    **zippo = 2
      zippo[2][1] = 3
*(*(zippo+2) + 1) = 3
```

指向数组的指针

- 如何定义指向多维数组的指针

```
int zippo[4][2];  /* an array of arrays of ints */
```

```
int (*p_zippo)[2];  /* a pointer to the array zippo */
```

```
int *p_zippo;  /* an improper pointer to the array zippo */
```

```
int **p_zippo;  /* an improper pointer to the array zippo */
```

```
int (*p_zippo)[4];  /* a wrong pointer to the array zippo */
```

```
int *p_zippo[2];  /* a wrong pointer to the array zippo */
```

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

```
 pz = 00B0FEE8,    pz + 1 = 00B0FEF0
pz[0] = 00B0FEE8, pz[0] + 1 = 00B0FEEC
  *pz = 00B0FEE8,   *pz + 1 = 00B0FEEC
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

指针的类型兼容性

- 除空指针（void *）外，不同类型指针不能相互转换
  - 不同类型指针需用**强制类型转换**
  - 不同类型指针运算无意义
  - 指向类型不同即为不同
  - 同为二级指针，细节不同

```
int n = 5;
double x;
int * p1 = &n;
double * pd = &x;
x = n; // implicit type conversion
pd = p1; // compile-time error
```

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2; // a pointer to a pointer
pt = &ar1[0][0]; // both pointer-to-int
pt = ar1[0]; // both pointer-to-int
pt = ar1; // not valid
pa = ar1; // both pointer-to-int[3]
pa = ar2; // not valid
p2 = &pt; // both pointer-to-int *
*p2 = ar2[0]; // both pointer-to-int
p2 = ar2; // not valid
```

数据类型与内存存储

- 数据类型：整数（含字符）和浮点数

<table>
<tr><td>HEX</td><td>48</td><td>65</td><td>6c</td><td>6c</td><td>6f</td><td>21</td><td>00</td><td>A3</td></tr>
<tr><td>UCHAR</td><td>72</td><td>101</td><td>108</td><td>108</td><td>111</td><td>33</td><td>0</td><td>163</td></tr>
<tr><td>CHAR</td><td>'H'</td><td>'e'</td><td>'l'</td><td>'l'</td><td>'o'</td><td>'!'</td><td>'\0'</td><td>'\xA3'</td></tr>
<tr><td>INT16</td><td colspan="2">25928</td><td colspan="2">27756</td><td colspan="2">8559</td><td colspan="2">41728</td></tr>
<tr><td>UINT16</td><td colspan="2">25928</td><td colspan="2">27756</td><td colspan="2">8559</td><td colspan="2">-23808</td></tr>
<tr><td>INT32</td><td colspan="4">1819043144</td><td colspan="4">-1560272529</td></tr>
<tr><td>UINT32</td><td colspan="4">1819043144</td><td colspan="4">2734694767</td></tr>
<tr><td>INT64</td><td colspan="8">-6701319483083168440</td></tr>
<tr><td>UINT64</td><td colspan="8">11745424590626383176</td></tr>
<tr><td>FLOAT</td><td colspan="4">1.14314e+027</td><td colspan="4">-6.94597e-018</td></tr>
<tr><td>DOUBLE</td><td colspan="8">-4.23295e-140</td></tr>
</table>

指针的类型兼容性

扩展内容

- 把指针变量赋值给指针常量表明不修改变量

```
int x = 20;
const int y = 23;
int * p1 = &x;
const int * p2 = &y;
const int ** pp2;
p1 = p2; // not safe -- assigning const to non-const
p2 = p1; // valid -- assigning non-const to const
pp2 = &p1; // not safe -- assigning nested pointer types
```

```
main.c:11:4: warning: assignment
discards ‘const’ qualifier from pointer
target type [enabled by default]
```

指针的类型兼容性

扩展内容

- 把指针常量赋值给指针变量是合法的但不推荐
  - const并非不能改变，只能靠自律：不要绕道修改const。

```
main.c:9:5: warning: assignment from incompatible
pointer type [enabled by default]
pp2 = &p1; // allowed, but const qualifier
disregarded
```

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1; // allowed, but const qualifier disregarded
*pp2 = &n; // valid, both const, but sets p1 to point at n
*p1 = 10; // valid, but tries to change const n
```

```
const int y;
const int * p2 = &y;
int * p1;
p1 = p2; // error in C++, possible warning in C
```

6. 指针的应用

利用指针改变函数中变量的值

- 按参数传递
  - 变量名只能在声明语句所在的代码块中使用
  - 参量的值改变，不影响参数值的改变
- 需要修改参量中的值，应利用指针
  - 函数接口传入内存地址
  - 函数内操作内存地址
  - 内存地址的修改是永久修改

```c
/* swap1.c -- first attempt at a swapping function */
#include <stdio.h>
void interchange(int u, int v); /* declare function */
int main(void)
{
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);
    return 0;
}
void interchange(int u, int v)  /* define function  */
{
    int temp;
    temp = u;
    u = v;
    v = temp;
}
```

```
Originally x = 5 and y = 10.
Now x = 5 and y = 10.
```

```c
/* swap2.c -- researching swap1.c */
#include <stdio.h>
void interchange(int u, int v);
int main(void)
{
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);
    return 0;
}
void interchange(int u, int v)
{
    int temp;
    printf("Originally u = %d and v = %d.\n", u , v);
    temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```

```
Originally x = 5 and y = 10.
Originally u = 5 and v = 10.
Now u = 10 and v = 5.
Now x = 5 and y = 10.
```

利用函数交换两个变量的值

- 左侧答案无法交换，右侧答案可以交换

```c
#include <stdio.h>
int swap(int a, int b)
{
    int t;
    t = a; a = b; b = t;
    printf("%d %d\n", a, b);
}
int main()
{
    int a = 3, b = 4;
    printf("%d %d\n", a, b);
    swap(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
```

```c
#include <stdio.h>
int swap(int *a, int *b)
{
    int t;
    t = *a; *a = *b; *b = t;
    printf("%d %d\n", *a, *b);
}
int main()
{
    int a = 3, b = 4;
    printf("%d %d\n", a, b);
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

利用函数交换两个变量的值

- 左侧错误：函数内外，即便同名变量也是不同的函数

环节（左侧）	main中a值	main中a地址	main中b值	main中b地址	swap中a值	swap中a地址	swap中b值	swap中b地址
进入swap之前	3	0x3008	4	0x3004		未开辟		未开辟
刚进入swap	3	0x3008	4	0x3004	3	0x4008	4	0x4004
退出swap之前	3	0x3008	4	0x3004	4	0x4008	3	0x4004
退出swap之后	3	0x3008	4	0x3004		已释放		已释放

环节（右侧）	main中a值	main中a地址	main中b值	main中b地址	swap中a值	swap中a地址	swap中b值	swap中b地址
进入swap之前	3	0x3008	4	0x3004		未开辟		未开辟
刚进入swap	3	0x3008	4	0x3004	0x3008	0x4008	0x3004	0x4004
退出swap之前	4	0x3008	3	0x3004	0x3008	0x4008	0x3004	0x4004
退出swap之后	4	0x3008	3	0x3004		已释放		已释放

指向数组的指针作为函数参量

- 指针作为函数的参量

```
int sum(int ar[], int n)
int sum(int * ar)
```

  - 两种形式是等价的，ar是指针
  - 虽然两种形式等价，但当ar表示数组时，应使用数组形式；当ar表示指针时，应使用指针形式，以免读者误解。

- 函数需要传入数组时，应传入数组长度
  - 只有在声明数组的代码块能通过数组名求出数组长度
  - 在函数中，无法得知外部数组的长度，容易导致越界

```c
// sum_arr1.c -- sums the elements of an array
// use %u or %lu if %zd doesn't work
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);

int main(void) {
    int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
    long answer;

    answer = sum(marbles, SIZE);
    printf("The total number of marbles is %ld.\n", answer);
    printf("The size of marbles is %zd bytes.\n", sizeof
marbles);

    return 0;
}
```

```
int sum(int ar[], int n)      // how big an array?
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++ )
        total += ar[i];
    printf("The size of ar is %zd bytes.\n", sizeof ar);

    return total;
}
```

```
The size of ar is 4 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

函数在参量使用指针操作数组

- 函数的参量传递数组名和长度*N*
  - 参照点是数组的起始位置，范围为0～*N*-1
- 函数的参量传递数组的起始位置*p*和结束位置*q*
  - 参照点是内存的起始位置，范围为*p*~*q*

```c
/* sum_arr2.c -- sums the elements of an array */
#include <stdio.h>
#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 of marbles is %ld.\n", answer);

    return 0;
}
```

```c
/* 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;
}
```

```
The total number of marbles is 190.
```

```c
/* order.c -- precedence in pointer operations */
#include <stdio.h>
int data[2] = {100, 200};
int moredata[2] = {300, 400};
int main(void)
{
    int * p1, * p2, * p3;
    p1 = p2 = data;
    p3 = moredata;
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
           *p1     ,   *p2     ,     *p3);
    printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
           *p1++     , *++p2     , (*p3)++);
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
           *p1     ,   *p2     ,     *p3);
    return 0;
}
```

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100, *++p2 = 200, (*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

指针和数组的区别联系

- 数组是常量，指针是变量
  - 数组名称的值是该数组元素的首地址，不能被赋值
  - 指针可以用数组赋值
- 指针组成表达式的方法与数组相同
  - 指针按偏移量取值的方法*(a+i)与数组取元素a[i]相同

```c
// pnt_add.c -- pointer addition
#include <stdio.h>
#define SIZE 4
int main(void)
{
    short dates [SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;
    pti = dates;    // assign address of array 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;
}
```

```
                 short
double
pointers + 0:   0046F9D0   0046F990
pointers + 1:   0046F9D2   0046F998
pointers + 2:   0046F9D4   0046F9A0
pointers + 3:   0046F9D6   0046F9A8
```

```c
/* day_mon3.c -- uses pointer notation */
#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 %2d has %d days.\n", index +1,
               *(days + index));   // same as days[index]

    return 0;
}
```

```
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
（此处省略数行）
Month 12 has 31 days.
```

指针的应用场景

- 在生命周期存续但作用域之外使用变量
  - 每个变量有其生命周期，在声明的代码段内用变量名访问
  - 在作用域以外使用变量通过变量名无法访问
    - 通过其内存首地址（指针）来定位
- 无法对数组名赋值，因而使用指针替代
- 在栈外开辟和使用较大的内存空间
  - 开辟后的内存空间应赋值给指针后使用

7. 变长数组和符合文字

变长数组（C99标准）

扩展内容

- C99允许声明数组时，长度可以为变量

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters]; // a VLA
```

- 变长数组是声明时用变量表示长度，并非长度可变

```
int sum2d(int rows, int cols, int ar[rows][cols]); // ar a VLA
```

```
int sum2d(int ar[rows][cols], int rows, int cols); // invalid order
```

```
int sum2d(int, int, int ar[*][*]); // ar a VLA, names omitted
```

变长数组（C99标准）

扩展内容

- 不推荐使用变长数组
  - 变长数组本质上是动态分配的数组，建议改用：
    - 开辟空间：指针=malloc(字节数)；释放空间：free(指针)
- 变长数组限制
  - 由于编译时长度未知，声明时不能初始化
  - 必须在程序块的范围内定义，不能在文件范围内定义
  - 作用域和生存时间为块的范围，不能是静态的或者外部的
  - 不能作为结构体或联合体的成员，只能为独立数组形式

```c
//vararr2d.c -- functions using VLAs
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]);
int main(void)
{
    int i, j;
    int rs = 3;
    int cs = 10;
    int junk[ROWS][COLS] = {
        {2,4,6,8},
        {3,5,7,9},
        {12,10,8,6}
    };
    int morejunk[ROWS-1][COLS+2] = {
        {20,30,40,50,60,70},
        {5,6,7,8,9,10}
    };
    int varr[rs][cs];  // VLA
    for (i = 0; i < rs; i++)
        for (j = 0; j < cs; j++)
            varr[i][j] = i * j + j;
    printf("3x5 array\n");
    printf("Sum of all elements = %d\n", sum2d(ROWS, COLS, junk));
    printf("2x6 array\n");
    printf("Sum of all elements = %d\n", sum2d(ROWS-1, COLS+2, morejunk));
    printf("3x10 VLA\n");
    printf("Sum of all elements = %d\n", sum2d(rs, cs, varr));
    return 0;
}
// function with a VLA parameter
int sum2d(int rows, int cols, int ar[rows][cols]) {
    int r, c;
    int tot = 0;
    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];
    return tot;
}
```

复合文字

扩展内容

- 普通数组的声明方法

```
int diva[2] = {10, 20};
```

- 复合文字
  - 没有名称，只能通过赋值给指针（数组可以么）使用

```
(int [2]){10, 20} // a compound literal
(int []){50, 20, 90} // a compound literal with 3 elements
int sum(const int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
int (*pt2)[4]; // declare a pointer to an array of 4-int arrays
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

```c
// flc.c -- funny-looking constants
#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;
}
```

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

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

C程序设计

T10

谢谢

厦门大学信息学院软件工程系

黄炜 副教授