C/C++

数组与指针

张晓平

武汉大学数学与统计学院

Table of contents

数组

数组

- 数组由一系列类型相同的元素构成。
- 数组声明必须包括元素的个数与类型。

```
float candy[365];
char code[12];
int states[50];
```

数组：数组初始化

```
int array[6] = {1, 2, 4, 6, 8, 10};
```

可用初始化列表（花括号括起来的一系列数值，用逗号隔开）对数组进行初始化。

数组：数组初始化

```
// day_mon1.c:
#include <stdio.h>
#define MONTHS 12
int main(void)
{
  int i;
  int days[MONTHS] = {31, 28, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31};
  for (i = 0; i < MONTHS; i++)
    printf("Month %2d has %2d days.\n",
           i+1, days[i]);
  return 0;
}
```

数组：数组初始化

```
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

数组：用 const 初始化数组

```
// day_mon1_const.c:
#include <stdio.h>
#define MONTHS 12
int main(void)
{
  const int days[MONTHS] = {31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31};
  int i;
  for (i = 0; i < MONTHS; i++)
    printf("Month %2d has %2d days.\n",
           i+1, days[i]);
  return 0;
}
```

数组：用 const 初始化数组

若用关键字 `const` 初始化数组，则该数组为只读数组，其元素均为常量，不允许修改。

数组：如果数组没有初始化，...

```c
// no_data.c:
#include <stdio.h>
#define SIZE 6
int main(void)
{
  int no_data[SIZE];
  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：如果数组没有初始化，...

```
i     no_data[i]
0              0
1              0
2              0
3              0
4     1606416376
5          32767
```

数组：如果数组没有初始化，...

同普通变量一样，数组元素的值在初始化之前是不确定的，编译器将使用存储单元中原有的数值。

数组：如果初始化列表的元素个数与数组大小不一致，...

```c
// some_data.c:
#include <stdio.h>
#define SIZE 6
int main(void)
{
  int some_data[SIZE] = {11, 12};
  int i;
  printf("%2s%14s\n", "i", "no_data[i]");
  for (i = 0; i < SIZE; i++)
    printf("%2d%14d\n", i, some_data[i]);
  return 0;
}
```

数组：如果初始化列表的元素个数与数组大小不一致，...

```
i      no_data[i]
0              11
1              12
2               0
3               0
4               0
5               0
```

数组：如果初始化列表的元素个数与数组大小不一致，...

- 若不初始化数组，则数组元素存储的是无用的数值；
- 若部分初始化数组，则未初始化的元素会被设置为 0；
- 若初始化列表中元素的个数大于数组大小，则编译器会警告或报错。

数组：如果初始化列表的元素个数与数组大小不一致，...

可以省略括号中的数字，让编译器自动匹配数组大小与初始化列表中的项目个数。

数组：如果初始化列表的元素个数与数组大小不一致，...

```
#include <stdio.h>
#define MONTHS 12
int main(void)
{
  int i;
  const int days[] = {31, 28, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31};
  for (i = 0; i < sizeof days/sizeof days[0]; i++)
    printf("Month %2d has %2d days.\n",
           i+1, days[i]);
  return 0;
}
```

数组：如果初始化列表的元素个数与数组大小不一致，...

数组中元素个数等于整个数组的大小除以单个元素的大小，其中 `sizeof days` 用于计算整个数组的大小，`sizeof days[0]` 用于计算一个元素的大小，均以字节为单位。

数组：指定初始化项目（C99）

若想对数组的最后一个元素初始化，

- 对于传统 C，需要对每一个元素都初始化之后，才能对某个元素进行初始化。

```
int arr[6] = {0, 0, 0, 0, 0, 212};
```

数组：指定初始化项目（C99）

若想对数组的最后一个元素初始化，

- 对于传统 C，需要对每一个元素都初始化之后，才能对某个元素进行初始化。

```
int arr[6] = {0, 0, 0, 0, 0, 212};
```

- 而对 C99，在初始化列表中使用带方括号的元素下标可指定某个特定的元素：

```
int arr[6] = {[5] = 212}; //set arr[5] to 212
```

数组：指定初始化项目（C99）

```c
// designate.c
#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("Month %2d has %2d days.\n",
           i+1, days[i]);
  return 0;
}
```

数组：指定初始化项目（C99）

```
Month  1 has 31 days.
Month  2 has 29 days.
Month  3 has  0 days.
Month  4 has  0 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has  0 days.
Month  9 has  0 days.
Month 10 has  0 days.
Month 11 has  0 days.
Month 12 has  0 days.
```

数组：指定初始化项目（C99）

数组的初始化规则

- 对于普通初始化，在初始化一个或多个元素后，未初始化元素都将被设置为 0。
- 若在一个指定初始化项目后有不止一个值，则这些值将对后续的数组元素初始化。
- 若多次对一个元素进行初始化，则最后一次有效。

数组：为数组赋值

- 声明完数组后，可用下标对数组成员赋值。
- C 不支持把数组当做一个整体来进行赋值，也不支持用花括号括起来的列表形式进行赋值（初始化时除外）。

数组：为数组赋值

```
int main(void)
{
  int arr1[5] = {1, 2, 3, 4};
  int arr2[5];

  arr2 = arr1;              // invalid
  arr2[5] = arr1[5];        // OK
  arr2[5] = {1, 2, 3, 4}; // invalid
}
```

数组：数组边界

使用数组时，下标不能超过数组的边界。例如，假设你有这样的声明：

```
int arr[20];
```

那么你在使用下标时，要确保其范围在 0 到 19 之间，编译器不会检查这种错误。

数组：数组边界 i

```
// bounds.c
#include <stdio.h>
#define SIZE 4
int main(void)
{
  int value1 = 14, value2 = 88;
  int arr[SIZE];
  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",
```

数组：数组边界 ii

```
          value1, value2);

  return 0;
}
```

数组：数组边界

```
value1 = 14, value2 = 88
-1 -1
 0 1
 1 3
 2 5
 3 7
 4 9
 5 32767
 6 424094912
value1 = -1, value2 = 9
```

使用超过数组边界的下标可能会改变其他变量的值。

数组：指定数组大小

```
int n = 5;
int m = 8;
float a1[5];                    //OK
float a2[5*2 + 1];              //OK
float a3[sizeof(int) + 1];//OK
float a4[-1];                   //Invalid
float a5[0];                    //Invalid
float a6[2.5];                  //Invalid
float a7[(int) 2.5];            //OK, float to int
float a8[n];                    //C99 OK
float a9[m];                    //C99 OK
```

多维数组

多维数组

编制程序，计算出年降水总量、年降水平均量，以及月降水平均量。

多维数组 i

```
/* 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 2000 - 2004
  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;
```

多维数组 ii

```
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", 2000 + year, subtot);
  total += subtot; // total for all years
}
printf("\nThe yearly average is %.1f inches.\n\n",
       total/YEARS);

printf("MONTHLY AVERAGES:\n");
printf("  Jan   Feb   Mar   Apr   May   Jun");
printf("  Jul   Aug   Sep   Oct   Nov   Dec\n");
for (month = 0; month < MONTHS; month++)
{ // for each month, sum
  for (year = 0, subtot =0; year < YEARS; year++)
```

多维数组 iii

```
      subtot += rain[year][month];
    printf(" %4.1f", subtot/YEARS);
  }
  printf("\n");

  return 0;
}
```

多维数组

```
YEAR RAINFALL (inches)
 2000             32.4
 2001             37.9
 2002             49.8
 2003             44.0
 2004             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
```

多维数组：初始化二维数组

回忆一下一维数组的初始化：

```
sometype ar1[5] = {val1, val2, val3, val4, val5
};
```

多维数组：初始化二维数组

对于二维数组 `rain[5][12]`，

- 它是包含 5 个元素的数组，而每个元素又是包含 12 个 `float`数的数组。
- 对某个元素初始化，即用一个初始化列表对一个一维 float 数组进行初始化。
- 要对整个二维数组初始化，可以采用逗号隔开的 5 个初始化列表进行初始化。

多维数组：初始化二维数组

```
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}
};
```

多维数组：初始化二维数组

- 用了 5 个数值列表，都用花括号括起来。
- 第一个列表赋给第一行，第二个列表赋给第二行，依此类推。
- 若第一个列表只有 10 个数值，则第一行前 10 个元素得以赋值，最后两个元素被设置为 0。
- 若列表中数值个数多于 12 个，则会被警告或报错；这些数值不会影响到下一行的赋值。

多维数组：初始化二维数组

初始化时，也可省略内部的花括号。

- 只要保证数值的个数正确，初始化效果就是一样。
- 如果数值个数不够，则会按照先后顺序来逐行赋值，未被赋值的元素会被初始化为 0。

多维数组：更多维数的数组

三维数组的声明方式：

```
int box[10][20][30];
```

- 直观理解：一维数组是排成一行的数据，二维数组是放在一个平面上的数据，三维数组是把平面数据一层一层地叠起来。
- 另一种理解：三维数组是数组的数组的数组。即：box 是包含 10 个元素的数组，其中每个元素又是包含 20 个元素的数组，这 20 个元素的每一个又是包含 30 个元素的数组。

指针与数组

指针与数组

- 计算机的硬件指令很大程度上依赖于地址，而指针为你使用地址提供了一种方法。
- 于是，使用指针使你能够以类似于计算机底层的方式来表达你的意愿，从而让程序能更高效地工作。
- 特别地，指针能很有效的处理数组，实际上数组是一种变相使用指针的形式。

指针与数组

请记住：**数组名是数组首元素的地址。**

指针与数组

请记住：**数组名是数组首元素的地址。**

若 `array` 为一个数组，则以下关系式为真：

```
array == &array[0];
```

指针与数组 i

```
// 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 %14s\n", "short", "double");

  for (index = 0; index < SIZE; index ++)
    printf("pointers + %d: %14p %14p\n",
           index, pti + index, ptf + index);

  return 0;
}
```

指针与数组

```
                        short           double
pointers + 0: 0x7fff5fbff7d0 0x7fff5fbff7b0
pointers + 1: 0x7fff5fbff7d2 0x7fff5fbff7b8
pointers + 2: 0x7fff5fbff7d4 0x7fff5fbff7c0
pointers + 3: 0x7fff5fbff7d6 0x7fff5fbff7c8
```

指针与数组

在 C 中，对指针加 1 的结果是对该指针增加一个存储单元。对数组而言，地址会增加到下一个元素的地址，而不是下一个字节。

指针定义小结

- 指针的数值就是它所指向的对象的地址。地址的内部表达方式由硬件决定，很多计算机都是以字节编址的。
- 在指针前用运算符 * 就可以得到该指针所指向的对象的值。
- 对指针加 1，等价于对指针的值加上它所指向的对象的字节大小。

指针与数组

```
dates + 2 == &dates[2];      // 相同的地址
*(dates + 2) == dates[2];    // 相同的值
```

可以用指针标识数组的每个元素，并得到每个元素的值。从本质上讲，这是对同一对象采用了两种不同的符号表示方法。

指针与数组

在描述数组时，C 确实借助了指针的概念。例如，定义 `array[n]` 时，

- 即：`*(array + n)`，
- 含义：“寻址到内存中的 `array`，然后移动 `n` 个单元，再取出数值”。

指针与数组

请注意 *(dates+2) 和 *dates+2 的区别。**取值运算符 * 的优先级高于 +**，故后者等价于 (*dates)+2。

```
*(dates + 2)   // dates 的第三个元素的值
*dates + 2     // dates 的第一个元素与 2 相加
```

指针与数组

```c
/* day_mon3.c -- uses pointer notation */
#include <stdio.h>
#define MONTHS 12
int main(void)
{
  int index;
  int days[MONTHS] = {31,28,31,30,31,30,
                      31,31,30,31,30,31};

  for (index = 0; index < MONTHS; index++)
    printf("Month %2d has %d days.\n",
           index +1, *(days + index));
                     // same as days[index]
  return 0;
}
```

函数、数组与指针

函数、数组与指针

编写函数，求一个数组的各元素之和。

函数、数组与指针

方式一：在函数中给定固定的数组大小。

```
int sum(int * ar)
{
  int i;
  int total = 0;

  for (i = 0; i < 10; i++)
    total += ar[i];

  return total;
}
```

函数、数组与指针

方式一：在函数中给定固定的数组大小。

```
int sum(int * ar)
{
  int i;
  int total = 0;

  for (i = 0; i < 10; i++)
    total += ar[i];

  return total;
}
```

但该函数仅在数组长度为 10 时可工作。

方式二：将数组大小作为参数传递给函数。

```
int sum(int * ar, int n)
{
  int i;
  int total = 0;

  for (i = 0; i < n; i++)
    total += ar[i];

  return total;
}
```

方式二：将数组大小作为参数传递给函数。

```
int sum(int * ar, int n)
{
  int i;
  int total = 0;

  for (i = 0; i < n; i++)
    total += ar[i];

  return total;
}
```

该方式更为灵活，第一个参数把数组地址和数组类型的信息传递给函数，第二个参数把数组的元素个数传递给函数。

函数、数组与指针

在做函数声明时，以下四种函数原型是等价的：

```
int sum(int * ar, int n);
int sum(int *, int);

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

函数、数组与指针

在定义函数时，名称不可以省略。故在定义时以下两种形式是等价的：

```
int sum(int * ar, int n)
{
   ...
}

int sum(int ar[], int n)
{
   ...
}
```

观察以下程序，其功能是同时打印原数组的大小和代表数组的函数参量的大小。

```
// sum_arr1.c -- sums the elements of an array
#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 %lu bytes.\n",
  sizeof marbles);
  return 0;
}

int sum(int *ar, int n)
{
  int i;
  int total = 0;

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

函数、数组与指针 iii

```
    return total;
}
```

```
The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

函数、数组与指针

```
The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

- marbles 的大小为 40 字节，因 marbles 包含 10 个 int 数，每个数占 4 个字节。
- ar 的大小为 8 个字节，因 ar 本身并不是一个数组，而是一个指向 marbles 首元素的指针。

函数、数组与指针：使用指针参数

以上程序中，sum() 使用一个指针参量来确定数组的开始，使用一个整数参量来指明数组的元素个数。但这并不是向函数传递数组信息的唯一办法。

函数、数组与指针：使用指针参数

另一种办法是传递两个指针，第一个指针指明数组的起始地址，第二个指针指明数组的结束地址。

函数、数组与指针：使用指针参数 i

```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;
}
```

函数、数组与指针：使用指针参数 ii

```
/* use pointer arithmetic */
int sump(int * start, int * end)
{
  int total = 0;
  while (start < end)
  {
    total += *start;
    start++;
  }
  return total;
}
```

函数、数组与指针：使用指针参数

- 指针 `start` 最初指向 `marbles` 的首元素，执行赋值表达式 `total += *start` 时，把首元素的值加到 `total` 上。
- 然后表达式 `start++` 使指针变量 `start` 加 1，指向数组的下一个元素。

函数、数组与指针：使用指针参数

函数 sump() 使用第二个指针来控制循环次数：

```
while (start < end)
```

因这是对一个不相等关系的判断，故处理的最后一个元素将是 end 所指向的位置之前的元素。这就意味着 end 实际指向的元素是在数组最后一个元素之后。

函数、数组与指针：使用指针参数

函数 sump() 使用第二个指针来控制循环次数：

```
while (start < end)
```

因这是对一个不相等关系的判断，故处理的最后一个元素将是 end 所指向的位置之前的元素。这就意味着 end 实际指向的元素是在数组最后一个元素之后。

C 保证在为数组分配存储空间时，指向数组之后的第一个位置的指针也是合法的。

函数、数组与指针：使用指针参数

函数 sump() 使用第二个指针来控制循环次数：

```
while (start < end)
```

因这是对一个不相等关系的判断，故处理的最后一个元素将是 end 所指向的位置之前的元素。这就意味着 end 实际指向的元素是在数组最后一个元素之后。

C 保证在为数组分配存储空间时，指向数组之后的第一个位置的指针也是合法的。

使用这种“越界”指针可使函数调用的形式更为整洁：

```
answer = sump(marbles, marbles + SIZE);
```

函数、数组与指针：使用指针参数

如果让 end 指向最后一个元素，函数调用的形式则为

```
answer = sump(marbles, marbles + SIZE - 1);
```

这种写法不仅看起来不整洁，也不容易被记住，从而容易导致编程错误。

函数、数组与指针：使用指针参数

如果让 end 指向最后一个元素，函数调用的形式则为

```
answer = sump(marbles, marbles + SIZE - 1);
```

这种写法不仅看起来不整洁，也不容易被记住，从而容易导致编程错误。

请记住：虽然 C 保证指针 marbles+SIZE 是合法的，但对该地址存储的内容 marbles[SIZE] 不作任何保证。

函数、数组与指针：使用指针参数

```
total += *start;
start++;
```

$\Longleftrightarrow$

```
total += *start++;
```

- `*` 和 `++` 优先级相同，从右往左结合。故 `++` 应用于 `start`，而不是应用于 `*start`。也就是说，是指针自增 1，而不是指针所指向的数据自增 1。

函数、数组与指针：使用指针参数

```
total += *start;
start++;
```

$\Longleftrightarrow$

```
total += *start++;
```

- `*` 和 `++` 优先级相同，从右往左结合。故 `++` 应用于 `start`，而不是应用于 `*start`。也就是说，是指针自增 1，而不是指针所指向的数据自增 1。
- 后缀形式 `total += *start++` 表示先把指针指向的数据加到 `total` 上，然后指针再自增 1。

函数、数组与指针：使用指针参数

```
total += *start;
start++;
```

$\Longleftrightarrow$

```
total += *start++;
```

- `*` 和 `++` 优先级相同，从右往左结合。故 `++` 应用于 `start`，而不是应用于 `*start`。也就是说，是指针自增 1，而不是指针所指向的数据自增 1。
- 后缀形式 `total += *start++` 表示先把指针指向的数据加到 `total` 上，然后指针再自增 1。
- 前缀形式 `total += *++start` 表示指针先自增 1，然后再把其指向的数据加到 `total` 上。

函数、数组与指针：使用指针参数

```
total += *start;
start++;
```

$\Longleftrightarrow$

```
total += *start++;
```

- `*` 和 `++` 优先级相同，从右往左结合。故 `++` 应用于 `start`，而不是应用于 `*start`。也就是说，是指针自增 1，而不是指针所指向的数据自增 1。
- 后缀形式 `total += *start++` 表示先把指针指向的数据加到 `total` 上，然后指针再自增 1。
- 前缀形式 `total += *++start` 表示指针先自增 1，然后再把其指向的数据加到 `total` 上。
- 若使用 `(*start)++`，则会使用 `start` 指向的数据，然后再使该数据自增 1，而不是使指针自增 1。此时，指针所指向的地址不变，但其中的元素会更新。

函数、数组与指针：关于优先级 i

```
// 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);
```

函数、数组与指针：关于优先级 ii

```
  printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n
  ", *p1++ , *++p2 , (*p3)++);
  printf("*p1 = %d, *p2 = %d, *p3 = %d\n", *p1,
  *p2, *p3);

  return 0;
}
```

函数、数组与指针：关于优先级

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

函数、数组与指针：小结

- 处理数组的函数实际上使用指针作为参数，不过在编写此类函数时，数组符号与指针符号都可以使用。若使用数组符号，则函数处理数组这一事实更加明显。
- 在 C 中，以下两个表达式等价

  `ar[i]` $\Longleftrightarrow$ `*(ar+i)`

  不管 `ar` 是一个数组名还是一个指针变量，这两个表达式都可以工作。然而只有当 `ar` 是一个指针变量时，才可以使用 `ar++` 这样的表达式。
- 指针符号更接近机器语言，并且某些编译器在编译时能生成效率更高的代码。

指针操作

指针操作

```
int urn[5] = {100,200,300,400,500};
int * ptr, * ptr1, * ptr2;
```

指针操作

1 赋值 (assignment)：可以把一个地址赋给指针。通常使用数组名或地址运算符 & 来进行地址赋值。如

```
ptr1 = urn;
ptr2 = &urn[2];
```

注意：地址应该与指针类型兼容，不能把一个 double 类型的地址赋给一个指向 int 的指针。

指针操作

2 取值（dereferencing）：运算符 * 可取出指针所指向的地址中存储的数据。

3 取指针地址：指针变量也有地址和数值，使用 & 运算符可以得到存储指针本身的地址。

指针操作

4 指针加上一个整数：该整数会和指针所指类型的字节数相乘，然后所得结果会加到初始地址上。

例如，若 `ptr = urn`，则 `ptr + 4` 的结果等同于 `&urn[4]`。

若相加结果越界，则结果不确定，除非超出数组最后一个元素的地址能确保有效。

5 增加指针的值：指针可以自增。

例如，若 `ptr = &urn[2]`，则执行 `ptr++`后，`ptr` 指向 `urn[3]`。

指针操作

6 指针减去一个整数：该整数会和指针所指类型的字节数相乘，然后所得结果会从初始地址中减掉。

例如，若 `ptr = &urn[4]`，则 `ptr - 2` 的结果等同于 `&urn[2]`。

7 减小指针的值：指针可以自减。

例如，若 `ptr = &urn[4]`，则执行 `ptr--` 后，`ptr` 指向 `urn[3]`。

指针操作

8 求差值：可求出两个指针间的差值。通常对分别指向同一个数组内两个元素的指针求差值，以求出元素之间的距离。

例如，若 `ptr1 = &urn[2]`，`ptr2 = &urn[4]`，则 `ptr2 - ptr1` 的值为 2。

有效指针差值运算的前提是参与运算的两个指针必须指向同一个数组

9 比较：可以使用关系运算符来比较两个指针的值，前提是两个指针具有相同的类型。

指针操作 i

```
// 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;
  ptr2 = &urn[2];

  printf("pointer value, dereferenced pointer, pointer
   address:\n");
  printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
         ptr1, *ptr1, &ptr1);

  ptr3 = ptr1 + 4;
```

指针操作 ii

```
printf("\nadding an int to a pointer:\n");
printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
       ptr1 + 4, *(ptr1 + 3));

ptr1++;
printf("\nvalues after ptr1++:\n");
printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
       ptr1, *ptr1, &ptr1);

ptr2--;
printf("\nvalues after --ptr2:\n");
printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
       ptr2, *ptr2, &ptr2);

--ptr1;
++ptr2;
```

指针操作 iii

```
  printf("\nPointers reset to original values:\n");
  printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);

  printf("\nsubtracting one pointer from another:\n");
  printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %ld\n",
         ptr2, ptr1, ptr2 - ptr1);
  printf("\nsubtracting an int from a pointer:\n");
  printf("ptr3 = %p, ptr3 - 2 = %p\n",
         ptr3, ptr3 - 2);

  return 0;
}
```

指针操作 i

```
pointer value, dereferenced pointer, pointer address:
ptr1 = 0x7fff5fbff7c0, *ptr1 =100, &ptr1 = 0
x7fff5fbff7b0

adding an int to a pointer:
ptr1 + 4 = 0x7fff5fbff7d0, *(ptr4 + 3) = 400

values after ptr1++:
ptr1 = 0x7fff5fbff7c4, *ptr1 =200, &ptr1 = 0
x7fff5fbff7b0

values after --ptr2:
ptr2 = 0x7fff5fbff7c4, *ptr2 = 200, &ptr2 = 0
x7fff5fbff7a8

Pointers reset to original values:
```

指针操作 ii

```
ptr1 = 0x7fff5fbff7c0, ptr2 = 0x7fff5fbff7c8

subtracting one pointer from another:
ptr2 = 0x7fff5fbff7c8, ptr1 = 0x7fff5fbff7c0, ptr2 -
ptr1 = 2

subtracting an int from a pointer:
ptr3 = 0x7fff5fbff7d0, ptr3 - 2 = 0x7fff5fbff7c8
```

指针操作：小结

- 关于指针，有两种形式的减法：可以用一个指针减去另一个指针得到一个整数，也可以从一个指针中减去一个整数得到一个指针。
- 当指针做自增和自减运算时，计算机并不检查指针是否仍然指向某个数组元素。C 保证指向数组元素的指针和指向数组后的第一个地址的指针是有效的。
- 可以对指向一个数组元素的指针进行取值运算，但不能对指向数组后的第一个地址的指针进行取值运算，尽管这样的指针是合法的。

指针操作：小结

使用指针时，不能对未初始化的指针取值。如

```
int  *pt;     //未初始化的指针
*pt  =  5;    //可怕的错误
```

指针操作：小结

使用指针时，不能对未初始化的指针取值。如

```
int  *pt;     //未初始化的指针
*pt  =  5;    //可怕的错误
```

`*pt = 5` 把数值 5 存储在 `pt` 所指向的地址，但由于 `pt` 未被初始化，它的值是随机的，这意味着不确定 5 会被存储到什么位置。这个位置也许对系统危害不大，但也许会覆盖程序数据或代码，甚至导致程序的崩溃。

指针操作：小结

切记：当创建一个指针时，系统只分配了用来存储指针本身的内存空间，并不分配用来存储数据的内存空间。因此在使用指针之前，必须给它赋予一个已分配的内存地址。

指针操作：小结

切记：当创建一个指针时，系统只分配了用来存储指针本身的内存空间，并不分配用来存储数据的内存空间。因此在使用指针之前，必须给它赋予一个已分配的内存地址。

- 可以把一个已存在的变量地址赋给指针；
- 使用 `malloc()` 来首先分配内存。

指针操作：小结

给定如下声明

```
int urn[3];
int * ptr1, * ptr2;
```

下标列出了一些合法的和非法的语句

合法	非法
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

保护数组内容

保护数组内容

在编写处理诸如 `int` 这样的基本类型的函数时，可以向函数传递 `int` 数值，也可以传递指向 `int` 的指针。

- 通常是直接传递数值；
- 只有需要在函数中修改该值时，才传递指针。

保护数组内容

对于处理数组的函数，只能传递指针，原因是这样能使程序效率更高。

- 若通过值向函数传递数组，那么函数中必须分配足够存放一个原数组的拷贝的存储空间，然后把原数组的所有数据复制到这个新数组中；
- 若简单地把数组的地址传递给函数，然后让函数直接读写原数组，程序效率会更高。

保护数组内容

传值仅使用原始数据的一份拷贝，这可以保证原数组不会被意外修改；而传址使得函数可以直接操作原始数据，从而能修改原数组。

保护数组内容

需要修改原数组的例子

以下函数的功能是给数组的每个元素加上同一个数值。

```
void add_to(double arr[], int n, double val)
{
  int i;
  for (i = 0; i < n; i++)
    arr[i] += val;
}
```

保护数组内容

需要修改原数组的例子

以下函数的功能是给数组的每个元素加上同一个数值。

```
void add_to(double arr[], int n, double val)
{
  int i;
  for (i = 0; i < n; i++)
    arr[i] += val;
}
```

该函数改变了数组的内容。之所以可以改变数组内容，是因为函数使用了指针，从而能够直接使用原始数据。

保护数组内容

不希望修改数据的例子

以下函数的功能是计算数组中所有元素的和，故该函数不希望数组的内容。但由于 ar 实际上是一个指针，故编程的错误可能导致原始数据遭到破坏。如表达式 arr[i]++ 会导致每个元素的值增加 1。

```
void sum(int arr[], int n)
{
  int i;
  int sum = 0;
  for (i = 0; i < n; i++)
    sum += arr[i]++;
}
```

保护数组内容：对形参使用 const

在 ANSI C 中，若设计意图是函数不改变数组的内容，那么可以在函数原型和定义中的形参声明中使用关键字 const。如

```
void sum(const int arr[], int n); // 原型

void sum(const int arr[], int n)  // 定义
{
  int i;
  int sum = 0;
  for (i = 0; i < n; i++)
    sum += arr[i];
}
```

这将告知编译器：函数应该把 arr 所指向的数组作为包含常量数据的数组看待。如果你意外的使用了诸如 arr[i]++ 之类的表达式，编译器将会发现这个错误并报告之，通知你函数试图修改常量。

保护数组内容：对形参使用 const

- 使用 `const` 并不要求原始数组固定不变，只是说明函数在处理数组时，应把数组当做是常量数组。
- 使用 `const` 可以对数组提供保护，可阻止函数修改调用函数中的数据。
- 如果函数想修改数组，那么在声明数组参量时就不要使用 `const`；如果函数不需要修改数组，那么在声明数组参量时最好使用 `const`。

保护数组内容：对形参使用 const i

```
/* 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);
```

保护数组内容：对形参使用 const ii

```
  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');
}
```

保护数组内容：对形参使用 const iii

```
/* 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;
}
```

保护数组内容：对形参使用 const

```
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
```

保护数组内容：有关 const 的其它内容

1. 使用 const 创建符号常量：

```
const double PI = 3.1415926;
```

也可用 #define 指令实现：

```
#define PI 3.1415926
```

保护数组内容：有关 const 的其它内容

2. 使用 `const` 还可创建数组常量、指针常量以及指向常量的指针。

保护数组内容：有关 const 的其它内容

以下代码使用 const 保护数组

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,
                          31,31,30,31,30,31};
...
days[9] = 44;    //编译错误
```

保护数组内容：有关 const 的其它内容

指向常量的指针不能用于修改数值。

```
double rates[4] = {8.9, 10.1, 9.4, 3.2};
const double * pd = rates;   //pd 指向数组开始处
*pd = 29.89;                 //不允许
pd[2] = 222.22;              //不允许
rates[0] = 99.99;            //允许，因为 rates 不是常量
pd++;                        //允许，让 pd 指向 rates[1]
```

保护数组内容：有关 const 的其它内容

通常把指向常量的指针用作函数参量，以表明函数不会使用这个指针来修改数据。如

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

保护数组内容：有关 const 的其它内容

关于指针赋值和 const 的一些规则

(a) 允许将常量或非常量数据的地址赋给指向常量的指针。

```
double rates[4] = {8.9, 10.1, 9.4, 3.2};
const double locked[4] = {0.8, 0.7, 0.2,
0.3};
const double * pc = rates;   //合法
pc = locked;                 //合法
pc = &rates[3];              //合法
```

保护数组内容：有关 const 的其它内容

关于指针赋值和 const 的一些规则

(b) 只有非常量数据的地址才可以赋给普通指针。

```
double rates[4] = {8.9, 10.1, 9.4, 3.2};
const double locked[4] = {0.8, 0.7, 0.2,
0.3};
double * pnc = rates;          //合法
pnc = locked;                   //非法
pnc = &rates[3];                //合法
```

这个规则是合理的，否则你就可以使用指针来修改被认为是常量的数据。

保护数组内容：有关 const 的其它内容

规则的应用

像 show_array 这样的函数，可以接受普通数组和常量数组的名称作为实参：

```
show_array(rates, 4);      //合法
show_array(locked, 4);     //合法
```

像 mult_array 这样的函数，不能接受常量数组的名称作为参数：

```
mult_array(rates, 4);      //合法
mult_array(locked, 4);     //不允许
```

因此，**在函数参量定义中使用 const，不仅可以保护数据，而且使函数可以使用声明为 const 的数组。**

保护数组内容：有关 const 的其它内容

3. 使用 const 来声明并初始化指针，以保证指针不会指向别处。

```
double rates[4] = {8.9, 10.1, 9.4, 3.2};
double const * pc = rates;   //pc 指向数组的开始处
pc = &rates[3];              //非法
*pc = 2.2;                   //合法，允许修改
```

rates[0] 的值

这样的指针可用于修改数据，但它所指向的地址不能改变。

保护数组内容：有关 const 的其它内容

4. 可使用两个 const 来创建指针，该指针既不可以更改所指向的地址，也不可以修改所指向的数据。

```
double rates[4] = {8.9, 10.1, 9.4, 3.2};
const double const * pc = rates;
pc = &rates[3];                          //非法
*pc = 2.2;                               //非法
```

指针与多维数组

指针与多维数组

本节主要介绍指针与多维数组的关系，以及为什么要知道它们之间的关系。

必须知道，函数通过指针来处理多维数组。

指针与多维数组

```
int  zippo[4][2];     //数组的数组
```

zippo 是数组首元素的地址，而 zippo 的首元素本身又是包含两个 int 的子数组，故 zippo 也是这个子数组（包含两个 int）的地址。

指针与多维数组

zippo 是数组首元素的地址，故 zippo == &zippo[0]。

zippo[0] 为包含两个 int 的数组，故 zippo[0] == &zippo[0][0]。

zippo 与 zippo[0] 开始于同一个地址，故 zippo == zippo[0]。

指针与多维数组

对一个指针加 1，是加上一个对应类型大小的数值。从这方面来看，zippo 和 zipp[0] 不同：

zippo 所指向的对象是两个 int，而 zipp[0] 所指向的对象是一个 int。

故 zippo+1 和 zippo[0]+1的结果不同。

指针与多维数组

对一个指针取值，得到的是改指针所指向对象的数值。

`zippo[0]` 是其首元素 `zippo[0][0]` 的地址，故

```
*(zippo[0]) == zippo[0][0]
```

即一个 `int` 值。

指针与多维数组

zippo 为其首元素 zippo[0] 的地址，故

```
    *zippo == zippo[0]
```

而

```
    zippo[0] == &zippo[0][0]
```

故

```
  *zippo == &zippo[0][0]
```

又因为

```
  *&zippo[0][0] == zippo[0][0]
```

从而有

```
  **zippo == zippo[0][0]
```

指针与多维数组

简而言之，`zippo` 是地址的地址，需要两次取值才可以得到通常的数值。地址的地址或指针的指针是双重间接的典型例子。

显然，增加数组维数会增加指针的复杂度。

指针与多维数组 i

```
/* 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]);
```

指针与多维数组 ii

```
  printf("**zippo        = %d\n", **zippo);
  printf("  zippo[2][1] = %d\n",   zippo[2][1]);

  printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo+2) + 1)
  );

  return 0;
}
```

指针与多维数组

```
  zippo     = 5fbff7b0,  zippo     + 1 = 5fbff7b8
  zippo[0]  = 5fbff7b0,  zippo[0]  + 1 = 5fbff7b4
 *zippo     = 5fbff7b0, *zippo     + 1 = 5fbff7b4
  zippo[0][0] = 2
 *zippo[0]    = 2
**zippo       = 2
  zippo[2][1] = 3
*(*(zippo+2) + 1) = 3
```

对 zippo 加 1 导致其值加 8，而对 zippo[0] 加 1 导致其值加 4。

表 1: 分析 `*(*(zippo+2)+1)`

`zippo`	`zippo[0]` 的地址
`zippo+2`	`zippo[2]` 的地址
`*(zippo+2)`	`zippo[2]`，也是 `zippo[2]` 首元素的地址
`*(zippo+2)+1`	数组 `zippo[2]` 的第 2 个元素的地址
`*(*(zippo+2)+1)`	数组 `zippo[2]` 的第 2 个元素

指针与多维数组

这里使用指针符号显示数据的意图并不是为了说明可以用它替代更简单的 `zippo[2][1]`，而是要说明当使用一个指向二维数组的指针进行取值时，最好不要使用指针符号，而应使用形式更简单的数组符号。

指向多维数组的指针

本小节着重回答：如何声明指向二维数组的指针 `pz`？

指向多维数组的指针

正确的声明方式为

```
int (* pz) [2];
```

该语句表明 pz 是指向包含两个 int 的数组的指针。**圆括号必不可少。**

指向多维数组的指针

若没有圆括号，

```
int * pax[2];
```

则

- 首先，方括号与 pax 结合，表示 pax 是包含两个某种元素的数组。
- 然后，和 * 结合，表示 pax 是两个指针组成的数组。
- 最后，用 int 来定义，表示 pax 是两个指向 int 的指针构成的数组。

这种声明会创建两个指向单个 int 的指针。

指向多维数组的指针 i

```
/* zippo2.c -- zippo info */
#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]);
```

指向多维数组的指针 ii

```
  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;
}
```

指向多维数组的指针

```
  pz     = 5fbff7b0,   pz     + 1 = 5fbff7b8
  pz[0]  = 5fbff7b0,   pz[0]  + 1 = 5fbff7b4
 *pz     = 5fbff7b0, *pz     + 1 = 5fbff7b4
  pz[0][0] = 2
 *pz[0]    = 2
**pz       = 2
  pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

指向多维数组的指针

尽管 pz 是一个指针，不是数组名，但仍可用 pz[2][1] 之类的数组符号。

要表示单个元素，可以使用数组符号或指针符号，并且在这两种表示中既可用数组名，也可用指针：

```
zippo[m][n] == *(*(zippo+m)+n)
pz[m][n] == *(*(pz+m)+n)
```

指针兼容性

指针之间的赋值规则比数值类型的赋值更严格。例如，可以不进行类型转换就直接把一个 int 数值赋给一个 double 变量。但对指针来说，这样的赋值绝不允许：

```
int n = 5;
double x;
int * pi = &n;
double * pd = &x;
x = n;                  // 隐藏的类型转换
pd = pi;                // 编译时错误
```

指针兼容性

假设有如下声明：

```
int * pt;
int (* pa)[3];
int ar1[2][3];
int ar2[3][2];
int ** p2;              // 指向指针的指针
```

则有如下结论：

```
pt = &ar1[0][0];    // 都指向 int
pt = ar1[0];        // 都指向 int
pa = ar1;           // 都指向 int[3]
p2 = &pt;           // 都指向 int *
```

指针兼容性

```
 pt  =  ar1;            //  非法
 pa  =  ar2;            //  非法
*p2  =  ar2[0];         //  都指向 int
 p2  =  ar2;            //  非法
```

- pt 指向 int，但 ar1 指向由 3 个 int 构成的数组；
- pa 指向由 3 个 int 构成的数组，而 ar2 指向由 2 个 int 构成的数组；
- p2 是指向 int 的指针的指针，而 ar2 是指向由 2 个 int 构成的数组的指针，两种类型不一致；
- *p2 为指向 int 的指针，它与 ar2[0] 是兼容的，因为 ar2[0] 指向 ar2[0][0]，而 ar2[0][0] 为 int 数值。

指针兼容性

考虑如下代码

```
int * p1;
const int * p2;
const int ** pp2;
p1  = p2;   // 非法：把 const 指针赋给非 const 指针
p2  = p1;   // 合法：把非 const 指针赋给 const 指针
pp2 = &p1;  // 非法：把非 const 指针赋给 const 指针
```

- 前面提过，把 const 指针赋给非 const 指针是错误的，因为可能会使用新指针来改变 const 数据。
- 但把非 const 指针赋给 const 指针是允许的，但前提是只进行一层间接运算。

指针兼容性

在进行两层间接运算时，这样的赋值不再安全。如果允许这样赋值，可能会产生如下问题：

```
int * p1;
const int ** pp2;
const int n = 13;
pp2 = &p1;   // 不允许，但我们假设允许
*pp2 = &n;   // 合法，两者都是 const，但这同时会使 p1 指向 n
*p1 = 10;    // 合法，但这将改变 const n 的值
```

函数与多维数组

本小节主要介绍如何来编写处理二维数组的函数。

- 需要很好地理解指针，以便正确地声明函数的参数；
- 在函数体内，使用数组符号来避免使用指针。

函数与多维数组

编程三个函数，分别求一个二维数组的行和、列和与总和，并写一个驱动程序测试它。

函数与多维数组 i

```
// 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 );
int sum2d(int (*ar)[COLS], int rows);

int main(void)
{
  int junk[ROWS][COLS] = {
    {2,4,6,8},
    {3,5,7,9},
    {12,10,8,6}
  };
```

函数与多维数组 ii

```
  sum_rows(junk, ROWS);
  sum_cols(junk, ROWS);
  printf("Sum of all elements = %d\n", sum2d(junk,
  ROWS));

  return 0;
}

void sum_rows(int ar[][COLS], int rows)
{
  int r, c, tot;

  for (r = 0; r < rows; r++)
  {
    tot = 0;
    for (c = 0; c < COLS; c++)
```

函数与多维数组 iii

```
      tot += ar[r][c];
    printf("row %d: sum = %d\n", r, tot);
  }
}

void sum_cols(int ar[][COLS], int rows)
{
  int r, c, 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);
  }
```

函数与多维数组 iv

```
}

int sum2d(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;
}
```

函数与多维数组

```
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
```

函数与多维数组

这些函数把数组名和行数作为参数传递给函数，都把 ar 看做是指向由 4 个 int 值构成的数组的指针。其中，列数在函数体内定义，而行数通过函数传递而得到。

这样的函数能处理列数为 4 的二维数组。

函数与多维数组

错误的声明方式

```
int sum2(int ar[][], int rows);
```

编译器会把数组符号转换为指针符号。这意味着，ar[1] 会被转换为 ar+1，此时编译器需要知道 ar 所指向对象的数据大小。

函数与多维数组

错误的声明方式

```
int sum2(int ar[][], int rows);
```

编译器会把数组符号转换为指针符号。这意味着，ar[1] 会被转换为 ar+1，此时编译器需要知道 ar 所指向对象的数据大小。

以下声明

```
int sum2(int ar[][4], int rows);
```

就表示 ar 指向由 4 个 int 值构成的数组，ar+1 表示“在这个地址上加 16 个字节大小”。

函数与多维数组

也可用如下声明方式

```
int sum2(int ar[3][4], int rows);
```

但第一个方括号中的内容（3）会被忽略。

函数与多维数组

一般地，声明 n 维数组的指针时，除了第一个方括号可以留空外，其它都需要填写数值。如

```
int sum4d(int ar[][4][5][6], int rows);
```

这是因为第一个方括号表示这是一个指针，其它方括号描述的是所指对象的数据类型。

函数与多维数组

一般地，声明 n 维数组的指针时，除了第一个方括号可以留空外，其它都需要填写数值。如

```
int sum4d(int ar[][4][5][6], int rows);
```

这是因为第一个方括号表示这是一个指针，其它方括号描述的是所指对象的数据类型。

其等效原型为

```
int sum4d(int (* ar)[4][5][6], int rows);
```

这里 `ar` 指向一个 $4 \times 5 \times 6$ 的 `int` 数组。

变长数组

变长数组

对于处理二维数组的函数，数组的行可以在函数调用时传递，但数组的列却只能预置在函数内部。例如函数定义为

```
#define COLS 4
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;
}
```

变长数组

假设定义了如下数组

```
int ar1[5][4];
int ar2[100][4];
int ar3[2][4];
```

可以使用以下函数调用

```
tot = sum2d(ar1, 5);
tot = sum2d(ar2, 100);
tot = sum2d(ar3, 2);
```

变长数组

如果要处理 6 行 5 列的数组，则需要新创建一个函数，其 COLS 定义为 5。这是因为数组的列数必须是常量：不能用一个变量来代替 COLS。

变长数组

创建一个处理任意二维数组的函数，也是有可能的，但比较繁琐（这样的函数需要把数组当做一维数组来传递，然后由函数计算每行的起始地址）。

变长数组

值得一提的是FORTRAN 语言允许在函数调用时指定二维的大小。虽然 FORTRAN 是很古老的编程语言，但多年来，数值计算专家们编制出了很多有用的 FORTRAN 计算库。C 正在逐渐代替 FORTRAN，因此如何能够简单地转换现有的 FORTRAN 库将是大有益处的。

变长数组

出于以上原因，C99 标准引入了变长数组 (VLA)，它允许使用变量定义数组各维数。

变长数组

出于以上原因，C99 标准引入了变长数组 (VLA)，它允许使用变量定义数组各维数。

你可以使用以下声明

```
int m = 4;
int n = 5;
double array[m][n];
```

但变长数组有一些限制，如变长数组必须是自动存储类的，这意味着它们必须在函数内部或作为函数参量声明，而且声明时不可以进行初始化。

变长数组

编写一个函数，用于计算任意二维 int 数组的和。

变长数组

1 以下代码示范如何声明一个带有二维变长数组参数的函数：

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

请注意在参量列表中，rows 和 cols 的声明需要早于 ar。

以下原型是错误的（顺序不对）：

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

可简写为

```
int sum2d(int, int, int ar[*][*]);
```

若省略名称，则需用星号代替方括号中省略的维数。

变长数组

2 函数定义为

```
int sum2d(int rows, int cols, int ar[rows][
cols])
{
  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;
}
```

变长数组 i

```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}
  };
```

变长数组 ii

```
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",
```

变长数组 iii

```
          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;
}
```

变长数组

```
3x5 array
Sum of all elements = 80
2x6 array
Sum of all elements = 315
3x10 VLA
Sum of all elements = 270
```

变长数组

注意，函数定义参量列表中的变长数组声明实际上并没有创建数组。和老语法一样，变长数组名实际上也是指针，也就是说带变长数组参量的函数实际上直接使用原数组，因此它有能力修改原数组。

变长数组

以下程序段指出了指针和实际数组是如何声明的。

```
{
   int thing[10][6];
   twoset(10, 6, thing);
   ...
}

void twoset(int n, int m, int ar[n][m])
//ar 是一个指针，它指向由 m 个 int 组成的数组
{
   int temp[n][m]; //temp 是一个 nxm 的 int 数组
   temp[0][0] = 2; //把 temp 的第一个元素设置为 2
   ar[0][0] = 2;   //把 thing[0][0] 设置为 2
}
```

关键概念

关键概念

- 数组是一种派生类型，因它建立在其他类型之上。
- 也就是说，你不是仅仅声明一个数组，而是声明了一个 `int` 数组、`float` 数组或其他类型的数组。
- 所谓的其他类型本身就可以是一种数组类型，此时便可得到数组的数组，即二维数组。

关键概念

- 编写处理数组的函数是有好处的，因为使用特定的函数执行特定的功能有助于程序的模块化。
- 使用数组名作为实参时，要知道并不是把整个数组传递给函数，而是传递它的地址；因此对应的形参是一个指针。
- 处理数组时，函数必须知道数组的地址和元素的个数。数组地址直接传递给函数，数组元素的个数可在函数内部设置，也可当做参数传递给函数。但后者更为通用，这样可以处理不同大小的数组。

关键概念

- 数组与指针之间联系紧密，指针符号和数组符号的运算往往可以互换使用。
- 正是这个原因，才允许处理数组的函数使用指针（而不是数组）作为形参，同时在函数中使用数组符号处理数组。