C进阶

一、指针

指针是什么？首先，它是一个值，这个值代表一个内存地址，因此指针相当于指向某个内存地址的路标。

字符 `*` 表示指针，通常跟在类型关键字的后面，表示指针指向的是什么类型的值。比如，`char*` 表示一个指向字符的指针，`float*` 表示一个指向 `float` 类型的值的指针。

```
int* intPtr;
```

1、指针变量

对指针变量的声明与对普通变量的声明基本一样，唯一的不同就是必须在指针变量名字前放置星号

```
int *p;
```

int 是指针指向的变量类型，上述声明说明 p 是指向 int **类型对象**的指针变量。用术语**对象** 来代替变量 ，是因为 p 可以指向**不属于变量的内存区域。**

C 语言要求每个指针变量只能指向一种特定类型（引用类型 ）的对象：

指针变量也可以指向另一个指针，即指向指针的指针（指向指针类型的指针）

2、*间接寻址运算符

`*` 这个符号除了表示指针以外，还可以作为运算符，用来取出指针变量所指向的内存地址里面的值。

为了获得对指针所指向对象的访问，可以使用* （间接寻址 ）运算符。如果 p 是指针，那么*p 表示 p 当前指向的对象。

3、&取地址运算符

`&` 运算符用来取出一个变量所在的内存地址。

为了找到变量的地址，可以使用& （取地址 ）运算符。如果 x 是变量，那么&x 就是 x 在内存中的地址。

`&` 运算符与 `*` 运算符互为逆运算，下面的表达式总是成立。

```
int i = 5;

if (i == *(&i)) // 正确
```

4、指针变量的初始化

声明指针变量之后，编译器会为指针变量本身分配一个内存空间，但是这个内存空间里面的值是随机的，也就是说，指针变量指向的值是随机的。这时一定不能去读写指针变量指向的地址，因为那个地址是随机地址，很可能会导致严重后果。

```
int* p;
*p = 1; // 错误
```

上面的代码是错的，因为 `p` 指向的那个地址是随机的，向这个随机地址里面写入 `1`，会导致意想不到的结果。

正确做法是指针变量声明后，必须先让它指向一个分配好的地址，然后再进行读写，这叫做指针变量的初始化。

```
int* p;
int i;

p = &i;
*p = 13;
```

上面示例中，`p` 是指针变量，声明这个变量后，`p` 会指向一个随机的内存地址。这时要将它指向一个已经分配好的内存地址，上例就是再声明一个整数变量 `i`，编译器会为 `i` 分配内存地址，然后让 `p` 指向 `i` 的内存地址（`p = &i;`）。完成初始化之后，就可以对 `p` 指向的内存地址进行赋值了（`*p = 13;`）。

为了防止读写未初始化的指针变量，可以养成习惯，将未初始化的指针变量设为 `NULL`。

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

`NULL` 在 C 语言中是一个常量，表示地址为 `0` 的内存空间，这个地址是无法使用的，读写该地址会报错。

5、指针的运算

指针本质上就是一个无符号整数，代表了内存地址。它可以进行运算，但是规则并不是整数运算的规则。

C 语言支持 3 种指针运算

- 指针加法：指针加上整数
- 指针减法：指针减去整数
- 指针相减：两个指针相减

（1）指针与整数值的加减运算

指针与整数值的运算，表示指针的移动。

```
short* j;
j = (short*)0x1234;
j = j + 1; // 0x1236
```

上面示例中，`j` 是一个指针，指向内存地址 `0x1234`。你可能以为 `j + 1` 等于 `0x1235`，但正确答案是 `0x1236`。原因是 `j + 1` 表示指针向内存地址的高位移动一个单位，而一个单位的 `short` 类型占据两个字节的宽度，所以相当于向高位移动两个字节。同样的，`j - 1` 得到的结果是 `0x1232`。

指针移动的单位，与指针指向的数据类型有关。数据类型占据多少个字节，每单位就移动多少个字节。

（2）指针与指针的减法

相同类型的指针允许进行减法运算，返回它们之间的距离，即相隔多少个数据单位。

高位地址减去低位地址，返回的是正值；低位地址减去高位地址，返回的是负值。

这时，减法返回的值属于 `ptrdiff_t` 类型，这是一个带符号的整数类型别名，具体类型根据系统不同而不同。这个类型的原型定义在头文件 `stddef.h` 里面。

```
short* j1;
short* j2;

j1 = (short*)0x1234;
j2 = (short*)0x1236;

ptrdiff_t dist = j2 - j1;
printf("%td\n", dist); // 1
```

上面示例中，`j1` 和 `j2` 是两个指向 short 类型的指针，变量 `dist` 是它们之间的距离，类型为 `ptrdiff_t`，值为 `1`，因为相差 2 个字节正好存放一个 short 类型的值。

（3）指针与指针的比较运算

指针之间的比较运算，比较的是各自的内存地址哪一个更大，返回值是整数 `1`（true）或 `0`（false）。

二、函数

函数是一段可以重复执行的代码。它可以接受不同的参数，完成对应的操作。下面的例子就是一个函数。

```
int plus_one(int n) {
  return n + 1;
}
```

上面的代码声明了一个函数 `plus_one()`。

函数声明的语法有以下几点，需要注意。

（1）返回值类型。函数声明时，首先需要给出返回值的类型，上例是 `int`，表示函数 `plus_one()` 返回一个整数。

（2）参数。函数名后面的圆括号里面，需要声明参数的类型和参数名，`plus_one(int n)` 表示这个函数有一个整数参数 `n`。

（3）函数体。函数体要写在大括号里面，后面（即大括号外面）不需要加分号。大括号的起始位置，可以跟函数名在同一行，也可以另起一行，本书采用同一行的写法。

（4）`return` 语句。`return` 语句给出函数的返回值，程序运行到这一行，就会跳出函数体，结束函数的调用。如果函数没有返回值，可以省略 `return` 语句，或者写成 `return;`。非 void 类型必须 return 返回值。

调用函数时，只要在函数名后面加上圆括号就可以了，实际的参数放在圆括号里面。

一、函数定义和声明

函数定义格式：

```
返回类型  函数名（形式参数）{
   复合语句;
```

```
}
```

函数声明：

通常情况下函数的定义总是在调用点之上，但是这样会给程序的结构带来问题，如果函数的定义在调用点之后，那么主函数无法调用函数就会出错。为了避免调用问题，可以在调用之前就为函数声明，使编译器率先知道函数定义的存在。

```
返回类型 函数名(形式参数);
```

二、参数的传值引用

如果函数的参数是一个变量，那么调用时，传入的是这个变量的值的拷贝，而不是变量本身。因为传入函数的是变量的拷贝，而不是变量本身，拷贝的变化，影响不到原始变量。这就叫做“传值引用”。

所以，如果参数变量发生变化，最好把它作为返回值传出来。

三、指针与函数

函数本身就是一段内存里面的代码，**C 语言允许通过指针获取函数。**

```
void print(int a) {
  printf("%d\n", a);
}
void (*print_ptr)(int) = &print;
```

上面示例中，变量 `print_ptr` 是一个函数指针，它指向函数 `print()` 的地址。函数 `print()` 的地址可以用 `&print` 获得。注意，`(*print_ptr)` 一定要写在圆括号里面，否则函数参数 `(int)` 的优先级高于 `*`，整个式子就会变成 `void* print_ptr(int)`。

有了函数指针，通过它也可以调用函数。

```
(*print_ptr)(10);
// 等同于
print(10);
```

比较特殊的是，C 语言还规定，**函数名本身就是指向函数代码的指针，通过函数名就能获取函数地址。也就是说，`print` 和 `&print` 是一回事。**

```
if (print == &print) // true
```

因此，上面代码的 `print_ptr` 等同于 `print`。

```
void (*print_ptr)(int) = &print;
// 或
void (*print_ptr)(int) = print;
if (print_ptr == print) // true
```

所以，对于任意函数，都有五种调用函数的写法。

```
// 写法一
print(10)
// 写法二
(*print)(10)
```

```
// 写法三
(&print)(10)
// 写法四
(*print_ptr)(10)
// 写法五
print_ptr(10)
```

为了简洁易读，一般情况下，函数名前面都不加 `*` 和 `&` 。

这种特性的一个应用是，**如果一个函数的参数或返回值，也是一个函数，那么函数原型可以写成下面这样。**

```
int compute(int (*myfunc)(int), int, int);
```

上面示例可以清晰地表明，函数 `compute()` 的第一个参数也是一个函数。

不仅可以通过指针来调用函数，指针真正对于函数有重要作用的是**指针作为函数的参数。指针作为参数的两个作用**

- **指针作为参数传值会节省大量的内存空间**
- **指针作为参数可以改变指向变量的值（实际参数的值）**

指针作为函数的参数在 C 语言中非常常见//scanf(”%d”,&i); scanf()函数就接收一个指针作为参数。

指针作为参数更重要的不是运用指针变量存储的内存地址，而主要是通过指针变量的指向对象关系，来获取指向对象的值，因为实际函数的调用中，内存地址参与运算没有任何意义。

指针高效的原因是在实际参数较大时，调用函数会需要大量的内存空间，变量的传递会浪费大量的时间和空间，而指针则会节省参数传递的时间和需要占用的内存空间。

指针作为函数参数的主要作用是改变实际参数的值

指针参数：max(array)&&min(array)

```
#include<stdio.h>
#define N 10
void max_min(int a[],int n,int *max,int *min);
int main(){
    int arr[N],i,big,small;
    printf("请输入%d 个数字：",N);
    for(i=0;i<N;i++){
        scanf("%d",&arr[i]);
    }
    max_min(arr,N,&big,&small);
    printf("最大的数：%d\n",big);
    printf("最小的数：%d\n",small);
    return 0;
}
void max_min(int a[],int n,int *max,int *min){
    int i;
    *max=*min=a[0];
    //用数组的第一个元素初始化两个变量，以确定比较参照
    for(i=0;i<n;i++){
        if(a[i]>*max){
            *max=a[i];
```

```
        }else if(a[i]<*min){
            *min=a[i];
        }
    }
}
```

四、exit()函数

`exit()` 函数用来终止整个程序的运行。一旦执行到该函数，程序就会立即结束。该函数的原型定义在头文件 `stdlib.h` 里面。

`exit()` 可以向程序外部返回一个值，它的参数就是程序的返回值。一般来说，使用两个常量作为它的参数：`EXIT_SUCCESS`（相当于 0）表示程序运行成功，`EXIT_FAILURE`（相当于 1）表示程序异常中止。这两个常数也是定义在 `stdlib.h` 里面。

```
// 程序运行成功
// 等同于 exit(0);
exit(EXIT_SUCCESS);

// 程序异常中止
// 等同于 exit(1);
exit(EXIT_FAILURE);
```

在 `main()` 函数里面，`exit()` 等价于使用 `return` 语句。其他函数使用 `exit()`，就是终止整个程序的运行，没有其他作用。

C 语言还提供了一个 `atexit()` 函数，用来登记 `exit()` 执行时额外执行的函数，用来做一些退出程序时的收尾工作。该函数的原型也是定义在头文件 `stdlib.h`。

```
int atexit(void (*func)(void));
```

`atexit()` 的参数是一个函数指针。注意，它的参数函数（下例的 `print`）不能接受参数，也不能有返回值。

```
void print(void) {
  printf("something wrong!\n");
}
atexit(print);
exit(EXIT_FAILURE);
```

上面示例中，`exit()` 执行时会先自动调用 `atexit()` 注册的 `print()` 函数，然后再终止程序。

五、函数说明符

C 语言提供了一些函数说明符，让函数用法更加明确。

1、extern 说明符

对于多文件的项目，源码文件会用到其他文件声明的函数。这时，当前文件里面，需要给出外部函数的原型，并用 `extern` 说明该函数的定义来自其他文件。

```
extern int foo(int arg1, char arg2);

```

```
int main(void) {
  int a = foo(2, 3);
  // ...
  return 0;
}
```

上面示例中，函数 `foo()` 定义在其他文件， `extern` 告诉编译器当前文件不包含该函数的定义。

由于函数原型默认就是 `extern` ，这里不加 `extern` ，效果是一样的。

2、static 说明符

默认情况下，每次调用函数时，函数的内部变量都会重新初始化，不会保留上一次运行的值。

`static` 说明符可以改变这种行为。

`static` 用于函数内部声明变量时，表示该变量只需要初始化一次，不需要在每次调用时都进行初始化。也就是说，它的值在两次调用之间保持不变。

```
#include <stdio.h>

void counter(void) {
  static int count = 1;  // 只初始化一次
  printf("%d\n", count);
  count++;
}
int main(void) {
  counter();  // 1
  counter();  // 2
  counter();  // 3
  counter();  // 4
}
```

上面示例中，函数 `counter()` 的内部变量 `count` ，使用 `static` 说明符修饰，表明这个变量只初始化一次，以后每次调用时都会使用上一次的值，造成递增的效果。

注意，`static` 修饰的变量初始化时，只能赋值为常量，不能赋值为变量。

```
int i = 3;
static int j = i; // 错误
```

上面示例中， `j` 属于静态变量，初始化时不能赋值为另一个变量 `i` 。

另外，在块作用域中，`static` 声明的变量有默认值 `0` 。

```
static int foo;
// 等同于
static int foo = 0;
```

`static` **可以用来修饰函数本身。**

```
static int Twice(int num) {
  int result = num * 2;
  return(result);
}
```

上面示例中，`static` 关键字表示该函数只能在当前文件里使用，如果没有这个关键字，其他文件也可以使用这个函数（通过声明函数原型）。

`static` 也可以用在参数里面，修饰参数数组。

```
int sum_array(int a[static 3], int n) {
  // ...
}
```

上面示例中，`static` 对程序行为不会有任何影响，只是用来告诉编译器，该数组长度至少为3，某些情况下可以加快程序运行速度。另外，需要注意的是，对于多维数组的参数，`static` 仅可用于第一维的说明。

3、const 说明符

函数参数里面的 `const` 说明符，表示函数内部不得修改该参数变量。

```
void f(int* p) {
  // ...
}
```

上面示例中，函数 `f()` 的参数是一个指针 `p`，函数内部可能会改掉它所指向的值 `*p`，从而影响到函数外部。

为了避免这种情况，可以在声明函数时，在指针参数前面加上 `const` 说明符，告诉编译器，函数内部不能修改该参数所指向的值。

```
void f(const int* p) {
  *p = 0; // 该行报错
}
```

上面示例中，声明函数时，`const` 指定不能修改指针 `p` 指向的值，所以 `*p = 0` 就会报错。

但是上面这种写法，只限制修改 `p` 所指向的值，而 `p` 本身的地址是可以修改的。

```
void f(const int* p) {
  int x = 13;
  p = &x; // 允许修改
}
```

上面示例中，`p` 本身是可以修改，`const` 只限定 `*p` 不能修改。

如果想限制修改 `p`，可以把 `const` 放在 `p` 前面。

```
void f(int* const p) {
  int x = 13;
  p = &x; // 该行报错
}
```

如果想同时限制修改 `p` 和 `*p`，需要使用两个 `const`。

```
void f(const int* const p) {
  // ...
}
```

六、可变参数

有些函数的参数数量是不确定的，声明函数的时候，可以使用省略号 `...` 表示可变数量的参数。

```
int printf(const char* format, ...);
```

上面示例是 `printf()` 函数的原型，除了第一个参数，其他参数的数量是可变的，与格式字符串里面的占位符数量有关。这时，就可以用 `...` 表示可变数量的参数。

注意， `...` 符号必须放在参数序列的结尾，否则会报错。

头文件 `stdarg.h` 定义了一些宏，可以操作可变参数。

（1） `va_list` ：一个数据类型，用来定义一个可变参数对象。它必须在操作可变参数时，首先使用。

（2） `va_start` ：一个函数，用来初始化可变参数对象。它接受两个参数，第一个参数是可变参数对象，第二个参数是原始函数里面，可变参数之前的那个参数，用来为可变参数定位。

（3） `va_arg` ：一个函数，用来取出当前那个可变参数，每次调用后，内部指针就会指向下一个可变参数。它接受两个参数，第一个是可变参数对象，第二个是当前可变参数的类型。

（4） `va_end` ：一个函数，用来清理可变参数对象。

下面是一个例子。

```
double average(int i, ...) {
  double total = 0;
  va_list ap;
  va_start(ap, i);
  for (int j = 1; j <= i; ++j) {
    total += va_arg(ap, double);
  }
  va_end(ap);
  return total / i;
}
```

上面示例中， `va_list ap` 定义 `ap` 为可变参数对象， `va_start(ap, i)` 将参数 `i` 后面的参数统一放入 `ap` ， `va_arg(ap, double)` 用来从 `ap` 依次取出一个参数，并且指定该参数为 double 类型， `va_end(ap)` 用来清理可变参数对象。

七、作用域

任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。

C 语言中有三个地方可以声明变量：

1. 在函数或块内部的**局部**变量
2. 在所有函数外部的**全局**变量
3. 在**形式**参数的函数参数定义中

全局变量与局部变量在内存中的区别：

- 全局变量保存在内存的全局存储区中，占用静态的存储单元；
- 局部变量保存在栈中，只有在所在函数被调用时才动态地为变量分配存储单元。

初始化局部变量和全局变量

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动对其初始化，如下所示：

数据类型	初始化默认值
int	0
char	’\0’
float	0
double	0
pointer	NULL

正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果，因为未初始化的变量会导致一些在内存位置中已经可用的垃圾值。

局部变量：

在某个函数或块的内部声明的变量称为局部变量。它们只能被该函数或该代码块内部的语句使用。局部变量在函数外部是不可知的。

全局变量：

全局变量是定义在函数外部，通常是在程序的顶部。全局变量在整个程序生命周期内都是有效的，在任意的函数内部能访问全局变量。

全局变量可以被任何函数访问。也就是说，全局变量在声明后整个程序中都是可用的。

局部变量和全局变量的名称可以相同，但是在函数内，如果两个名字相同，会使用局部变量值，全局变量不会被使用。

形式参数：

函数的参数，形式参数，被当作该函数内的局部变量，如果与全局变量同名它们会优先使用。

三、数组

一、一维数组

数组是含有多个数据值的数据结构，并且每个数据值具有相同的数据类型。这些数据值称为元素，可以根据元素在数组中的位置把它们选出来。

```
int arr[10];//声明一个长度为 10 元素类型为 int 的数组
```

1、数组的下标

为了存取特定的数组元素，可以通过数组的下标获取数组元素，也可以在循环中对数组遍历

```
int arr[10];
arr[0]//数组的第一个元素
arr[9]//数组的最后一个元素
```

2、数组的初始化

与标准变量一样数组也可以进行初始化的操作，

- 数组初始化式：是用一个大括号括起来的常量数值列表来进行初始化

```
int arr[10]={1,2,3,4,5,6,7,8,9,10};
```

- 初始化列表比数组长度短是允许的，剩余的元素会赋值为 0，但初始化列表比数组长是错误的,初始化列表完全为空也是错误的，可以用 0 替代。

```
int arr[10]={1,2,3,4,5};
//arr=={1,2,3,4,5,0,0,0,0,0}
int arr[10]={1,2,3,4,5,6,7,8,9,10,11};//err
int arr[10]={}//err
int arr[10]={0}
```

如果数组元素中有一部分未确定值，可以指定初始化,未指定的元素将默认为 0

```
int arr[10]={[0]=1,[5]=6,[9]=10}
```

3、数组的长度（sizeof）

`sizeof` 运算符会返回整个数组的字节长度。

```
int a[] = {22, 37, 3490};
int arrLen = sizeof(a); // 12
```

上面示例中， `sizeof` 返回数组 `a` 的字节长度是 `12` 。

由于数组成员都是同一个类型，每个成员的字节长度都是一样的，所以数组整体的字节长度除以某个数组成员的字节长度，就可以得到数组的成员数量。

```
sizeof(a) / sizeof(a[0])
```

上面示例中， `sizeof(a)` 是整个数组的字节长度， `sizeof(a[0])` 是数组成员的字节长度，相除就是数组的成员数量。

注意， `sizeof` 返回值的数据类型是 `size_t` ，所以 `sizeof(a) / sizeof(a[0])` 的数据类型也是 `size_t` 。在 `printf()` 里面的占位符，要用 `%zd` 或 `%zu` 。

```
int x[12];

printf("%zu\n", sizeof(x));     // 48
printf("%zu\n", sizeof(int));  // 4
printf("%zu\n", sizeof(x) / sizeof(int)); // 12
```

上面示例中， `sizeof(x) / sizeof(int)` 就可以得到数组成员数量 `12` 。

二、多维数组

C 语言允许声明多个维度的数组，有多少个维度，就用多少个方括号，比如二维数组就使用两个方括号。

```
int board[10][10];
```

上面示例声明了一个二维数组，第一个维度有 10 个成员，第二个维度也有 10 个成员。

多维数组可以理解成，上层维度的每个成员本身就是一个数组。比如上例中，第一个维度的每个成员本身就是一个有 10 个成员的数组，因此整个二维数组共有 100 个成员（10 x 10 = 100）。

三维数组就使用三个方括号声明，以此类推。

```
int c[4][5][6];
```

引用二维数组的每个成员时，需要使用两个方括号，同时指定两个维度。

```
board[0][0] = 13;
board[9][9] = 13;
```

注意，`board[0][0]` 不能写成 `board[0, 0]`，因为 `0, 0` 是一个逗号表达式，返回第二个值，所以 `board[0, 0]` 等同于 `board[0]`。

跟一维数组一样，多维数组每个维度的第一个成员也是从 `0` 开始编号。

多维数组也可以使用大括号，一次性对所有成员赋值。

```
int a[2][5] = {
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9}
};
```

上面示例中，`a` 是一个二维数组，这种赋值写法相当于将第一维的每个成员写成一个数组。

这种写法不用为每个成员都赋值，缺少的成员会自动设置为 `0`。

多维数组也可以指定位置，进行初始化赋值。

```
int a[2][2] = {[0][0] = 1, [1][1] = 2};
```

上面示例中，指定了 `[0][0]` 和 `[1][1]` 位置的值，其他位置就自动设为 `0`。

不管数组有多少维度，在内存里面都是线性存储，`a[0][0]` 的后面是 `a[0][1]`，`a[0][1]` 的后面是 `a[1][0]`，以此类推。因此，多维数组也可以使用单层大括号赋值，下面的语句与上面的赋值语句是完全等同的。

```
int a[2][2] = {1, 0, 0, 2};
```

三、变长数组

数组声明的时候，数组长度除了使用常量，也可以使用变量。这叫做变长数组（variable-length array，简称 VLA）。

```
int n = x + y;
int arr[n];
```

上面示例中，数组 `arr` 就是变长数组，因为它的长度取决于变量 `n` 的值，编译器没法事先确定，只有运行时才能知道 `n` 是多少。

变长数组的根本特征，就是数组长度只有运行时才能确定。它的好处是程序员不必在开发时，随意为数组指定一个估计的长度，程序可以在运行时为数组分配精确的长度。

变长数组也可以用于多维数组。

```
int m = 4;
int n = 5;
int c[m][n];
```

四、常量数组

在数组前面声明一个 const 类型限定，就可以将数组变为常量数组，意味着程序不应该对数组进行修改操作。

五、指针和数组

C 语言的数组和指针的关系十分紧密，当指针指向数组元素时可以对指针进行算数运算（加减运算）通过这种运算我们可以用指针替代数组下标对数组进行处理

数组指针的运算：

- 指针加法：指针加上整数
- 指针减法：指针减去整数
- 指针相减：两个指针相减

指针加法：

指针加上整数产生指向特定数组元素的指针，指针 p 加上整数 j 就是原先指向元素后的第 j 个元素。指针 p 指向数组元素 a[i]，那么 p+j 就指向 a[i+j]（前提是 i+j 不能超过数组的长度）。

```
int a[10],*p,*q,j=3;
p=&a[2];
q=p+j;
//*q==a[5]
```

指针减法：

指针的减法和指针的加法原理相同，不同的是加法是指针指向向下标大的一方向移动，减法就是指针指向下下标小的一方向移动

```
int a[10],*p,*q,j=3;
p=&a[8];
q=p-j;
//*q==a[5]
```

指针相减：

两个指针相减时，结果为指针之间的距离（用数组元素的个数来度量），p 指向 a[i]，q 指向 a[j]，那么 p–q=i–j。

```
p = &a[5];
q = &a[1];
i = p - q; /* i is 4 */
i = q - p; /* i is -4 */
```

指针比较：

可以用关系运算符（< 、<= 、> 和>= ）和判等运算符（== 和!= ）进行指针比较。只有在两个指针指向同一数组时，用关系运算符进 行的指针比较才有意义。比较的结果依赖于数组中两个元素的相对位 置。

```
p = &a[5];
q = &a[1];
//赋值后 p <= q 的值是 0，而 p >= q 的值是 1。
```

指向复合常量的指针：

指针指向由复合字面量创建的数组中的某个元素是合法 的。

```
int *p3=(int[]){1,2,3,4,5};
```

```
//*p==a[0] 指向数组的第一个元素
```

指针处理数组：

指针的算术运算允许通过对指针变量进行重复自增来访问数组的元素。

```
#define N 10 ...
int a[N], sum, *p; ...
sum = 0;
for (p = &a[0]; p < &a[N]; p++) sum += *p;
```

> for 语句中的条件 p < &a[N] 值得特别说明一下。尽管元素 a[N] 不存在（数组 a 的下标从 0 到 N–1），但是对它使用取地址运算符是 合法的。因为循环不会尝试检查 a[N] 的值，所以在上述方式下使用 a[N] 是非常安全的。执行循环体时 p 依次等于&a[0],&a[1], ...,&a[N–1]，但是当p 等于&a[N] 时，循环终止。

C 程序经常在处理数组元素的语句中组合* （间接寻址）运算符和 ++ 运算符。++自加运算符有前缀和后缀之分，不同位置与指针的结合有不同的操作

一个问题是：把值存入一个数组元素中，然后前进到下一个数组

数组的下标写法是：

```
a[i++]=j;
```

p 指针指向的写法是：

```
*p++=j;
```

表达式	含义
p++ 或(p++)	自增前表达式的值是*p ，以后再自增p
(*p)++	自增前表达式的值是*p ，以后再自增*p
++p 或(++p)	先自增p ，自增后表达式的值是*p
++*p 或++(*p)	先自增*p ，自增后表达式的值是*p

数组名作为指针：

指针的算数运算是指针和数组之间一种相互关联的方法，但并不是唯一的关系

另一种指针和数组的关系是：

可以用数组的名字作为指向数组第一个元素的指针

这种关系能够简化指针的算术运算，也能够使指针和数组更加通用

```
int a[10];
*a=0;
```

```
//*a==&a[0]
```

用 a 作为指向数组第一个元素的指针，可以修改 a[0]：

通常情况下，a + i 等同于&a[i] （两者都表示指向数组 a 中元素 i 的指针），并且*(a+i) 等价于 a[i] （两者都表示元素 i 本 身）。换句话说，可以把数组的取下标操作看成是指针算术运算的一 种形式。

虽然可以把数组名用作指针，但是不能给数组名赋新的 值。试图使数组名指向

数组型的实际参数：

数组名作为参数在传递给函数时，总是被视为指针

```
int largest(int arr[],int n){
    int i,max;
    max=arr[0];
    for(i=0;i<n;i++){
        if(max<arr[i]){
            max=arr[i];
        }
    }
    return max;
}
```

在这个函数的调用中，数组作为参数并没有全部复制，而数组名作为指针指向参数数组的第一个元素，这

数组型形式参数看作为指针会产生重要的影响

在给函数传递普通变量时，变量的值会被复制，任何对相应的形 式参数的改变都不会影响到变量。反之，因为没有对数组本身进 行复制，所以作为实际参数的数组是可能被改变的。

```
void arr_one(int a[],int n){
    int i;
    for(i=0;i<n;i++){
      a[i]=1;
    }
}
//这里作为实际参数的数组被改变
```

六、数组的复制

由于数组名是指针，所以复制数组不能简单地复制数组名。

```
int* a;
int b[3] = {1, 2, 3};

a = b;
```

上面的写法，结果不是将数组 `b` 复制给数组 `a`，而是让 `a` 和 `b` 指向同一个数组。

复制数组最简单的方法，还是使用循环，将数组元素逐个进行复制。

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```

上面示例中，通过将数组 `b` 的成员逐个复制给数组 `a`，从而实现数组的赋值。

另一种方法是使用 `memcpy()` 函数（定义在头文件 `string.h`），直接把数组所在的那一段内存，再复制一份。

```
memcpy(a, b, sizeof(b));
```

上面示例中，将数组 `b` 所在的那段内存，复制给数组 `a`。这种方法要比循环复制数组成员要快。

七、数组作为参数

声明参数数组

数组作为函数的参数，一般会同时传入数组名和数组长度。

```
int sum_array(int a[], int n) {
  // ...
}

int a[] = {3, 5, 7, 3};
int sum = sum_array(a, 4);
```

上面示例中，函数 `sum_array()` 的第一个参数是数组本身，也就是数组名，第二个参数是数组长度。

由于数组名就是一个指针，如果只传数组名，那么函数只知道数组开始的地址，不知道结束的地址，所以才需要把数组长度也一起传入。

如果函数的参数是多维数组，那么除了第一维的长度可以当作参数传入函数，其他维的长度需要写入函数的定义。

```
int sum_array(int a[][4], int n) {
  // ...
}

int a[2][4] = {
  {1, 2, 3, 4},
  {8, 9, 10, 11}
};
int sum = sum_array(a, 2);
```

上面示例中，函数 `sum_array()` 的参数是一个二维数组。第一个参数是数组本身（`a[][4]`），这时可以不写第一维的长度，因为它作为第二个参数，会传入函数，但是一定要写第二维的长度 `4`。

这是因为函数内部拿到的，只是数组的起始地址 `a`，以及第一维的成员数量 `2`。如果要正确计算数组的结束地址，还必须知道第一维每个成员的字节长度。写成 `int a[][4]`，编译器就知道了，第一维每个成员本身也是一个数组，里面包含了 4 个整数，所以每个成员的字节长度就是 `4 * sizeof(int)`。

变长数组作为参数

变长数组作为函数参数时，写法略有不同。

```
int sum_array(int n, int a[n]) {
  // ...
}

int a[] = {3, 5, 7, 3};
int sum = sum_array(4, a);
```

上面示例中，数组 `a[n]` 是一个变长数组，它的长度取决于变量 `n` 的值，只有运行时才能知道。所以，变量 `n` 作为参数时，顺序一定要在变长数组前面，这样运行时才能确定数组 `a[n]` 的长度，否则就会报错。

因为函数原型可以省略参数名，所以变长数组的原型中，可以使用 `*` 代替变量名，也可以省略变量名。

```
int sum_array(int, int [*]);
int sum_array(int, int []);
```

上面两种变长函数的原型写法，都是合法的。

变长数组作为函数参数有一个好处，就是多维数组的参数声明，可以把后面的维度省掉了。

```
// 原来的写法
int sum_array(int a[][4], int n);

// 变长数组的写法
int sum_array(int n, int m, int a[n][m]);
```

上面示例中，函数 `sum_array()` 的参数是一个多维数组，按照原来的写法，一定要声明第二维的长度。但是使用变长数组的写法，就不用声明第二维长度了，因为它可以作为参数传入函数。

数组字面量作为参数

C 语言允许将数组字面量作为参数，传入函数。

```
// 数组变量作为参数
int a[] = {2, 3, 4, 5};
int sum = sum_array(a, 4);

// 数组字面量作为参数
int sum = sum_array((int []){2, 3, 4, 5}, 4);
```

上面示例中，两种写法是等价的。第二种写法省掉了数组变量的声明，直接将数组字面量传入函数。`{2, 3, 4, 5}` 是数组值的字面量，`(int [])` 类似于强制的类型转换，告诉编译器怎么理解这组值。

四、字符串

1、字符串

C 语言没有单独的字符串类型，字符串被当作字符数组，即 `char` 类型的数组。比如，字符串“Hello”是当作数组 `{'H', 'e', 'l', 'l', 'o'}` 处理的。

编译器会给数组分配一段连续内存，所有字符储存在相邻的内存单元之中。在字符串结尾，C 语言会自动添加一个全是二进制 `0` 的字节，写作 `\0` 字符，表示字符串结束。字符 `\0` 不同于字符 `0`，前者的 ASCII 码是 0（二进制形式 `00000000`），后者的 ASCII 码是 48（二进制形式 `00110000`）。所以，字符串“Hello”实际储存的数组是 `{'H', 'e', 'l', 'l', 'o', '\0'}`。

所有字符串的最后一个字符，都是 `\0`。这样做的好处是，C 语言不需要知道字符串的长度，就可以读取内存里面的字符串，只要发现有一个字符是 `\0`，那么就知道字符串结束了。

字符串写成数组的形式，是非常麻烦的。C 语言提供了一种简写法，双引号之中的字符，会被自动视为字符数组。

```
{'H', 'e', 'l', 'l', 'o', '\0'}

// 等价于
"Hello"
```

上面两种字符串的写法是等价的，内部存储方式都是一样的。双引号里面的字符串，不用自己添加结尾字符 `\0`，C 语言会自动添加。

注意，双引号里面是字符串，单引号里面是字符，两者不能互换。如果把 `Hello` 放在单引号里面，编译器会报错。

```
// 报错
'Hello'
```

另一方面，即使双引号里面只有一个字符（比如 `"a"`），也依然被处理成字符串（存储为 2 个字节），而不是字符 `'a'`（存储为 1 个字节）。

如果字符串内部包含双引号，则该双引号需要使用反斜杠转义。

```
"She replied, \"It does.\""
```

反斜杠还可以表示其他特殊字符，比如换行符（`\n`）、制表符（`\t`）等。

```
"Hello, world!\n"
```

如果字符串过长，可以在需要折行的地方，使用反斜杠（`\`）结尾，将一行拆成多行。

```
"hello \
world"
```

上面这种写法有一个缺点，就是第二行必须顶格书写，如果想包含缩进，那么缩进也会被计入字符串。为了解决这个问题，**C 语言允许合并多个字符串字面量，只要这些字符串之间没有间隔，或者只有空格，C 语言会将它们自动合并。**

```
char greeting[50] = "Hello, ""how are you ""today!";
// 等同于
char greeting[50] = "Hello, how are you today!";
```

这种新写法支持多行字符串的合并。

```
char greeting[50] = "Hello, "
  "how are you "
  "today!";
```

`printf()` 使用占位符 `%s` 输出字符串。

```
printf("%s\n", "hello world")
```

2、字符串变量

字符串变量可以声明成一个字符数组，也可以声明成一个指针，指向字符数组。

```
// 写法一
char s[14] = "Hello, world!";

// 写法二
char* s = "Hello, world!";
```

上面两种写法都声明了一个字符串变量 `s` 。如果采用第一种写法，由于字符数组的长度可以让编译器自动计算，所以声明时可以省略字符数组的长度。

```
char s[] = "Hello, world!";
```

上面示例中，编译器会将数组 `s` 的长度指定为 14，正好容纳后面的字符串。

字符数组的长度，可以大于字符串的实际长度。

```
char s[50] = "hello";
```

上面示例中，字符数组 `s` 的长度是 `50` ，但是字符串“hello”的实际长度只有 6（包含结尾符号 `\0` ），所以后面空出来的 44 个位置，都会被初始化为 `\0` 。

字符数组的长度，不能小于字符串的实际长度。

```
char s[5] = "hello";//err
```

上面示例中，字符串数组 `s` 的长度是 `5` ，小于字符串“hello”的实际长度 6，这时编译器会报错。因为如果只将前 5 个字符写入，而省略最后的结尾符号 `\0` ，这很可能导致后面的字符串相关代码出错。

字符指针和字符数组，这两种声明字符串变量的写法基本是等价的，但是有两个差异。

第一个差异是，指针指向的字符串，在 C 语言内部被当作常量，不能修改字符串本身。

```
char* s = "Hello, world!";
s[0] = 'z'; // 错误
```

如果使用数组声明字符串变量，就没有这个问题，可以修改数组的任意成员。

```
char s[] = "Hello, world!";
s[0] = 'z';
```

为什么字符串声明为指针时不能修改，声明为数组时就可以修改？原因是系统会将字符串的字面量保存在内存的常量区，这个区是不允许用户修改的。声明为指针时，指针变量存储的值是一个指向常量区的内存地址，因此用户不能通过这个地址去修改常量区。*但是，声明为数组时，编译器会给数组单独分配一段内存，字符串字面量会被编译器解释成字符数组，逐个字符写入这段新分配的内存之中，而这段新内存是允许修改的。*

为了提醒用户，字符串声明为指针后不得修改，可以在声明时使用 `const` 说明符，保证该字符串是只读的。

```
const char* s = "Hello, world!";
```

上面字符串声明为指针时，使用了 `const` 说明符，就保证了该字符串无法修改。一旦修改，编译器肯定会报错。

第二个差异是，指针变量可以指向其它字符串。

```
char* s = "hello";
s = "world";
```

字符指针可以指向另一个字符串。

但是，字符数组变量不能指向另一个字符串。

```
char s[] = "hello";
s = "world"; // 报错
```

上面示例中，字符数组的数组名，总是指向初始化时的字符串地址，不能修改。

同样的原因，声明字符数组后，不能直接用字符串赋值。

```
char s[10];
s = "abc"; // 错误
```

上面示例中，不能直接把字符串赋值给字符数组变量，会报错。原因是字符数组的变量名，跟所指向的数组是绑定的，不能指向另一个地址。

为什么数组变量不能赋值为另一个数组？原因是数组变量所在的地址无法改变，或者说，编译器一旦为数组变量分配地址后，这个地址就绑定这个数组变量了，这种绑定关系是不变的。C 语言也因此规定，数组变量是一个不可修改的左值，即不能用赋值运算符为它重新赋值。

想要重新赋值，必须使用 C 语言原生提供的 `strcpy()` 函数，通过字符串拷贝完成赋值。这样做以后，数组变量的地址还是不变的，即 `strcpy()` 只是在原地址写入新的字符串，而不是让数组变量指向新的地址。

```
char s[10];
strcpy(s, "abc");
```

3、字符库函数

1）strlen()函数

`strlen()` 函数返回字符串的字节长度，不包括末尾的空字符 `\0` 。

它的参数是字符串变量，返回的是 `size_t` 类型的无符号整数，除非是极长的字符串，一般情况下当作 `int` 类型处理即可。

`strlen()` 的原型在标准库的 `string.h` 文件中定义，使用时需要加载头文件 `string.h` 。

注意，字符串长度（ `strlen()` ）与字符串变量长度（ `sizeof()` ），是两个不同的概念。

```
char s[50] = "hello";
printf("%d\n", strlen(s));  // 5
printf("%d\n", sizeof(s));  // 50
```

2）strcpy()函数

字符串的复制，不能使用赋值运算符，直接将一个字符串赋值给字符数组变量。

```
char str1[10];
char str2[10];
str1 = "abc"; // 报错
str2 = str1;  // 报错
```

上面两种字符串的复制写法，都是错的。因为数组的变量名是一个固定的地址，不能修改，使其指向另一个地址。

如果是字符指针，赋值运算符（`=`）只是将一个指针的地址复制给另一个指针，而不是复制字符串。

```
char* s1;
char* s2;
s1 = "abc";
s2 = s1;
```

C 语言提供了 `strcpy()` 函数，用于将一个字符串的内容复制到另一个字符串，相当于字符串赋值。该函数的原型定义在 `string.h` 头文件里面。

```
strcpy(char dest[], const char source[])
```

`strcpy()` 接受两个参数，第一个参数是目的字符串数组，第二个参数是源字符串数组。复制字符串之前，必须要保证第一个参数的长度不小于第二个参数，否则虽然不会报错，但会溢出第一个字符串变量的边界，发生难以预料的结果。第二个参数的 `const` 说明符，表示这个函数不会修改第二个字符串。

`strcpy()` 也可以用于字符数组的赋值。

```
char str[10];
strcpy(str, "abcd");
```

`strcpy()` 的返回值是一个字符串指针（即 `char*`），指向第一个参数。

`strcpy()` 返回值的另一个用途，是连续为多个字符数组赋值。

```
strcpy(str1, strcpy(str2, "abcd"));
```

上面示例调用两次 `strcpy()`，完成两个字符串变量的赋值。

`strcpy()` 的第一个参数最好是一个已经声明的数组，而不是声明后没有进行初始化的字符指针。

```
char* str;
strcpy(str, "hello world"); // 错误
```

上面的代码是有问题的。`strcpy()` 将字符串分配给指针变量 `str`，但是 `str` 并没有进行初始化，指向的是一个随机的位置，因此字符串可能被复制到任意地方。

如果不用 `strcpy()`，自己实现字符串的拷贝，可以用下面的代码。

```
char* strcpy(char* dest, const char* source) {
  char* ptr = dest;
  while (*dest++ = *source++);
  return ptr;
}
int main(void) {
```

```
  char str[25];
  strcpy(str, "hello world");
  printf("%s\n", str);
  return 0;
}
```

上面代码中，关键的一行是 `while (*dest++ = *source++)` *，这是一个循环，依次将* `source` *的每个字符赋值给* `dest` *，然后移向下一个位置，直到遇到* `\0` *，循环判断条件不再为真，从而跳出循环。其中，* `*dest++` *这个表达式等同于* `*(dest++)` *，即先返回* `dest` *这个地址，再进行自增运算移向下一个位置，而* `*dest` *可以对当前位置赋值。*

`strcpy()` *函数有安全风险，因为它并不检查目标字符串的长度，是否足够容纳源字符串的副本，可能导致写入溢出。***如果不能保证不会发生溢出，建议使用** `strncpy()` **函数代替。**

3）strncpy()函数

`strncpy()` 跟 `strcpy()` 的用法完全一样，只是多了第 3 个参数，用来指定复制的最大字符数，防止溢出目标字符串变量的边界。

```
char* strncpy(
  char* dest,
  char* src,
  size_t n
);
```

上面原型中，第三个参数 `n` 定义了复制的最大字符数。如果达到最大字符数以后，源字符串仍然没有复制完，就会停止复制，这时目的字符串结尾将没有终止符 `\0` ，这一点务必注意。

如果源字符串的字符数小于 `n` ，则 `strncpy()` 的行为与 `strcpy()` 完全一致。

`strncpy()` 也可以用来拷贝部分字符串。

```
char s1[40];
char s2[12] = "hello world";

strncpy(s1, s2, 5);
s1[5] = '\0';

printf("%s\n", s1); // hello
```

上示例中，指定只拷贝前 5 个字符。

4）strcat()函数

`strcat()` 函数用于连接字符串。它接受两个字符串作为参数，把第二个字符串的副本添加到第一个字符串的末尾。这个函数会改变第一个字符串，但是第二个字符串不变。

该函数的原型定义在 `string.h` 头文件里面。

```
char* strcat(char* s1, const char* s2);
```

`strcat()` 的返回值是一个字符串指针，指向第一个参数。

```
char s1[12] = "hello";
char s2[6] = "world";

```

```
strcat(s1, s2);
puts(s1); // "helloworld"
```

上面示例中，调用 `strcat()` 以后，可以看到字符串 `s1` 的值变了。

注意， `strcat()` 的第一个参数的长度，必须足以容纳添加第二个参数字符串。否则，拼接后的字符串会溢出第一个字符串的边界，写入相邻的内存单元，这是很危险的，建议使用下面的 `strncat()` 代替。

5）strncat()函数

`strncat()` 用于连接两个字符串，用法与 `strcat()` 完全一致，只是增加了第三个参数，指定最大添加的字符数。在添加过程中，一旦达到指定的字符数，或者在源字符串中遇到空字符 `\0` ，就不再添加了。它的原型定义在 `string.h` 头文件里面。

```
char* strncat(
  const char* dest,
  const char* src,
  size_t n
);
```

`strncat()` 返回第一个参数，即目标字符串指针。

为了保证连接后的字符串，不超过目标字符串的长度， `strncat()` 通常会写成下面这样。

```
strncat(
  str1,
  str2,
  sizeof(str1) - strlen(str1) - 1
);
```

`strncat()` 总是会在拼接结果的结尾，自动添加空字符 `\0` ，所以第三个参数的最大值，应该是 `str1` 的变量长度减去 `str1` 的字符串长度，再减去 `1` 。下面是一个用法实例。

```
char s1[10] = "Monday";
char s2[8] = "Tuesday";

strncat(s1, s2, 3);
puts(s1); // "MondayTue"
```

上面示例中， `s1` 的变量长度是 10，字符长度是 6，两者相减后再减去 1，得到 `3` ，表明 `s1` 最多可以再添加三个字符，所以得到的结果是 `MondayTue` 。

6）strcmp()函数

如果要比较两个字符串，无法直接比较，只能一个个字符进行比较，C 语言提供了 `strcmp()` 函数。

`strcmp()` 函数用于比较两个字符串的内容。该函数的原型如下，定义在 `string.h` 头文件里面。

```
int strcmp(const char* s1, const char* s2);
```

按照字典顺序，如果两个字符串相同，返回值为 `0` ；如果 `s1` 小于 `s2` ， `strcmp()` 返回值小于 0；如果 `s1` 大于 `s2` ，返回值大于 0。

下面是一个用法示例。

```
// s1 = Happy New Year
// s2 = Happy New Year
// s3 = Happy Holidays

strcmp(s1, s2) // 0
strcmp(s1, s3) // 大于 0
strcmp(s3, s1) // 小于 0
```

注意，`strcmp()` 只用来比较字符串，不用来比较字符。因为字符就是小整数，直接用相等运算符（`==`）就能比较。所以，不要把字符类型（`char`）的值，放入 `strcmp()` 当作参数。

7）strncmp()函数

由于 `strcmp()` 比较的是整个字符串，C 语言又提供了 `strncmp()` 函数，只比较到指定的位置。

该函数增加了第三个参数，指定了比较的字符数。它的原型定义在 `string.h` 头文件里面。

```
int strncmp(
  const char* s1,
  const char* s2,
  size_t n
);
```

它的返回值与 `strcmp()` 一样。如果两个字符串相同，返回值为 `0`；如果 `s1` 小于 `s2`，`strcmp()` 返回值小于 0；如果 `s1` 大于 `s2`，返回值大于 0。

下面是一个例子。

```
char s1[12] = "hello world";
char s2[12] = "hello C";

if (strncmp(s1, s2, 5) == 0) {
  printf("They all have hello.\n");
}
```

上面示例只比较两个字符串的前 5 个字符。

8）sprintf()、snprintf()函数

`sprintf()` 函数跟 `printf()` 类似，但是用于将数据写入字符串，而不是输出到显示器。该函数的原型定义在 `stdio.h` 头文件里面。

```
int sprintf(char* s, const char* format, ...);
```

`sprintf()` 的第一个参数是字符串指针变量，其余参数和 `printf()` 相同，即第二个参数是格式字符串，后面的参数是待写入的变量列表。

```
char first[6] = "hello";
char last[6] = "world";
char s[40];

sprintf(s, "%s %s", first, last);
```

```
printf("%s\n", s); // hello world
```

上面示例中，`sprintf()` 将输出内容组合成“hello world”，然后放入了变量 `s` 。

`sprintf()` 的返回值是写入变量的字符数量（不计入尾部的空字符 `\0` ）。如果遇到错误，返回负值。

`sprintf()` 有严重的安全风险，如果写入的字符串过长，超过了目标字符串的长度，`sprintf()` 依然会将其写入，导致发生溢出。为了控制写入的字符串的长度，C 语言又提供了另一个函数 `snprintf()` 。

`snprintf()` 只比 `sprintf()` 多了一个参数 `n` ，用来控制写入变量的字符串不超过 `n - 1` 个字符，剩下一个位置写入空字符 `\0` 。下面是它的原型。

```
int snprintf(char*s, size_t n, const char* format, ...);
```

`snprintf()` 总是会自动写入字符串结尾的空字符。如果你尝试写入的字符数超过指定的最大字符数，`snprintf()` 会写入 n – 1 个字符，留出最后一个位置写入空字符。

下面是一个例子。

```
snprintf(s, 12, "%s %s", "hello", "world");
```

上面的例子中，`snprintf()` 的第二个参数是 12，表示写入字符串的最大长度不超过 12（包括尾部的空字符）。

`snprintf()` 的返回值是写入格式字符串的字符数量（不计入尾部的空字符 `\0` ）。如果 `n` 足够大，返回值应该小于 `n` ，但是有时候格式字符串的长度可能大于 `n` ，那么这时返回值会大于 `n` ，但实际上真正写入变量的还是 `n-1` 个字符。如果遇到错误，返回一个负值。因此，返回值只有在非负并且小于 `n` 时，才能确认完整的格式字符串写入了变量。

4、字符串数组

如果一个数组的每个成员都是一个字符串，需要通过二维的字符数组实现。每个字符串本身是一个字符数组，多个字符串再组成一个数组。

```
char weekdays[7][10] = {
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
};
```

上面示例就是一个字符串数组，一共包含 7 个字符串，所以第一维的长度是 7。其中，最长的字符串的长度是 10（含结尾的终止符 `\0` ），所以第二维的长度统一设为 10。

因为第一维的长度，编译器可以自动计算，所以可以省略。

```
char weekdays[][10] = {
  "Monday",
  "Tuesday",
```

```
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
};
```

上面示例中，二维数组第一维的长度，可以由编译器根据后面的赋值，自动计算，所以可以不写。

数组的第二维，长度统一定为 10，有点浪费空间，因为大多数成员的长度都小于 10。解决方法就是把数组的第二维，从字符数组改成字符指针。

```
char* weekdays[] = {
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
};
```

上面的字符串数组，其实是一个一维数组，成员就是 7 个字符指针，每个指针指向一个字符串（字符数组）。

遍历字符串数组的写法如下。

```
for (int i = 0; i < 7; i++) {
  printf("%s\n", weekdays[i]);
}
```