

# C 语言入门教程

作者：阮一峰

GitHub 地址：<https://github.com/wangdoc/clang-tutorial>

整理：沉默王二

微信搜索「**沉默王二**」关注作者的原创微信公众号，回复「**PDF**」获取[上千本经典的计算机必读书籍](#)，附下载地址。

>	01、入门	昨天上午
>	02、工具	2021年
>	03、框架	2020年
>	04、数据库	2020年
>	05、并发编程	2020年
>	06、底层	2020年
>	07、性能优化	2020年
>	08、设计模式	2021年
√	09、操作系统	昨天上午
	 操作系统原理第4版.epub	2020年
	 现代操作系统原书(第3版).epub	2020年
	 《深入理解计算机系统》.pdf	2020年
	 程序是怎样跑起来的.pdf	2020年
	 计算机是怎样跑起来的.pdf	2020年
	 计算机系统概论中文版.pdf	2020年
√	10、计算机网络	昨天上午
>	Linux 网络编程	2020年
>	TCP-IP 详解	2020年
>	wireshark	2020年
	 计算机网络-自顶向下方法.pdf	2020年
	 图解HTTP[上野 宣].pdf	2020年
	 图解tcpip(第5版).pdf	2020年
	 网络是怎样连接的.pdf	2020年
>	11、数据结构与算法	2021年
>	12、面试	2021年
>	13、大数据	2020年
>	14、架构	2020年
>	15、扩展	2021年
>	16、管理	2020年
>	17、加餐	2021年
√	18、活着	2021年
	 程序员健康指南[2014.9].pdf	2020年
	 颈椎康复指南.pdf	2020年

# 1、C 语言简介

## 历史

C 语言最初是作为 Unix 系统的开发工具而发明的。

1969年，美国贝尔实验室的肯·汤普森（Ken Thompson）与丹尼斯·里奇（Dennis Ritchie）一起开发了 Unix 操作系统。Unix 是用汇编语言写的，无法移植到其他计算机，他们决定使用高级语言重写。但是，当时的高级语言无法满足他们的要求，汤普森就在 BCPL 语言的基础上发明了 B 语言。

1972年，丹尼斯·里奇和布莱恩·柯林汉（Brian Kernighan）又在 B 语言的基础上重新设计了一种新语言，这种新语言取代了 B 语言，所以称为 C 语言。

1973年，整个 Unix 系统都使用 C 语言重写。此后，这种语言开始快速流传，广泛用于各种操作系统和系统软件的开发。

1988年，美国国家标准协会（ANSI）正式将 C 语言标准化，标志着 C 语言开始稳定和规范化。

几十年后的今天，C 语言依然是最广泛使用、最流行的系统编程语言之一，Unix 和 Linux 系统现在还是使用 C 语言开发。

## C 语言的特点

C 语言能够长盛不衰、广泛应用，主要原因是它有一些鲜明的特点。

### （1）低级语言

C 语言能够直接操作硬件、管理内存、跟操作系统对话，这使得它是一种非常接近底层的语言，也就是低级语言，非常适合写需要跟硬件交互、有极高性能要求的程序。

### （2）可移植性

C 语言的原始设计目的，是将 Unix 系统移植到其他计算机架构。这使得它从一开始就非常注重可移植性，C 程序可以相对简单地移植到各种硬件架构和操作系统。

除了计算机，C 语言现在还是嵌入式系统的首选编程语言，汽车、照相机、家用电器等设备的底层系统都是用 C 语言编程，这也是因为它良好的可移植性。

### （3）简单性

C 语言的语法相对简单，语法规则不算太多，也几乎没有语法糖。一般来说，如果两个语法可以完成几乎相同的事情，C 语言就只会提供一种，这样大大减少了语言的复杂性。

而且，C 语言的语法都是基础语法，不提供高级的数据结构，比如 C 语言没有“类”（class），复杂的数据结构都需要自己构造。

#### （4）灵活性

C 语言对程序员的限制很少。它假设程序员知道自己在干嘛，不会限制你做各种危险的操作，你干什么都可以，后果也由自己负责。

C 语言的哲学是“信任程序员，不要妨碍他们做事”。比如，它让程序员自己管理内存，不提供内存自动清理功能。另外，也不提供类型检查、数组的负索引检查、指针位置的检查等保护措施。

表面上看，这似乎很危险，但是对于高级程序员来说，却有了更大的编程自由。不过，这也使得 C 语言的 debug 不太容易。

#### （5）总结

上面这些特点，使得 C 语言可以写出性能非常强、完全发挥硬件潜力的程序，而且 C 语言的编译器实现难度相对较低。但是另一方面，C 语言代码容易出错，一般程序员不容易写好。

此外，当代很多流行语言都是以 C 语言为基础，比如 C++、Java、C#、JavaScript 等等。学好 C 语言有助于对这些语言加深理解。

## C 语言的版本

历史上，C 语言有过多个版本。

#### （1）K&R C

K&R C 指的是 C 语言的原始版本。1978年，C 语言的发明者丹尼斯·里奇（Dennis Ritchie）和布莱恩·柯林（Brian Kernighan）合写了一本著名的教材《C 编程语言》（The C programming language）。由于 C 语言还没有成文的语法标准，这本书就成了公认标准，以两位作者的姓氏首字母作为版本简称“K&R C”。

#### （2）ANSI C（又称 C89 或 C90）

C 语言的原始版本非常简单，对很多情况的描述非常模糊，加上 C 语法依然在快速发展，要求将 C 语言标准化的呼声越来越高。

1989年，美国国家标准协会（ANSI）制定了一套 C 语言标准。1990年，国际标准化组织（ISO）通过了这个标准。它被称为“ANSI C”，也可以按照发布年份，称为“C89 或 C90”。

#### （3）C95

1995年，美国国家标准协会对1989年的那个标准，进行了补充，加入多字节字符和宽字符的支持。这个版

本称为 C95。

#### (4) C99

C 语言标准的第一次大型修订，发生在1999年，增加了许多语言特性，比如双斜杠（//）的注释语法。这个版本称为 C99，是目前最流行的 C 版本。

#### (5) C11

2011年，标准化组织再一次对 C 语言进行修订，增加了 Unicode 和多线程的支持。这个版本称为 C11。

#### (6) C17

C11 标准在2017年进行了修补，但发布是在2018年。新版本只是解决了 C11 的一些缺陷，没有引入任何新功能。这个版本称为 C17。

#### (7) C2x

标准化组织正在讨论 C 语言的下一个版本，据说可能会在2023年通过，到时就会称为 C23。

## C 语言的编译

C 语言是一种编译型语言，源码都是文本文件，本身无法执行。必须通过编译器，生成二进制的可执行文件，才能执行。编译器将代码从文本翻译成二进制指令的过程，就称为编译阶段，又称为“编译时”（compile time），跟运行阶段（又称为“运行时”）相区分。

目前，最常见的 C 语言编译器是自由软件基金会推出的 GCC 编译器，它可以免费使用。本书也使用这个编译器。Linux 和 Mac 系统可以直接安装 GCC，Windows 系统可以安装 MinGW。但是，也可以不用这么麻烦，网上有在线编译器，能够直接在网页上模拟运行 C 代码，查看结果，下面就是两个这样的工具。

- CodingGround: [https://tutorialspoint.com/compile\\_c\\_online.php](https://tutorialspoint.com/compile_c_online.php)
- OnlineGDB: [https://onlinegdb.com/online\\_c\\_compiler](https://onlinegdb.com/online_c_compiler)

本书的例子都使用 GCC 在命令行进行编译。

## Hello World 示例

C 语言的源代码文件，通常以后缀名 .c 结尾。下面是一个简单的 C 程序 hello.c。它就是一个普通的文本文件，任何文本编译器都能用来写。

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

上面这个程序的唯一作用，就是在屏幕上面显示“Hello World”。

这里不讲解这些代码是什么意思，只是作为一个例子，让大家看看 C 代码应该怎么编译和运行。假设你已经安装好了 GCC 编译器，可以打开命令行，执行下面的命令。

```
$ gcc hello.c
```

上面命令使用 gcc 编译器，将源文件 hello.c 编译成二进制代码。注意，\$ 是命令行提示符，你真正需要输入的是 \$ 后面的部分。

运行这个命令以后，默认会在当前目录下生成一个编译产物文件 a.out（assembler output 的缩写）。执行该文件，就会在屏幕上输出 Hello World。

```
$ ./a.out
Hello World
```

GCC 的 -o 参数可以指定编译产物的文件名。

```
$ gcc -o hello hello.c
```

上面命令的 -o hello 指定，编译产物的文件名为 hello（取代 a.out）。编译后就会生成一个名叫 hello 的可执行文件，相当于为 a.out 指定了名称。执行该文件，也会得到同样的结果。

```
$ ./hello
Hello World
```

GCC 的 -std= 参数还可以指定按照哪个 C 语言的标准进行编译。

```
$ gcc -std=c99 hello.c
```

上面命令指定按照 C99 标准进行编译。

《C 语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 沉默王二 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 2、C 语言基本语法

### 语句

C 语言的代码由一行行语句（statement）组成。语句就是程序执行的一个操作命令。C 语言规定，语句必须使用分号结尾，除非有明确规定可以不写分号。

```
int x = 1;
```

上面就是一个变量声明语句，声明整数变量 `x`，并且将值设为 `1`。

多个语句可以写在一行。

```
int x; x = 1;
```

上面示例是两个语句写在一行。所以，语句之间的换行符并不是必需的，只是为了方便阅读代码。

一个语句也可以写成多行，这时就要依靠分号判断语句在哪一行结束。

```
int x;  
x  
=  
1  
;
```

上面示例中，第二个语句 `x = 1`；被拆成了四行。编译器会自动忽略代码里面的换行。

单个分号也是有效语句，称为“空语句”，虽然毫无作用。

```
;
```

## 表达式

C 语言的各种计算，主要通过表达式完成。表达式（expression）是一个计算式，用来获取值。

```
1 + 2
```

上面代码就是一个表达式，用来获取 `1 + 2` 这个算术计算的结果。

表达式加上分号，也可以成为语句，但是没有实际的作用。

```
8;  
3 + 4;
```

上面示例是两个表达式，加上分号以后成为语句。

表达式与语句的区别主要是两点：

- 语句可以包含表达式，但是表达式本身不构成语句。
- 表达式都有返回值，语句不一定有。因为语句用来执行某个命令，很多时候不需要返回值，比如变量声明语句（`int x = 1`）就没有返回值。

## 语句块

C 语言允许多个语句使用一对大括号 `{}`，组成一个块，也称为复合语句（compounded statement）。在语法上，语句块可以视为多个语句组成的一个复合语句。



```
{  
    int x;  
    x = 1;  
}
```

上面示例中，大括号形成了一个语句块。

大括号的结尾不需要添加分号。

## 空格

C 语言里面的空格，主要用来帮助编译器区分语法单位。如果语法单位不用空格就能区分，空格就不是必须的，只是为了增加代码的可读性。

```
int x = 1;  
// 等同于  
int x=1;
```

上面示例中，赋值号（=）前后有没有空格都可以，因为编译器这里不借助空格，就能区分语法单位。

语法单位之间的多个空格，等同于单个空格。

```
int    x =    1;
```

上面示例中，各个语法单位之间的多个空格，跟单个空格的效果是一样的。

空格还用来表示缩进。多层级的代码有没有缩进，其实对于编译器来说并没有差别，没有缩进的代码也是完全可以运行的。强调代码缩进，只是为了增强代码可读性，便于区分代码块。

大多数 C 语言的风格要求是，下一级代码比上一级缩进4个空格。为了书写的紧凑，本书采用缩写两个空格。

```
// 缩进四个空格
if (x > 0)
    printf("positive\n");

// 缩进两个空格
if (x > 0)
    printf("positive\n");
```

只包含空格的行被称为空白行，编译器会完全忽略该行。

## 注释

注释是对代码的说明，编译器会忽略注释，也就是说，注释对实际代码没有影响。

C 语言的注释有两种表示方法。第一种方法是將注释放在 `/*...*/` 之间，内部可以分行。

```
/* 注释 */

/*
    这是一行注释
*/
```

这种注释可以插在行内。

```
int open(char* s /* file name */, int mode);
```

上面示例中，`/* file name */` 用来对函数参数进行说明，跟在它后面的代码依然会有效执行。

这种注释一定不能忘记写结束符号 `*/`，否则很导致错误。

```
printf("a "); /* 注释一
printf("b ");
printf("c "); /* 注释二 */
printf("d ");
```

上面示例的原意是，第一行和第三行代码的尾部，有两个注释。但是，第一行注释忘记写结束符号，导致注释一延续到第三行结束。

第二种写法是将注释放在双斜杠 `//` 后面，从双斜杠到行尾都属于注释。这种注释只能是单行，可以放在行首，也可以放在一行语句的结尾。这是 C99 标准新增的语法。

```
// 这是一行注释

int x = 1; // 这也是注释
```

不管是哪一种注释，都不能放在双引号里面。双引号里面的注释符号，会成为字符串的一部分，解释为普通符号，失去注释作用。

```
printf("// hello /* world */ ");
```

上面示例中，双引号里面的注释符号，都会被视为普通字符，没有注释作用。

编译时，注释会被替换成一个空格，所以 `min/* space */Value` 会变成 `min Value`，而不是 `minValue`。

## printf()

### 基本用法

本书的示例会大量用到 `printf()` 函数，这里先介绍一下这个函数。

`printf()` 的作用是将参数文本输出到屏幕。它名字里面的 `f` 代表 `format`（格式化），表示可以定制输出文本的格式。

```
printf("Hello World");
```

上面命令会在屏幕上输出一行文字“Hello World”。

`printf()` 不会在行尾自动添加换行符，运行结束后，光标就停留在输出结束的地方，不会自动换行。为了让光标移到下一行的开头，可以在输出文本的结尾，添加一个换行符 `\n`。

```
printf("Hello World\n");
```

如果文本内部有换行，也是通过插入换行符来实现。

```
printf("Hello\nWorld\n");
```

上面示例先输出一个 Hello ，然后换行，在下一行开头输出 World ，然后又是一个换行。

上面示例也可以写成两个 printf() ，效果完全一样。

```
printf("Hello\n");
printf("World\n");
```

printf() 是在标准库的头文件 stdio.h 定义的。使用这个函数之前，必须在源码文件头部引入这个头文件。

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
}
```

上面示例中，只有在源码头部加上 #include <stdio.h> ，才能使用 printf() 这个函数。#include 指令的详细解释，请看《预处理器》一章。

## 占位符

printf() 可以在输出文本中指定占位符。所谓“占位符”，就是这个位置可以用其他值代入。

```
// 输出 There are 3 apples
printf("There are %i apples\n", 3);
```

上面示例中，There are %i apples\n 是输出文本，里面的 %i 就是占位符，表示这个位置要用其他值来替换。占位符的第一个字符一律为百分号 % ，第二个字符表示占位符的类型，%i 表示这里代入的值必须是一个整数。

printf() 的第二个参数就是替换占位符的值，上面的例子是整数 3 替换 %i 。执行后的输出结果就是 There are 3 apples 。

常用的占位符除了 %i ，还有 %s 表示代入的是字符串。

```
printf("%s will come tonight\n", "Jane");
```

上面示例中，`%s` 表示代入的是一个字符串，所以 `printf()` 的第二个参数就必须是字符串，这个例子是 Jane 。执行后的输出就是 Jane will come tonight 。

输出文本里面可以使用多个占位符。

```
printf("%s says it is %i o'clock\n", "Ben", 21);
```

上面示例中，输出文本 `%s says it is %i o'clock` 有两个占位符，第一个是字符串占位符 `%s` ，第二个是整数占位符 `%i` ，分别对应 `printf()` 的第二个参数（ Ben ）和第三个参数（ 21 ）。执行后的输出就是 Ben says it is 21 o'clock 。

`printf()` 参数与占位符是一一对应关系，如果有 `n` 个占位符，`printf()` 的参数就应该有 `n + 1` 个。如果参数个数少于对应的占位符，`printf()` 可能会输出内存中的任意值。

`printf()` 的占位符有许多种类，与 C 语言的数据类型相对应。下面按照字母顺序，列出常用的占位符，方便查找，具体含义在后面章节介绍。

- `%a` ：浮点数。
- `%A` ：浮点数。
- `%c` ：字符。
- `%d` ：十进制整数。
- `%e` ：使用科学计数法的浮点数，指数部分的 `e` 为小写。
- `%E` ：使用科学计数法的浮点数，指数部分的 `E` 为大写。
- `%i` ：整数，基本等同于 `%d` 。
- `%f` ：小数（包含 `float` 类型和 `double` 类型）。
- `%g` ：6个有效数字的浮点数。整数部分一旦超过6位，就会自动转为科学计数法，指数部分的 `e` 为小写。
- `%G` ：等同于 `%g` ，唯一的区别是指数部分的 `E` 为大写。
- `%hd` ：十进制 short int 类型。
- `%ho` ：八进制 short int 类型。
- `%hx` ：十六进制 short int 类型。
- `%hu` ：unsigned short int 类型。
- `%ld` ：十进制 long int 类型。
- `%lo` ：八进制 long int 类型。
- `%lx` ：十六进制 long int 类型。
- `%lu` ：unsigned long int 类型。
- `%lld` ：十进制 long long int 类型。
- `%llo` ：八进制 long long int 类型。
- `%llx` ：十六进制 long long int 类型。
- `%llu` ：unsigned long long int 类型。

- `%Le`：科学计数法表示的 long double 类型浮点数。
- `%Lf`：long double 类型浮点数。
- `%n`：已输出的字符串数量。该占位符本身不输出，只将值存储在指定变量之中。
- `%o`：八进制整数。
- `%p`：指针。
- `%s`：字符串。
- `%u`：无符号整数（unsigned int）。
- `%x`：十六进制整数。
- `%zd`：size\_t 类型。
- `%%`：输出一个百分号。

## 输出格式

`printf()` 可以定制占位符的输出格式。

### (1) 限定宽度

`printf()` 允许限定占位符的最小宽度。

```
printf("%5d\n", 123); // 输出为 " 123"
```

上面示例中，`%5d` 表示这个占位符的宽度至少为5位。如果不满5位，对应的值的前面会添加空格。

输出的值默认是右对齐，即输出内容前面会有空格；如果希望改成左对齐，在输出内容后面添加空格，可以在占位符的 `%` 的后面插入一个 `-` 号。

```
printf("%-5d\n", 123); // 输出为 "123 "
```

上面示例中，输出内容 `123` 的后面添加了空格。

对于小数，这个限定符会限制所有数字的最小显示宽度。

```
// 输出 " 123.450000"  
printf("%12f\n", 123.45);
```

上面示例中，`%12f` 表示输出的浮点数最少要占据12位。由于小数的默认显示精度是小数点后6位，所以 `123.45` 输出结果的头部会添加2个空格。

### (2) 总是显示正负号

默认情况下，`printf()` 不对正数显示 + 号，只对负数显示 - 号。如果想让正数也输出 + 号，可以在占位符的 % 后面加一个 + 。

```
printf("%+d\n", 12); // 输出 +12
printf("%+d\n", -12); // 输出 -12
```

上面示例中，`%+d` 可以确保输出的数值，总是带有正负号。

### (3) 限定小数位数

输出小数时，有时希望限定小数的位数。举例来说，希望小数点后面只保留两位，占位符可以写成 `%.2f` 。

```
// 输出 Number is 0.50
printf("Number is %.2f\n", 0.5);
```

上面示例中，如果希望小数点后面输出3位（`0.500`），占位符就要写成 `%.3f` 。

这种写法可以与限定宽度占位符，结合使用。

```
// 输出为 " 0.50"
printf("%6.2f\n", 0.5);
```

上面示例中，`%6.2f` 表示输出字符串最小宽度为6，小数位数为2。所以，输出字符串的头部有两个空格。

最小宽度和小数位数这两个限定值，都可以用 \* 代替，通过 `printf()` 的参数传入。

```
printf("%*.*f\n", 6, 2, 0.5);

// 等同于
printf("%6.2f\n", 0.5);
```

上面示例中，`%*.*f` 的两个星号通过 `printf()` 的两个参数 6 和 2 传入。

### (4) 输出部分字符串

`%s` 占位符用来输出字符串，默认是全部输出。如果只想输出开头的部分，可以用 `%.[m]s` 指定输出的长度，其中 `[m]` 代表一个数字，表示所要输出的长度。

```
// 输出 hello
printf("%.5s\n", "hello world");
```

上面示例中，占位符 `%.5s` 表示只输出字符串“hello world”的前5个字符，即“hello”。

## 标准库，头文件

程序需要用到的功能，不一定需要自己编写，C 语言可能已经自带了。程序员只要去调用这些自带的功能，就省得自己编写代码了。举例来说，`printf()` 这个函数就是 C 语言自带的，只要去调用它，就能实现在屏幕上输出内容。

C 语言自带的所有这些功能，统称为“标准库”（standard library），因为它们是写入标准的，到底包括哪些功能，应该怎么使用的，都是规定好的，这样才能保证代码的规范和可移植。

不同的功能定义在不同的文件里面，这些文件统称为“头文件”（header file）。如果系统自带某一个功能，就一定还会自带描述这个功能的头文件，比如 `printf()` 的头文件就是系统自带的 `stdio.h`。头文件的后缀通常是 `.h`。

如果要使用某个功能，就必须先加载对应的头文件，加载使用的是 `#include` 命令。这就是为什么使用 `printf()` 之前，必须先加载 `stdio.h` 的原因。

```
#include <stdio.h>
```

注意，加载头文件的 `#include` 语句不需要分号结尾，详见《预处理器》一章。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！





## 3、变量

变量（variable）可以理解成一块内存区域的名字。通过变量名，可以引用这块内存区域，获取里面存储的值。由于值可能发生变化，所以称为变量，否则就是常量了。

### 变量名

变量名在 C 语言里面属于标识符（identifier），命名有严格的规范。

- 只能由字母（包括大写和小写）、数字和下划线（`_`）组成。
- 不能以数字开头。
- 长度不能超过63个字符。

下面是一些无效变量名的例子。

```
$zj
j**p
2cat
Hot-tab
tax rate
don't
```

上面示例中，每一行的变量名都是无效的。

变量名区分大小写，`star`、`Star`、`STAR` 都是不同的变量。

并非所有的词都能用作变量名，有些词在 C 语言里面有特殊含义（比如 `int`），另一些词是命令（比如 `continue`），它们都称为关键字，不能用作变量名。另外，C 语言还保留了一些词，供未来使用，这些保留字也不能用作变量名。下面就是 C 语言主要的关键字和保留字。

```
auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto,
if, inline, int, long, register, restrict, return, short, signed, sizeof, static, struct, switch, typedef,
union, unsigned, void, volatile, while
```

另外，两个下划线开头的变量名，以及一个下划线 + 大写英文字母开头的变量名，都是系统保留的，自己不应该起这样的变量名。

### 变量的声明

C 语言的变量，必须先声明后使用。如果一个变量没有声明，就直接使用，会报错。

每个变量都有自己的类型（type）。声明变量时，必须把变量的类型告诉编译器。

```
int height;
```

上面代码声明了变量 `height`，并且指定类型为 `int`（整数）。

如果几个变量具有相同类型，可以在同一行声明。

```
int height, width;
```

```
// 等同于
```

```
int height;
```

```
int width;
```

注意，声明变量的语句必须以分号结尾。

一旦声明，变量的类型就不能在运行时修改。

## 变量的赋值

C 语言会在变量声明时，就为它分配内存空间，但是不会清除内存里面原来的值。这导致声明变量以后，变量会是一个随机的值。所以，变量一定要赋值以后才能使用。

赋值操作通过赋值运算符（`=`）完成。

```
int num;  
num = 42;
```

上面示例中，第一行声明了一个整数变量 `num`，第二行给这个变量赋值。

变量的值应该与类型一致，不应该赋予不是同一个类型的值，比如 `num` 的类型是整数，就不应该赋值为小数。虽然 C 语言会自动转换类型，但是应该避免赋值运算符两侧的类型不一致。

变量的声明和赋值，也可以写在一行。

```
int num = 42;
```

多个相同类型变量的赋值，可以写在同一行。

```
int x = 1, y = 2;
```

注意，赋值表达式有返回值，等于等号右边的值。

```
int x, y;  
  
x = 1;  
y = (x = 2 * x);
```

上面代码中，变量 `y` 的值就是赋值表达式（`x = 2 * x`）的返回值 `2`。

由于赋值表达式有返回值，所以 C 语言可以写出多重赋值表达式。

```
int x, y, z, m, n;  
  
x = y = z = m = n = 3;
```

上面的代码是合法代码，一次为多个变量赋值。赋值运算符是从右到左执行，所以先为 `n` 赋值，然后依次为 `m`、`z`、`y` 和 `x` 赋值。

C 语言有左值（left value）和右值（right value）的概念。左值是可以放在赋值运算符左边的值，一般是变量；右值是可以放在赋值运算符右边的值，一般是一个具体的值。这是为了强调有些值不能放在赋值运算符的左边，比如 `x = 1` 是合法的表达式，但是 `1 = x` 就会报错。

## 变量的作用域

作用域（scope）指的是变量生效的范围。C 语言的变量作用域主要有两种：文件作用域（file scope）和块作用域（block scope）。

文件作用域（file scope）指的是，在源码文件顶层声明的变量，从声明的位置到文件结束都有效。

```
int x = 1;  
  
int main(void) {  
    printf("%i\n", x);  
}
```

上面示例中，变量 `x` 是在文件顶层声明的，从声明位置开始的整个当前文件都是它的作用域，可以在这个范围的任何地方读取这个变量，比如函数 `main()` 内部就可以读取这个变量。

块作用域（block scope）指的是由大括号（`{ }`）组成的代码块，它形成一个单独的作用域。凡是在块作用域里面声明的变量，只在当前代码块有效，代码块外部不可见。

```
int a = 12;

if (a == 12) {
    int b = 99;
    printf("%d %d\n", a, b); // 12 99
}

printf("%d\n", a); // 12
printf("%d\n", b); // 出错
```

上面例子中，变量 `b` 是在 `if` 代码块里面声明的，所以对于大括号外面的代码，这个变量是不存在的。

代码块可以嵌套，即代码块内部还有代码块，这时就形成了多层的块作用域。它的规则是：内层代码块可以使用外层声明的变量，但外层不可以使用内层声明的变量。如果内层的变量与外层同名，那么会在当前作用域覆盖外层变量。

```
{
    int i = 10;

    {
        int i = 20;
        printf("%d\n", i); // 20
    }

    printf("%d\n", i); // 10
}
```

上面示例中，内层和外层都有一个变量 `i`，每个作用域都会优先使用当前作用域声明的 `i`。

最常见的块作用域就是函数，函数内部声明的变量，对于函数外部是不可见的。`for` 循环也是一个块作用域，循环变量只对循环体内部可见，外部是不可见的。

```
for (int i = 0; i < 10; i++)  
    printf("%d\n", i);  
  
printf("%d\n", i); // 出错
```

上面示例中，for 循环省略了大括号，但依然是一个块作用域，在外部读取循环变量 i，编译器就会报错。

比较特殊的是，for 的循环条件部分是一个单独的作用域，跟循环体内部不是同一个作用域。

```
for (int i = 0; i < 5; i++) {  
    int i = 999;  
    printf("%d\n", i);  
}  
  
printf("%d\n", i); // 非法
```

上面示例中，for 的循环变量是 i，循环体内部也声明了一个变量 i，会优先读取。但由于循环条件部分是一个单独的作用域，所以循环体内部的 i 不会修改掉循环变量 i，因此这段代码的运行结果就是打印5次 999。另外，一旦 for 循环结束，循环变量 i 的作用域就消失了，变量不再存在，所以最后一行读取变量 i 就报错了。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



# 4、运算符

C 语言的运算符非常多，一共有 50 多种，可以分成若干类。

## 算术运算符

算术运算符专门用于算术运算，主要有下面几种。

- `+`：正值运算符（一元运算符）
- `-`：负值运算符（一元运算符）
- `+`：加法运算符（二元运算符）
- `-`：减法运算符（二元运算符）
- `*`：乘法运算符
- `/`：除法运算符
- `%`：余值运算符

(1) `+`，`-`

`+` 和 `-` 既可以作为一元运算符，也可以作为二元运算符。所谓“一元运算符”，指的是只需要一个运算数就可以执行。一元运算符 `-` 用来改变一个值的正负号。

```
int x = -12;
```

上面示例中，`-` 将 `12` 这个值变成 `-12`。

一元运算符 `+` 对正负值没有影响，是一个完全可以省略的运算符，但是写了也不会报错。

```
int x = -12;  
int y = +x;
```

上面示例中，变量 `y` 的值还是 `-12`，因为 `+` 不会改变正负值。

二元运算符 `+` 和 `-` 用来完成加法和减法。

```
int x = 4 + 22;  
int y = 61 - 23;
```

(2) `*`

运算符 `*` 用来完成乘法。

```
int num = 5;
printf("%i\n", num * num); // 输出 25
```

### (3) `/`

运算符 `/` 用来完成除法。注意，两个整数相除，得到还是一个整数。

```
float x = 6 / 4;
printf("%f\n", x); // 输出 1.000000
```

上面示例中，尽管变量 `x` 的类型是 `float`（浮点数），但是 `6 / 4` 得到的结果是 `1.0`，而不是 `1.5`。原因就在于 C 语言里面的整数除法是整除，只会返回整数部分，丢弃小数部分。

如果希望得到浮点数的结果，两个运算数必须至少有一个浮点数，这时 C 语言就会进行浮点数除法。

```
float x = 6.0 / 4; // 或者写成 6 / 4.0
printf("%f\n", x); // 输出 1.500000
```

上面示例中，`6.0 / 4` 表示进行浮点数除法，得到的结果就是 `1.5`。

下面是另一个例子。

```
int score = 5;
score = (score / 20) * 100;
```

上面的代码，你可能觉得经过运算，`score` 会等于 `25`，但实际上 `score` 等于 `0`。这是因为 `score / 20` 是整除，会得到一个整数值 `0`，所以乘以 `100` 后得到的也是 `0`。

为了得到预想的结果，可以将除数 `20` 改成 `20.0`，让整除变成浮点数除法。

```
score = (score / 20.0) * 100;
```

### (4) `%`

运算符 `%` 表示求模运算，即返回两个整数相除的余值。这个运算符只能用于整数，不能用于浮点数。

```
int x = 6 % 4; // 2
```

负数求模的规则是，结果的正负号由第一个运算数的正负号决定。

```
11 % -5 // 1  
-11 % -5 // -1  
-11 % 5 // -1
```

上面示例中，第一个运算数的正负号（ 11 或 -11 ）决定了结果的正负号。

#### （5）赋值运算的简写形式

如果变量对自身的值进行算术运算，C 语言提供了简写形式，允许将赋值运算符和算术运算符结合成一个运算符。

- +=
- -=
- \*=
- /=
- %=

下面是一些例子。

```
i += 3; // 等同于 i = i + 3  
i -= 8; // 等同于 i = i - 8  
i *= 9; // 等同于 i = i * 9  
i /= 2; // 等同于 i = i / 2  
i %= 5; // 等同于 i = i % 5
```

## 自增运算符，自减运算符

C 语言提供两个运算符，对变量自身进行 + 1 和 - 1 的操作。

- ++：自增运算符
- --：自减运算符

```
i++; // 等同于 i = i + 1  
i--; // 等同于 i = i - 1
```



这两个运算符放在变量的前面或后面，结果是不一样的。 `++var` 和 `--var` 是先执行自增或自减操作，再返回操作后 `var` 的值； `var++` 和 `var--` 则是先返回操作前 `var` 的值，再执行自增或自减操作。

```
int i = 42;
int j;

j = (i++ + 10);
// i: 43
// j: 52

j = (++i + 10)
// i: 44
// j: 54
```

上面示例中，自增运算符的位置差异，会导致变量 `j` 得到不同的值。这样的写法很容易出现意料之外的结果，为了消除意外，可以改用下面的写法。

```
/* 写法一 */
j = (i + 10);
i++;

/* 写法二 */
i++;
j = (i + 10);
```

上面示例中，变量 `i` 的自增运算与返回值是分离的两个步骤，这样就不太会出错，也提高了代码的可读性。

## 关系运算符

C 语言用于比较的表达式，称为“关系表达式”（relational expression），里面使用的运算符就称为“关系运算符”（relational operator），主要有下面6个。

- `>` 大于运算符
- `<` 小于运算符
- `>=` 大于等于运算符
- `<=` 小于等于运算符
- `==` 相等运算符
- `!=` 不相等运算符

下面是一些例子。

```
a == b;  
a != b;  
a < b;  
a > b;  
a <= b;  
a >= b;
```

关系表达式通常返回 0 或 1，表示真伪。C 语言中，0 表示伪，所有非零值表示真。比如，`20 > 12` 返回 1，`12 > 20` 返回 0。

关系表达式常用于 if 或 while 结构。

```
if (x == 3) {  
    printf("x is 3.\n");  
}
```

注意，相等运算符 `==` 与赋值运算符 `=` 是两个不一样的运算符，不要混淆。有时候，可能会不小心写出下面的代码，它可以运行，但很容易出现意料之外的结果。

```
if (x = 3) ...
```

上面示例中，原意是 `x == 3`，但是不小心写成 `x = 3`。这个式子表示对变量 `x` 赋值 3，它的返回值为 3，所以 if 判断总是为真。

为了防止出现这种错误，有的程序员喜欢将变量写在等号的右边。

```
if (3 == x) ...
```

这样的话，如果把 `==` 误写成 `=`，编译器就会报错。

```
/* 报错 */  
if (3 = x) ...
```

另一个需要避免的错误是，多个关系运算符不宜连用。

```
i < j < k
```

上面示例中，连续使用两个小于运算符。这是合法表达式，不会报错，但是通常达不到想要的结果，即不是保证变量 `j` 的值在 `i` 和 `k` 之间。因为表示运算符是从左到右计算，所以实际执行的是下面的表达式。

```
(i < j) < k
```

上面式子中，`i < j` 返回 `0` 或 `1`，所以最终是 `0` 或 `1` 与变量 `k` 进行比较。如果想要判断变量 `j` 的值是否在 `i` 和 `k` 之间，应该使用下面的写法。

```
i < j && j < k
```

## 逻辑运算符

逻辑运算符提供逻辑判断功能，用于构建更复杂的表达式，主要有下面三个运算符。

- `!`：否运算符（改变单个表达式的真伪）。
- `&&`：与运算符（两侧的表达式都为真，则为真，否则为伪）。
- `||`：或运算符（两侧至少有一个表达式为真，则为真，否则为伪）。

下面是与运算符的例子。

```
if (x < 10 && y > 20)
    printf("Doing something!\n");
```

上面示例中，只有 `x < 10` 和 `y > 20` 同时为真，`x < 10 && y > 20` 才会为真。

下面是否运算符的例子。

```
if (!(x < 12))
    printf("x is not less than 12\n");
```

上面示例中，由于否运算符 `!` 具有比 `<` 更高的优先级，所以必须使用括号，才能对表达式 `x < 12` 进行否运算。当然，合理的写法是 `if (x >= 12)`，这里只是为了举例。

对于逻辑运算符来说，任何非零值都表示真，零值表示伪。比如，`5 || 0` 会返回 `1`，`5 && 0` 会返回 `0`。

逻辑运算符还有一个特点，它总是先对左侧的表达式求值，再对右边的表达式求值，这个顺序是保证的。如果左边的表达式满足逻辑运算符的条件，就不再对右边的表达式求值。这种情况称为“短路”。

```
if (number != 0 && 12/number == 2)
```

上面示例中，如果 && 左侧的表达式（`number != 0`）为伪，即 `number` 等于 0 时，右侧的表达式（`12/number == 2`）是不会执行的。因为这时左侧表达式返回 0，整个 && 表达式肯定为伪，就直接返回 0，不再执行右侧的表达式了。

由于逻辑运算符的执行顺序是先左后右，所以下面的代码是有问题的。

```
while ((x++ < 10) && (x + y < 20))
```

上面示例中，执行左侧表达式后，变量 `x` 的值就已经变了。等到执行右侧表达式的时候，是用新的值在计算，这通常不是原始意图。

## 位运算符

C 语言提供一些位运算符，用来操作二进制位（bit）。

### （1）取反运算符 `~`

取反运算符 `~` 是一个一元运算符，用来将每一个二进制位变成相反值，即 0 变成 1，1 变成 0。

```
// 返回 01101100  
~ 10010011
```

上面示例中，`~` 对每个二进制位取反，就得到了一个新的值。

注意，`~` 运算符不会改变变量的值，只是返回一个新的值。

### （2）与运算符 `&`

与运算符 `&` 将两个值的每一个二进制位进行比较，返回一个新的值。当两个二进制位都为 1，就返回 1，否则返回 0。

```
// 返回 00010001  
10010011 & 00111101
```

上面示例中，两个八位二进制数进行逐位比较，返回一个新的值。

与运算符 `&` 可以与赋值运算符 `=` 结合，简写成 `&=` 。

```
int val = 3;
val = val & 0377;

// 简写成
val &= 0377;
```

### (3) 或运算符 `|`

或运算符 `|` 将两个值的每一个二进制位进行比较，返回一个新的值。两个二进制位只要有一个为 `1`（包含两个都为 `1` 的情况），就返回 `1`，否则返回 `0`。

```
// 返回 10111111
10010011 | 00111101
```

或运算符 `|` 可以与赋值运算符 `=` 结合，简写成 `|=` 。

```
int val = 3;
val = val | 0377;

// 简写为
val |= 0377;
```

### (4) 异或运算符 `^`

异或运算符 `^` 将两个值的每一个二进制位进行比较，返回一个新的值。两个二进制位有且仅有一个为 `1`，就返回 `1`，否则返回 `0`。

```
// 返回 10101110
10010011 ^ 00111101
```

异或运算符 `^` 可以与赋值运算符 `=` 结合，简写成 `^=` 。

```
int val = 3;
val = val ^ 0377;

// 简写为
val ^= 0377;
```

#### (5) 左移运算符 <<

左移运算符 << 将左侧运算数的每一位，向左移动指定的位数，尾部空出来的位置使用 0 填充。

```
// 1000101000
10001010 << 2
```

上面示例中，10001010 的每一个二进制位，都向左侧移动了两位。

左移运算符相当于将运算数乘以2的指定次方，比如左移2位相当于乘以4（2的2次方）。

左移运算符 << 可以与赋值运算符 = 结合，简写成 <<= 。

```
int val = 1;
val = val << 2;

// 简写为
val <<= 2;
```

#### (6) 右移运算符 >>

右移运算符 >> 将左侧运算数的每一位，向右移动指定的位数，尾部无法容纳的值将丢弃，头部空出来的位置使用 0 填充。

```
// 返回 00100010
10001010 >> 2
```

上面示例中，10001010 的每一个二进制位，都向右移动两位。最低的两位 10 被丢弃，头部多出来的两位补 0，所以最后得到 00100010。

注意，右移运算符最好只用于无符号整数，不要用于负数。因为不同系统对于右移后如何处理负数的符号位，有不同的做法，可能会得到不一样的结果。

右移运算符相当于将运算数除以2的指定次方，比如右移2位就相当于除以4（2的2次方）。

右移运算符 `>>` 可以与赋值运算符 `=` 结合，简写成 `>>=`。

```
int val = 1;
val = val >> 2;

// 简写为
val >>= 2;
```

## 逗号运算符

逗号运算符用于将多个表达式写在一起，从左到右依次运行每个表达式。

```
x = 10, y = 20;
```

上面示例中，有两个表达式（`x = 10` 和 `y = 20`），逗号使得它们可以放在同一条语句里面。

逗号运算符返回最后一个表达式的值，作为整个语句的值。

```
int x;
x = 1, 2, 3;
```

上面示例中，逗号的优先级低于赋值运算符，所以先执行赋值运算，再执行逗号运算，变量 `x` 等于 `1`。

## 运算优先级

优先级指的是，如果一个表达式包含多个运算符，哪个运算符应该优先执行。各种运算符的优先级是不一样的。

```
3 + 4 * 5;
```

上面示例中，表达式 `3 + 4 * 5` 里面既有加法运算符（`+`），又有乘法运算符（`*`）。由于乘法的优先级高于加法，所以会先计算 `4 * 5`，而不是先计算 `3 + 4`。

如果两个运算符优先级相同，则根据运算符是左结合，还是右结合，决定执行顺序。大部分运算符是左结合（从左到右执行），少数运算符是右结合（从右到左执行），比如赋值运算符（`=`）。

```
5 * 6 / 2;
```

上面示例中，`*` 和 `/` 的优先级相同，它们都是左结合运算符，所以从左到右执行，先计算 `5 * 6`，再计算 `6 / 2`。

运算符的优先级顺序很复杂。下面是部分运算符的优先级顺序（按照优先级从高到低排列）。

- 圆括号（`()`）
- 自增运算符（`++`），自减运算符（`--`）
- 一元运算符（`+` 和 `-`）
- 乘法（`*`），除法（`/`）
- 加法（`+`），减法（`-`）
- 关系运算符（`<`、`>` 等）
- 赋值运算符（`=`）

由于圆括号的优先级最高，可以使用它改变其他运算符的优先级。

```
int x = (3 + 4) * 5;
```

上面示例中，由于添加了圆括号，加法会先于乘法进行运算。

完全记住所有运算符的优先级没有必要，解决方法是多用圆括号，防止出现意料之外的情况，也有利于提高代码的可读性。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！





## 5、流程控制

C 语言的程序是顺序执行，即先执行前面的语句，再执行后面的语句。开发者如果想要控制程序执行的流程，就必须使用流程控制的语法结构，主要是条件执行和循环执行。

### if 语句

if 语句用于条件判断，满足条件时，就执行指定的语句。

```
if (expression) statement
```

上面式子中，表达式 expression 为真（值不为 0）时，就执行 statement 语句。

if 后面的判断条件 expression 外面必须有圆括号，否则会报错。语句体部分 statement 可以是一个语句，也可以是放在大括号里面的复合语句。下面是一个例子。

```
if (x == 10) printf("x is 10");
```

上面示例中，当变量 x 为 10 时，就会输出一行文字。对于只有一个语句的语句体，语句部分通常另起一行。

```
if (x == 10)
    printf("x is 10\n");
```

如果有多条语句，就需要把它们放在大括号里面，组成一个复合语句。

```
if (line_num == MAX_LINES) {
    line_num = 0;
    page_num++;
}
```

if 语句可以带有 else 分支，指定条件不成立时（表达式 expression 的值为 0），所要执行的代码。

```
if (expression) statement
else statement
```

下面是一个例子。

```
if (i > j)
    max = i;
else
    max = j;
```

如果 else 的语句部分多于一行，同样可以把它们放在大括号里面。

else 可以与另一个 if 语句连用，构成多重判断。

```
if (expression)
    statement
else if (expression)
    statement
...
else if (expression)
    statement
else
    statement
```

如果有多个 if 和 else，可以记住这样一条规则，else 总是跟最近的 if 匹配。

```
if (number > 6)
    if (number < 12)
        printf("The number is more than 6, less than 12.\n");
    else
        printf("It is wrong number.\n");
```

上面示例中，else 部分匹配最近的 if（即 `number < 12`），所以如果 `number` 等于6，就不会执行 else 的部分。

这样很容易出错，为了提供代码的可读性，建议使用大括号，明确 else 匹配哪一个 if。

```
if (number > 6) {
    if (number < 12) {
        printf("The number is more than 6, less than 12.\n");
    }
} else {
    printf("It is wrong number.\n");
}
```

上面示例中，使用了大括号，就可以清晰地看出 else 匹配外层的 if 。

## 三元运算符 ?:

C 语言有一个三元表达式 ?:，可以用作 if...else 的简写形式。

```
<expression1> ? <expression2> : <expression3>
```

这个操作符的含义是，表达式 expression1 如果为 true（非0值），就执行 expression2，否则执行 expression3。

下面是一个例子，返回两个值之中的较大值。

```
(i > j) ? i : j;
```

上面的代码等同于下面的 if 语句。

```
if (i > j)
    return i;
else
    return j;
```

## switch 语句

switch 语句是一种特殊形式的 if...else 结构，用于判断条件有多个结果的情况。它把多重的 else if 改成更易用、可读性更好的形式。

```
switch (expression) {  
    case value1: statement  
    case value2: statement  
    default: statement  
}
```

上面代码中，根据表达式 `expression` 不同的值，执行相应的 `case` 分支。如果找不到对应的值，就执行 `default` 分支。

下面是一个例子。

```
switch (grade) {  
    case 0:  
        printf("False");  
        break;  
    case 1:  
        printf("True");  
        break;  
    default:  
        printf("Illegal");  
}
```

上面示例中，根据变量 `grade` 不同的值，会执行不同的 `case` 分支。如果等于 `0`，执行 `case 0` 的部分；如果等于 `1`，执行 `case 1` 的部分；否则，执行 `default` 的部分。`default` 表示处理以上所有 `case` 都不匹配的情况。

每个 `case` 语句体的结尾，都应该有一个 `break` 语句，作用是跳出整个 `switch` 结构，不再往下执行。如果缺少 `break`，就会导致继续执行下一个 `case` 或 `default` 分支。

```
switch (grade) {  
    case 0:  
        printf("False");  
    case 1:  
        printf("True");  
        break;  
    default:  
        printf("Illegal");  
}
```

上面示例中，`case 0` 的部分没有 `break` 语句，导致这个分支执行完以后，不会跳出 `switch` 结构，继续执行 `case 1` 分支。

利用这个特点，如果多个 `case` 分支对应同样的语句体，可以写成下面这样。

```
switch (grade) {  
    case 0:  
    case 1:  
        printf("True");  
        break;  
    default:  
        printf("Illegal");  
}
```

上面示例中，`case 0` 分支没有任何语句，导致 `case 0` 和 `case 1` 都会执行同样的语句体。

`case` 后面的语句体，不用放在大括号里面，这也是为什么需要 `break` 的原因。

`default` 分支用来处理前面的 `case` 都不匹配的情况，最好放在所有 `case` 的后面，这样就不用写 `break` 语句。这个分支是可选的，如果没有该分支，遇到所有的 `case` 都不匹配的情况，就会直接跳出整个 `switch` 代码块。

## while 语句

`while` 语句用于循环结构，满足条件时，不断执行循环体。

```
while (expression)  
    statement
```

上面代码中，如果表达式 `expression` 为非零值（表示真），就会执行 `statement` 语句，然后再次判断 `expression` 是否为零；如果 `expression` 为零（表示伪）就跳出循环，不再执行循环体。

```
while (i < n)  
    i = i + 2;
```

上面示例中，只要 `i` 小于 `n`，`i` 就会不断增加2。

如果循环体有多个语句，就需要使用大括号将这些语句组合在一起。

```
while (expression) {  
    statement;  
    statement;  
}
```

下面是一个例子。

```
i = 0;  
  
while (i < 10) {  
    printf("i is now %d!\n", i);  
    i++;  
}  
  
printf("All done!\n");
```

上面代码中，循环体会执行10次，每次将 `i` 增加 1，直到等于 10 才退出循环。

只要条件为真，`while` 会产生无限循环。下面是一种常见的无限循环的写法。

```
while (1) {  
    // ...  
}
```

上面的示例虽然是无限循环，但是循环体内部可以用 `break` 语句跳出循环。

## do...while 结构

`do...while` 结构是 `while` 的变体，它会先执行一次循环体，然后再判断是否满足条件。如果满足的话，就继续执行循环体，否则跳出循环。

```
do statement  
while (expression);
```

上面代码中，不管条件 `expression` 是否成立，循环体 `statement` 至少会执行一次。每次 `statement` 执行完毕，就会判断一次 `expression`，决定是否结束循环。

```
i = 10;

do --i;
while (i > 0);
```

上面示例中，变量 `i` 先减去1，再判断是否大于0。如果大于0，就继续减去1，直到 `i` 等于 0 为止。

如果循环部分有多条语句，就需要放在大括号里面。

```
i = 10;

do {
    printf("i is %d\n", i);
    i++;
} while (i < 10);

printf("All done!\n");
```

上面例子中，变量 `i` 并不满足小于 10 的条件，但是循环体还是会执行一次。

## for 语句

`for` 语句是最常用的循环结构，通常用于精确控制循环次数。

```
for (initialization; continuation; action)
    statement;
```

上面代码中，`for` 语句的条件部分（即圆括号里面的部分）有三个表达式。

- `initialization`：初始化表达式，用于初始化循环变量，只执行一次。
- `continuation`：判断表达式，只要为 `true`，就会不断执行循环体。
- `action`：循环变量处理表达式，每轮循环结束后执行，使得循环变量发生变化。

循环体部分的 `statement` 可以是一条语句，也可以是放在大括号里面的复合语句。下面是一个例子。

```
for (int i = 10; i > 0; i--)
    printf("i is %d\n", i);
```

上面示例中，循环变量 `i` 在 `for` 的第一个表达式里面声明，该变量只用于本次循环。离开循环体之后，就会失效。

条件部分的三个表达式，每一个都可以有多个语句，语句与语句之间使用逗号分隔。

```
int i, j;
for (i = 0, j = 999; i < 10; i++, j--) {
    printf("%d, %d\n", i, j);
}
```

上面示例中，初始化部分有两个语句，分别对变量 `i` 和 `j` 进行赋值。

`for` 的三个表达式都不是必需的，甚至可以全部省略，这会形成无限循环。

```
for (;;) {
    printf("本行会无限循环地打印。\n" );
}
```

上面示例由于没有判断条件，就会形成无限循环。

## break 语句

`break` 语句有两种用法。一种是与 `switch` 语句配套使用，用来中断某个分支的执行，这种用法前面已经介绍过了。另一种用法是在循环体内部跳出循环，不再进行后面的循环了。

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;
    }
}
```

上面示例中，`break` 语句使得循环跳到下一个 `i` 。

```
while ((ch = getchar()) != EOF) {
    if (ch == '\n') break;
    putchar(ch);
}
```



上面示例中，一旦读到换行符（`\n`），`break` 命令就跳出整个 `while` 循环，不再继续读取了。

注意，`break` 命令只能跳出循环体和 `switch` 结构，不能跳出 `if` 结构。

```
if (n > 1) {  
    if (n > 2) break; // 无效  
    printf("hello\n");  
}
```

上面示例中，`break` 语句是无效的，因为它不能跳出外层的 `if` 结构。

## continue 语句

`continue` 语句用于在循环体内部终止本轮循环，进入下一轮循环。只要遇到 `continue` 语句，循环体内部后面的语句就不执行了，回到循环体的头部，开始执行下一轮循环。

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        printf("%d, %d\n", i, j);  
        continue;  
    }  
}
```

上面示例中，有没有 `continue` 语句，效果一样，都表示跳到下一个 `j`。

```
while ((ch = getchar()) != '\n') {  
    if (ch == '\t') continue;  
    putchar(ch);  
}
```

上面示例中，只要读到的字符是制表符（`\t`），就用 `continue` 语句跳过该字符，读取下一个字符。

## goto 语句

`goto` 语句用于跳到指定的标签名。这会破坏结构化编程，建议不要轻易使用，这里为了语法的完整，介绍一下它的用法。

```
char ch;

top: ch = getchar();

if (ch == 'q')
    goto top;
```

上面示例中，`top` 是一个标签名，可以放在正常语句的前面，相当于为这行语句做了一个标记。程序执行到 `goto` 语句，就会跳转到它指定的标签名。

```
infinite_loop:
    print("Hello, world!\n");
    goto infinite_loop;
```

上面的代码会产生无限循环。

`goto` 的一个主要用法是跳出多层循环。

```
for(...) {
    for (...) {
        while (...) {
            do {
                if (some_error_condition)
                    goto bail;
            } while(...);
        }
    }
}

bail:
// ... ..
```

上面代码有很复杂的嵌套循环，不使用 `goto` 的话，想要完全跳出所有循环，写起来很麻烦。

`goto` 的另一个用途是提早结束多重判断。

```
if (do_something() == ERR)
    goto error;
if (do_something2() == ERR)
    goto error;
if (do_something3() == ERR)
    goto error;
if (do_something4() == ERR)
    goto error;
```

上面示例有四个判断，只要有一个发现错误，就使用 goto 跳过后面的判断。

注意，goto 只能在同一个函数之中跳转，并不能跳转到其他函数。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 6、数据类型

C 语言的每一种数据，都是有类型（type）的，编译器必须知道数据的类型，才能操作数据。所谓“类型”，就是相似的数据所拥有的共同特征，那么一旦知道某个值的数据类型，就能知道该值的特征和操作方式。

基本数据类型有三种：字符（char）、整数（int）和浮点数（float）。复杂的类型都是基于它们构建的。

### 字符类型

字符类型指的是单个字符，类型声明使用 char 关键字。

```
char c = 'B';
```

上面示例声明了变量 `c` 是字符类型，并将其赋值为字母 `B`。

C 语言规定，字符常量必须放在单引号里面。

在计算机内部，字符类型使用一个字节（8位）存储。C 语言将其当作整数处理，所以字符类型就是宽度为一个字节的整数。每个字符对应一个整数（由 ASCII 码确定），比如 `B` 对应整数 `66`。

字符类型在不同计算机的默认范围是不一样的。一些系统默认为 `-128` 到 `127`，另一些系统默认为 `0` 到 `255`。这两种范围正好都能覆盖 `0` 到 `127` 的 ASCII 字符范围。

只要在字符类型的范围之内，整数与字符是可以互换的，都可以赋值给字符类型的变量。

```
char c = 66;  
// 等同于  
char c = 'B';
```

上面示例中，变量 `c` 是字符类型，赋给它的值是整数`66`。这跟赋值为字符 `B` 的效果是一样的。

两个字符类型的变量可以进行数学运算。

```
char a = 'B'; // 等同于 char a = 66;  
char b = 'C'; // 等同于 char b = 67;  
  
printf("%d\n", a + b); // 输出 133
```

上面示例中，字符类型变量 `a` 和 `b` 相加，视同两个整数相加。占位符 `%d` 表示输出十进制整数，因此输出结果为`133`。

单引号本身也是一个字符，如果要表示这个字符常量，必须使用反斜杠转义。

```
char t = '\'';
```

上面示例中，变量 `t` 为单引号字符，由于字符常量必须放在单引号里面，所以内部的单引号要使用反斜杠转义。

这种转义的写法，主要用来表示 ASCII 码定义的一些无法打印的控制字符，它们也属于字符类型的值。

- `\a`：警报，这会使得终端发出警报声或出现闪烁，或者两者同时发生。
- `\b`：退格键，光标回退一个字符，但不删除字符。
- `\f`：换页符，光标移到下一页。在现代系统上，这已经反映不出来了，行为改成类似于 `\v`。
- `\n`：换行符。
- `\r`：回车符，光标移到同一行的开头。
- `\t`：制表符，光标移到下一个水平制表位，通常是下一个8的倍数。
- `\v`：垂直分隔符，光标移到下一个垂直制表位，通常是下一行的同一列。
- `\0`：null 字符，代表没有内容。注意，这个值不等于数字0。

转义写法还能使用八进制和十六进制表示一个字符。

- `\nn`：字符的八进制写法，`nn` 为八进制值。
- `\xnn`：字符的十六进制写法，`nn` 为十六进制值。

```
char x = 'B';  
char x = 66;  
char x = '\102'; // 八进制  
char x = '\x42'; // 十六进制
```

上面示例的四种写法都是等价的。

## 整数类型

### 简介

整数类型用来表示较大的整数，类型声明使用 `int` 关键字。

```
int a;
```

上面示例声明了一个整数变量 `a`。

不同计算机的 `int` 类型的大小是不一样的。比较常见的是使用4个字节（32位）存储一个 `int` 类型的值，但是2个字节（16位）或8个字节（64位）也有可能使用。它们可以表示的整数范围如下。

- 16位：-32,768 到 32,767。
- 32位：-2,147,483,648 到 2,147,483,647。
- 64位：-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807。

## signed, unsigned

C 语言使用 `signed` 关键字，表示一个类型带有正负号，包含负值；使用 `unsigned` 关键字，表示该类型不带有正负号，只能表示零和正整数。

对于 `int` 类型，默认是带有正负号的，也就是说 `int` 等同于 `signed int`。由于这是默认情况，关键字 `signed` 一般都省略不写，但是写了也不算错。

```
signed int a;  
// 等同于  
int a;
```

`int` 类型也可以不带正负号，只表示非负整数。这时就必须使用关键字 `unsigned` 声明变量。

```
unsigned int a;
```

整数变量声明为 `unsigned` 的好处是，同样长度的内存能够表示的最大整数值，增大了一倍。比如，16位的 `signed int` 最大值为32,767，而 `unsigned int` 的最大值增大到了65,535。

`unsigned int` 里面的 `int` 可以省略，所以上面的变量声明也可以写成下面这样。

```
unsigned a;
```

字符类型 `char` 也可以设置 `signed` 和 `unsigned`。

```
signed char c; // 范围为 -128 到 127  
unsigned char c; // 范围为 0 到 255
```

注意，C 语言规定 `char` 类型默认是否带有正负号，由当前系统决定。这就是说，`char` 不等同于 `signed char`，它有可能是 `signed char`，也有可能是 `unsigned char`。这一点与 `int` 不同，`int` 就是等同于 `signed int`。

## 整数的子类型

如果 `int` 类型使用4个或8个字节表示一个整数，对于小整数，这样做很浪费空间。另一方面，某些场合需要更大的整数，8个字节还不够。为了解决这些问题，C 语言在 `int` 类型之外，又提供了三个整数的子类型。这样有利于更精细地限定整数变量的范围，也有利于更好地表达代码的意图。

- `short int`（简称为 `short`）：占用空间不多于 `int`，一般占用2个字节（整数范围为-32768~32767）。
- `long int`（简称为 `long`）：占用空间不少于 `int`，至少为4个字节。
- `long long int`（简称为 `long long`）：占用空间多于 `long`，至少为8个字节。

```
short int a;  
long int b;  
long long int c;
```

上面代码分别声明了三种整数子类型的变量。

默认情况下，`short`、`long`、`long long` 都是带符号的（signed），即 `signed` 关键字省略了。它们也可以声明为不带符号（unsigned），使得能够表示的最大值扩大一倍。

```
unsigned short int a;  
unsigned long int b;  
unsigned long long int c;
```

C 语言允许省略 `int`，所以变量声明语句也可以写成下面这样。

```
short a;  
unsigned short a;  
  
long b;  
unsigned long b;  
  
long long c;  
unsigned long long c;
```

不同的计算机，数据类型的字节长度是不一样的。确实需要32位整数时，应使用 `long` 类型而不是 `int` 类型，可以确保不少于4个字节；确实需要64位的整数时，应该使用 `long long` 类型，可以确保不少于8个字节。另一方面，为了节省空间，只需要16位整数时，应使用 `short` 类型；需要8位整数时，应该使用 `char` 类型。

## 整数类型的极限值

有时候需要查看，当前系统不同整数类型的最大值和最小值，C 语言的头文件 `limits.h` 提供了相应的常量，比如 `SCHAR_MIN` 代表 `signed char` 类型的最小值 -128，`SCHAR_MAX` 代表 `signed char` 类型的最大值 127。

为了代码的可移植性，需要知道某种整数类型的极限值时，应该尽量使用这些常量。

- `SCHAR_MIN` , `SCHAR_MAX` : `signed char` 的最小值和最大值。
- `SHRT_MIN` , `SHRT_MAX` : `short` 的最小值和最大值。
- `INT_MIN` , `INT_MAX` : `int` 的最小值和最大值。
- `LONG_MIN` , `LONG_MAX` : `long` 的最小值和最大值。
- `LLONG_MIN` , `LLONG_MAX` : `long long` 的最小值和最大值。
- `UCHAR_MAX` : `unsigned char` 的最大值。
- `USHRT_MAX` : `unsigned short` 的最大值。
- `UINT_MAX` : `unsigned int` 的最大值。
- `ULONG_MAX` : `unsigned long` 的最大值。
- `ULLONG_MAX` : `unsigned long long` 的最大值。

## 整数的进制

C 语言的整数默认都是十进制数，如果要表示八进制数和十六进制数，必须使用专门的表示法。

八进制使用 `0` 作为前缀，比如 `017` 、 `0377` 。

```
int a = 012; // 八进制，相当于十进制的10
```

十六进制使用 `0x` 或 `0X` 作为前缀，比如 `0xf` 、 `0X10` 。

```
int a = 0x1A2B; // 十六进制，相当于十进制的6699
```

有些编译器使用 `0b` 前缀，表示二进制数，但不是标准。

```
int x = 0b101010;
```

注意，不同的进制只是整数的书写方法，不会对整数的实际存储方式产生影响。所有整数都是二进制形式存储，跟书写方式无关。不同进制可以混合使用，比如 `10 + 015 + 0x20` 是一个合法的表达式。

`printf()` 的进制相关占位符如下。

- `%d` : 十进制整数。
- `%o` : 八进制整数。
- `%x` : 十六进制整数。
- `%#o` : 显示前缀 `0` 的八进制整数。
- `%#x` : 显示前缀 `0x` 的十六进制整数。



- `%#X` : 显示前缀 `0X` 的十六进制整数。

```
int x = 100;
printf("dec = %d\n", x); // 100
printf("octal = %o\n", x); // 144
printf("hex = %x\n", x); // 64
printf("octal = %#o\n", x); // 0144
printf("hex = %#x\n", x); // 0x64
printf("hex = %#X\n", x); // 0X64
```

## 浮点数类型

任何有小数点的数值，都会被编译器解释为浮点数。所谓“浮点数”就是使用  $m * b^e$  的形式，存储一个数值， $m$  是小数部分， $b$  是基数（通常是 2）， $e$  是指数部分。这种形式是精度和数值范围的一种结合，可以表示非常大或者非常小的数。

浮点数的类型声明使用 `float` 关键字，可以用来声明浮点数变量。

```
float c = 10.5;
```

上面示例中，变量 `c` 的就是浮点数类型。

`float` 类型占用4个字节（32位），其中8位存放指数的值和符号，剩下24位存放小数的值和符号。`float` 类型至少能够提供（十进制的）6位有效数字，指数部分的范围为（十进制的） $-37$  到  $37$ ，即数值范围为  $10^{-37}$  到  $10^{37}$ 。

有时候，32位浮点数提供的精度或者数值范围还不够，C 语言又提供了另外两种更大的浮点数类型。

- `double` : 占用8个字节（64位），至少提供13位有效数字。
- `long double` : 通常占用16个字节。

注意，由于存在精度限制，浮点数只是一个近似值，它的计算是不精确的，比如 C 语言里面 `0.1 + 0.2` 并不等于 `0.3`，而是有一个很小的误差。

```
if (0.1 + 0.2 == 0.3) // false
```

C 语言允许使用科学计数法表示浮点数，使用字母 `e` 来分隔小数部分和指数部分。

```
double x = 123.456e+3; // 123.456 x 10^3
// 等同于
double x = 123.456e3;
```

上面示例中，e 后面如果是加号 +，加号可以省略。注意，科学计数法里面 e 的前后，不能存在空格。

另外，科学计数法的小数部分如果是 0.x 或 x.0 的形式，那么 0 可以省略。

```
0.3E6
// 等同于
.3E6

3.0E6
// 等同于
3.E6
```

## 布尔类型

C 语言原来并没有为布尔值单独设置一个类型，而是使用整数 0 表示伪，所有非零值表示真。

```
int x = 1;
if (x) {
    printf("x is true!\n");
}
```

上面示例中，变量 x 等于 1，C 语言就认为这个值代表真，从而会执行判断体内部的代码。

C99 标准添加了类型 \_Bool，表示布尔值。但是，这个类型其实只是整数类型的别名，还是使用 0 表示伪，1 表示真，下面是一个示例。

```
_Bool isNormal;

isNormal = 1;
if (isNormal)
    printf("Everything is OK.\n");
```

头文件 stdbool.h 定义了另一个类型别名 bool，并且定义了 true 代表 1、false 代表 0。只要加载这个头文件，就可以使用这几个关键字。

```
#include <stdbool.h>

bool flag = false;
```

上面示例中，加载头文件 `stdbool.h` 以后，就可以使用 `bool` 定义布尔值类型，以及 `false` 和 `true` 表示真伪。

## 字面量的类型

字面量（literal）指的是代码里面直接出现的值。

```
int x = 123;
```

上面代码中，`x` 是变量，`123` 就是字面量。

编译时，字面量也会写入内存，因此编译器必须为字面量指定数据类型，就像必须为变量指定数据类型一样。

一般情况下，十进制整数字面量（比如 `123`）会被编译器指定为 `int` 类型。如果一个数值比较大，超出了 `int` 能够表示的范围，编译器会将其指定为 `long int`。如果数值超过了 `long int`，会被指定为 `unsigned long`。如果还不够大，就指定为 `long long` 或 `unsigned long long`。

小数（比如 `3.14`）会被指定为 `double` 类型。

## 字面量后缀

有时候，程序员希望为字面量指定一个不同的类型。比如，编译器将一个整数字面量指定为 `int` 类型，但是程序员希望将其指定为 `long` 类型，这时可以为该字面量加上后缀 `l` 或 `L`，编译器就知道要把这个字面量的类型指定为 `long`。

```
int x = 123L;
```

上面代码中，字面量 `123` 有后缀 `L`，编译器就会将其指定为 `long` 类型。这里 `123L` 写成 `123l`，效果也是一样的，但是建议优先使用 `L`，因为小写的 `l` 容易跟数字 `1` 混淆。

八进制和十六进制的值，也可以使用后缀 `l` 和 `L` 指定为 `Long` 类型，比如 `020L` 和 `0x20L`。

```
int y = 0377L;  
int z = 0x7fffL;
```

如果希望指定为无符号整数 `unsigned int`，可以使用后缀 `u` 或 `U`。

```
int x = 123U;
```

`L` 和 `U` 可以结合使用，表示 `unsigned long` 类型。`L` 和 `U` 的大小写和组合顺序无所谓。

```
int x = 123LU;
```

对于浮点数，编译器默认指定为 `double` 类型，如果希望指定为其他类型，需要在小数后面添加后缀 `f` (`float`) 或 `l` (`long double`)。

科学计数法也可以使用后缀。

```
1.2345e+10F  
1.2345e+10L
```

总结一下，常用的字面量后缀有下面这些。

- `f` 和 `F`：`float` 类型。
- `l` 和 `L`：对于整数是 `long int` 类型，对于小数是 `long double` 类型。
- `ll` 和 `LL`：`Long Long` 类型，比如 `3LL`。
- `u` 和 `U`：表示 `unsigned int`，比如 `15U`、`0377U`。

`u` 还可以与其他整数后缀结合，放在前面或后面都可以，比如 `10UL`、`10ULL` 和 `10LLU` 都是合法的。

下面是一些示例。

```
int          x = 1234;
long int     x = 1234L;
long long int x = 1234LL

unsigned int      x = 1234U;
unsigned long int x = 1234UL;
unsigned long long int x = 1234ULL;

float x      = 3.14f;
double x     = 3.14;
long double x = 3.14L;
```

## 溢出

每一种数据类型都有数值范围，如果存放的数值超出了这个范围（小于最小值或大于最大值），需要更多的二进制位存储，就会发生溢出。大于最大值，叫做向上溢出（overflow）；小于最小值，叫做向下溢出（underflow）。

一般来说，编译器不会对溢出报错，会正常执行代码，但是会忽略多出来的二进制位，只保留剩下的位，这样往往会得到意想不到的结果。所以，应该避免溢出。

```
unsigned char x = 255;
x = x + 1;

printf("%d\n", x); // 0
```

上面示例中，变量 `x` 加 1，得到的结果不是 256，而是 0。因为 `x` 是 `unsigned char` 类型，最大值是 255（二进制 11111111），加 1 后就发生了溢出，256（二进制 100000000）的最高位 1 被丢弃，剩下的值就是 0。

再看下面的例子。

```
unsigned int ui = UINT_MAX; // 4,294,967,295
ui++;
printf("ui = %u\n", ui); // 0
ui--;
printf("ui = %u\n", ui); // 4,294,967,295
```

上面示例中，常量 `UINT_MAX` 是 `unsigned int` 类型的最大值。如果加 1，对于该类型就会溢出，从而得到 0；而 0 是该类型的最小值，再减 1，又会得到 `UINT_MAX`。

溢出很容易被忽视，编译器又不会报错，所以必须非常小心。

```
for (unsigned int i = n; i >= 0; --i) // 错误
```

上面代码表面看似没有问题，但是循环变量 `i` 的类型是 `unsigned int`，这个类型的最小值是 0，不可能得到小于 0 的结果。当 `i` 等于 0，再减去 1 的时候，并不会返回 -1，而是返回 `unsigned int` 的类型最大值，这个值总是大于等于 0，导致无限循环。

为了避免溢出，最好方法就是将运算结果与类型的极限值进行比较。

```
unsigned int ui;
unsigned int sum;

// 错误
if (sum + ui > UINT_MAX) too_big();
else sum = sum + ui;

// 正确
if (ui > UINT_MAX - sum) too_big();
else sum = sum + ui;
```

上面示例中，变量 `sum` 和 `ui` 都是 `unsigned int` 类型，它们相加的和还是 `unsigned int` 类型，这就有可能发生溢出。但是，不能通过相加的和是否超出了最大值 `UINT_MAX`，来判断是否发生了溢出，因为 `sum + ui` 总是返回溢出后的结果，不可能大于 `UINT_MAX`。正确的比较方法是，判断 `UINT_MAX - sum` 与 `ui` 之间的大小关系。

下面是另一种错误的写法。

```
unsigned int i = 5;
unsigned int j = 7;

if (i - j < 0) // 错误
  printf("negative\n");
else
  printf("positive\n");
```

上面示例的运算结果，会输出 `positive` 。原因是变量 `i` 和 `j` 都是 `unsigned int` 类型，`i - j` 的结果也是这个类型，最小值为 `0` ，不可能得到小于 `0` 的结果。正确的写法是写成下面这样。

```
if (j > i) // ....
```

## sizeof 运算符

`sizeof` 是 C 语言提供的一个运算符，返回某种数据类型或某个值占用的字节数量。它的参数可以是数据类型的关键字，也可以是变量名或某个具体的值。

```
// 参数为数据类型
int x = sizeof(int);

// 参数为变量
int i;
sizeof(i);

// 参数为数值
sizeof(3.14);
```

上面的第一个示例，返回得到 `int` 类型占用的字节数量（通常是 `4` 或 `8` ）。第二个示例返回整数变量占用字节数量，结果与前一个示例完全一样。第三个示例返回浮点数 `3.14` 占用的字节数量，由于浮点数的字面量一律存储为 `double` 类型，所以会返回 `8` ，因为 `double` 类型占用的8个字节。

`sizeof` 运算符的返回值，C 语言只规定是无符号整数，并没有规定具体的类型，而是留给系统自己去决定，`sizeof` 到底返回什么类型。不同的系统中，返回值的类型有可能是 `unsigned int` ，也有可能 是 `unsigned long` ，甚至是 `unsigned long long` ，对应的 `printf()` 占位符分别是 `%u` 、`%lu` 和 `%llu` 。这样不利于程序的可移植性。

C 语言提供了一个解决方法，创造了一个类型别名 `size_t` ，用来统一表示 `sizeof` 的返回值类型。该别名定义在 `stddef.h` 头文件（引入 `stdio.h` 时会自动引入）里面，对应当前系统的 `sizeof` 的返回值类型，可能是 `unsigned int` ，也可能是 `unsigned long` 。

C 语言还提供了一个常量 `SIZE_MAX` ，表示 `size_t` 可以表示的最大整数。所以，`size_t` 能够表示的整数范围为 `[0, SIZE_MAX]` 。

`printf()` 有专门的占位符 `%zd` 或 `%zu` ，用来处理 `size_t` 类型的值。

```
printf("%zd\n", sizeof(int));
```

上面代码中，不管 `sizeof` 返回值的类型是什么，`%zd` 占位符（或 `%zu`）都可以正确输出。

如果当前系统不支持 `%zd` 或 `%zu`，可使用 `%u`（unsigned int）或 `%lu`（unsigned long int）代替。

## 类型的自动转换

某些情况下，C 语言会自动转换某个值的类型。

### 赋值运算

赋值运算符会自动将右边的值，转成左边变量的类型。

#### （1）浮点数赋值给整数变量

浮点数赋予整数变量时，C 语言直接丢弃小数部分，而不是四舍五入。

```
int x = 3.14;
```

上面示例中，变量 `x` 是整数类型，赋给它的值是一个浮点数。编译器会自动把 `3.14` 先转为 `int` 类型，丢弃小数部分，再赋值给 `x`，因此 `x` 的值是 `3`。

这种自动转换会导致部分数据的丢失（`3.14` 丢失了小数部分），所以最好不要跨类型赋值，尽量保证变量与所要赋予的值是同一个类型。

注意，舍弃小数部分时，不是四舍五入，而是整个舍弃。

```
int x = 12.99;
```

上面示例中，`x` 等于 `12`，而不是四舍五入的 `13`。

#### （2）整数赋值给浮点数变量

整数赋值给浮点数变量时，会自动转为浮点数。

```
float y = 12 * 2;
```

上面示例中，变量 `y` 的值不是 `24`，而是 `24.0`，因为等号右边的整数自动转为了浮点数。

#### （3）窄类型赋值给宽类型



字节宽度较小的整数类型，赋值给字节宽度较大的整数变量时，会发生类型提升，即窄类型自动转为宽类型。

比如，`char` 或 `short` 类型赋值给 `int` 类型，会自动提升为 `int`。

```
char x = 10;
int i = x + y;
```

上面示例中，变量 `x` 的类型是 `char`，由于赋值给 `int` 类型，所以会自动提升为 `int`。

#### （4）宽类型赋值给窄类型

字节宽度较大的类型，赋值给字节宽度较小的变量时，会发生类型降级，自动转为后者的类型。这时可能会发生截值（truncation），系统会自动截去多余的二进制位，导致难以预料的结果。

```
int i = 321;
char ch = i; // ch 的值是 65 （321 - 256）
```

上面例子中，变量 `ch` 是 `char` 类型，宽度是8个二进制位。变量 `i` 是 `int` 类型，将 `i` 赋值给 `ch`，后者只能容纳 `i`（二进制形式为 `101000001`，共9位）的后八位，前面多出来的二进制位被丢弃，保留后八位就变成了 `01000001`（十进制的65，相当于字符 `A`）。

浮点数赋值给整数类型的值，也会发生截值，浮点数的小数部分会被截去。

```
double pi = 3.14159;
int i = pi; // i 的值为 3
```

上面示例中，`i` 等于 `3`，`pi` 的小数部分被截去了。

## 混合类型的运算

不同类型的值进行混合计算时，必须先转成同一个类型，才能进行计算。转换规则如下：

（1）整数与浮点数混合运算时，整数转为浮点数类型，与另一个运算数类型相同。

```
3 + 1.2 // 4.2
```

上面示例是 `int` 类型与 `float` 类型的混合计算，`int` 类型的 3 会先转成 `float` 的 3.0，再进行计算，得到 4.2。

(2) 不同的浮点数类型混合运算时，宽度较小的类型转为宽度较大的类型，比如 `float` 转为 `double`，`double` 转为 `long double`。

(3) 不同的整数类型混合运算时，宽度较小的类型会提升为宽度较大的类型。比如 `short` 转为 `int`，`int` 转为 `long` 等，有时还会将带符号的类型 `signed` 转为无符号 `unsigned`。

下面例子的执行结果，可能会出人意料。

```
int a = -5;
if (a < sizeof(int))
    do_something();
```

上面示例中，变量 `a` 是带符号整数，`sizeof(int)` 是 `size_t` 类型，这是一个无符号整数。按照规则，`signed int` 自动转为 `unsigned int`，所以 `a` 会自动转成无符号整数 4294967291（转换规则是 -5 加上无符号整数的最大值，再加1），导致比较失败，`do_something()` 不会执行。

所以，最好避免无符号整数与有符号整数的混合运算。因为这时 C 语言会自动将 `signed int` 转为 `unsigned int`，可能不会得到预期的结果。

## 整数类型的运算

两个相同类型的整数运算时，或者单个整数的运算，一般来说，运算结果也属于同一类型。但是有一个例外，宽度小于 `int` 的类型，运算结果会自动提升为 `int`。

```
unsigned char a = 66;

if ((-a) < 0) printf("negative\n");
else printf("positive\n");
```

上面示例中，变量 `a` 是 `unsigned char` 类型，这个类型不可能小于0，但是 `-a` 会自动转为 `int` 类型，导致上面的代码输入 `negative`。

再看下面的例子。

```
unsigned char a = 1;
unsigned char b = 255;
unsigned char c = 255;

if ((a - 5) < 0) do_something();
if ((b + c) > 300) do_something();
```

上面示例中，表达式 `a - 5` 和 `b + c` 都会自动转为 `int` 类型，所以函数 `do_something()` 会执行两次。

## 函数

函数的参数和返回值，会自动转成函数定义里指定的类型。

```
int dostuff(int, unsigned char);

char m = 42;
unsigned short n = 43;
long long int c = dostuff(m, n);
```

上面示例中，参数变量 `m` 和 `n` 不管原来的类型是什么，都会转成函数 `dostuff()` 定义的参数类型。

下面是返回值自动转换类型的例子。

```
char func(void) {
    int a = 42;
    return a;
}
```

上面示例中，函数内部的变量 `a` 是 `int` 类型，但是返回的值是 `char` 类型，因为函数定义中返回的是这个类型。

## 类型的显式转换

原则上，应该避免类型的自动转换，防止出现意料之外的结果。C 语言提供了类型的显式转换，允许手动转换类型。

只要在一个值或变量的前面，使用圆括号指定类型（`type`），就可以将这个值或变量转为指定的类型，这叫做“类型指定”（`casting`）。

```
(unsigned char) ch
```

上面示例将变量 `ch` 转成无符号的字符类型。

```
long int y = (long int) 10 + 12;
```

上面示例中，`(long int)` 将 `10` 显式转为 `long int` 类型。这里的显示转换其实是不必要的，因为赋值运算符会自动将右边的值，转为左边变量的类型。

## 可移植类型

C 语言的整数类型（`short`、`int`、`long`）在不同计算机上，占用的字节宽度可能是不一样的，无法提前知道它们到底占用多少个字节。

程序员有时控制准确的字节宽度，这样的话，代码可以有更好的可移植性，头文件 `stdint.h` 创造了一些新的类型别名。

(1) 精确宽度类型(exact-width integer type)，保证某个整数类型的宽度是确定的。

- `int8_t`：8位有符号整数。
- `int16_t`：16位有符号整数。
- `int32_t`：32位有符号整数。
- `int64_t`：64位有符号整数。
- `uint8_t`：8位无符号整数。
- `uint16_t`：16位无符号整数。
- `uint32_t`：32位无符号整数。
- `uint64_t`：64位无符号整数。

上面这些都是类型别名，编译器会指定它们指向的底层类型。比如，某个系统中，如果 `int` 类型为32位，`int32_t` 就会指向 `int`；如果 `long` 类型为32位，`int32_t` 则会指向 `long`。

下面是一个使用示例。

```

#include <stdio.h>
#include <stdint.h>

int main(void) {
    int32_t x32 = 45933945;
    printf("x32 = %d\n", x32);
    return 0;
}

```

上面示例中，变量 `x32` 声明为 `int32_t` 类型，可以保证是32位的宽度。

(2) 最小宽度类型（minimum width type），保证某个整数类型的最小长度。

- `int_least8_t`
- `int_least16_t`
- `int_least32_t`
- `int_least64_t`
- `uint_least8_t`
- `uint_least16_t`
- `uint_least32_t`
- `uint_least64_t`

上面这些类型，可以保证占据的字节不少于指定宽度。比如，`int_least8_t` 表示可以容纳8位有符号整数的最小宽度的类型。

(3) 最快的最小宽度类型（fast minimum width type），可以使整数计算达到最快的类型。

- `int_fast8_t`
- `int_fast16_t`
- `int_fast32_t`
- `int_fast64_t`
- `uint_fast8_t`
- `uint_fast16_t`
- `uint_fast32_t`
- `uint_fast64_t`

上面这些类型是保证字节宽度的同时，追求最快的运算速度，比如 `int_fast8_t` 表示对于8位有符号整数，运算速度最快的类型。这是因为某些机器对于特定宽度的数据，运算速度最快，举例来说，32位计算机对于32位数据的运算速度，会快于16位数据。

(4) 可以保存指针的整数类型。

- `intptr_t`：可以存储指针（内存地址）的有符号整数类型。
- `uintptr_t`：可以存储指针的无符号整数类型。

(5) 最大宽度整数类型，用于存放最大的整数。

- `intmax_t`：可以存储任何有效的有符号整数的类型。
- `uintmax_t`：可以存放任何有效的无符号整数的类型。

上面的这两个类型的宽度比 `long long` 和 `unsigned long` 更大。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 7、指针

指针是 C 语言最重要的概念之一，也是最难理解的概念之一。

### 简介

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;
```

上面示例声明了一个变量 `intPtr`，它是一个指针，指向的内存地址存放的是一个整数。

星号 `*` 可以放在变量名与类型关键字之间的任何地方，下面的写法都是有效的。

```
int    *intPtr;
int *  intPtr;
int*   intPtr;
```

本书使用星号紧跟在类型关键字后面的写法（即 `int* intPtr;`），因为这样可以体现，指针变量就是一个普通变量，只不过它的值是内存地址而已。

这种写法有一个地方需要注意，如果同一行声明两个指针变量，那么需要写成下面这样。

```
// 正确
int * foo, * bar;

// 错误
int* foo, bar;
```

上面示例中，第二行的执行结果是，`foo` 是整数指针变量，而 `bar` 是整数变量，即 `*` 只对第一个变量生效。

一个指针指向的可能还是指针，这时就要用两个星号 `**` 表示。

```
int** foo;
```

上面示例表示变量 `foo` 是一个指针，指向的还是一个指针，第二个指针指向的则是一个整数。

## \* 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。

```
void increment(int* p) {
    *p = *p + 1;
}
```

上面示例中，函数 `increment()` 的参数是一个整数指针 `p`。函数体里面，`*p` 就表示指针 `p` 所指向的那个值。对 `*p` 赋值，就表示改变指针所指向的那个地址里面的值。

上面函数的作用是将参数值加 1。该函数没有返回值，因为传入的是地址，函数体内部对该地址包含的值的操作，会影响到函数外部，所以不需要返回值。事实上，函数内部通过指针，将值传到外部，是 C 语言的常用方法。

变量地址而不是变量值传入函数，还有一个好处。对于需要大量存储空间的大型变量，复制变量值传入函数，非常浪费时间和空间，不如传入指针来得高效。

## & 运算符

& 运算符用来取出一个变量所在的内存地址。

```
int x = 1;
printf("x's address is %p\n", &x);
```

上面示例中，x 是一个整数变量，&x 就是 x 的值所在的内存地址。printf() 的 %p 是内存地址的占位符，可以打印出内存地址。

上一小节中，参数变量加 1 的函数，可以像下面这样使用。

```
void increment(int* p) {
    *p = *p + 1;
}

int x = 1;
increment(&x);
printf("%d\n", x); // 2
```

上面示例中，调用 increment() 函数以后，变量 x 的值就增加了1，原因就在于传入函数的是变量 x 的地址 &x。

& 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
```

## 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。

## 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运算。

### （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) 指针与指针的加法运算

指针只能与整数值进行加减运算，两个指针进行加法是非法的。

```
unsigned short* j;  
unsigned short* k;  
x = j + k; // 非法
```

上面示例是两个指针相加，这是非法的。

## (3) 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("%d\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。

#### (4) 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）。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 8、函数

### 简介

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;`。

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，就像下面这样。

```
int a = plus_one(13);  
// a 等于 14
```

函数调用时，参数个数必须与定义里面的参数个数一致，参数过多或过少都会报错。

```
int plus_one(int n) {  
    return n + 1;  
}  
  
plus_one(2, 2); // 报错  
plus_one();    // 报错
```

上面示例中，函数 `plus_one()` 只能接受一个参数，传入两个参数或不传参数，都会报错。

函数必须声明后使用，否则会报错。也就是说，一定要在使用 `plus_one()` 之前，声明这个函数。如果像下面这样写，编译时会报错。

```
int a = plus_one(13);  
  
int plus_one(int n) {  
    return n + 1;  
}
```

上面示例中，在调用 `plus_one()` 之后，才声明这个函数，编译就会报错。

C 语言标准规定，函数只能声明在源码文件的顶层，不能声明在其他函数内部。

不返回值的函数，使用 `void` 关键字表示返回值的类型。没有参数的函数，声明时要用 `void` 关键字表示参数类型。

```
void myFunc(void) {  
    // ...  
}
```

上面的 `myFunc()` 函数，既没有返回值，调用时也不需要参数。

函数可以调用自身，这就叫做递归（recursion）。下面是斐波那契数列的例子。

```
unsigned long Fibonacci(unsigned n) {  
    if (n > 2)  
        return Fibonacci(n - 1) + Fibonacci(n - 2);  
    else  
        return 1;  
}
```

上面示例中，函数 `Fibonacci()` 调用了自身，大大简化了算法。

## main()

C 语言规定，`main()` 是程序的入口函数，即所有的程序一定要包含一个 `main()` 函数。程序总是从这个函数开始执行，如果没有该函数，程序就无法启动。其他函数都是通过它引入程序的。

`main()` 的写法与其他函数一样，要给出返回值的类型和参数的类型，就像下面这样。

```
int main(void) {  
    printf("Hello World\n");  
    return 0;  
}
```

上面示例中，最后的 `return 0;` 表示函数结束运行，返回 `0`。

C 语言约定，返回值 `0` 表示函数运行成功，如果返回其他非零整数，就表示运行失败，代码出了问题。系统根据 `main()` 的返回值，作为整个程序的返回值，确定程序是否运行成功。

正常情况下，如果 `main()` 里面省略 `return 0` 这一行，编译器会自动加上。所以，写成下面这样，效果完全一样。

```
int main(void) {  
    printf("Hello World"\n);  
}
```

由于 C 语言只会对 `main()` 函数默认添加返回值，对其他函数不会这样做，所以建议总是保留 `return` 语句，以便形成统一的代码风格。

## 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。

```
void increment(int a) {  
    a++;  
}  
  
int i = 10;  
increment(i);  
  
printf("%d\n", i); // 10
```

上面示例中，调用 `increment(i)` 以后，变量 `i` 本身不会发生变化，还是等于 `10`。因为传入函数的是 `i` 的拷贝，而不是 `i` 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。

```
int increment(int a) {  
    a++;  
    return a;  
}  
  
int i = 10;  
i = increment(i);  
  
printf("%d\n", i); // 11
```

再看下面的例子，`Swap()` 函数用来交换两个变量的值，由于传值引用，下面的写法不会生效。

```
void Swap(int x, int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}  
  
int a = 1;  
int b = 2;  
Swap(a, b); // 无效
```

上面的写法不会产生交换变量值的效果，因为传入的变量是原始变量 `a` 和 `b` 的拷贝，不管函数内部怎么操作，都影响不了原始变量。

如果想要传入变量本身，只有一个办法，就是传入变量的地址。

```
void Swap(int* x, int* y) {  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}  
  
int a = 1;  
int b = 2;  
Swap(&a, &b);
```

上面示例中，通过传入变量 `x` 和 `y` 的地址，函数内部就可以直接操作该地址，从而实现交换两个变量的值。

虽然跟传参无关，这里特别提一下，函数不要返回内部变量的指针。

```
int* f(void) {  
    int i;  
    // ...  
    return &i;  
}
```

上面示例中，函数返回内部变量 `i` 的指针，这种写法是错的。因为当函数结束运行时，内部变量就消失了，这时指向内部变量 `i` 的内存地址就是无效的，再去使用这个地址是非常危险的。

## 函数指针

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) = 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。

## 函数原型

前面说过，函数必须先声明，后使用。由于程序总是先运行 main() 函数，导致所有其他函数都必须在 main() 函数之前声明。

```
void func1(void) {
}

void func2(void) {
}

int main(void) {
    func1();
    func2();
    return 0;
}
```

上面代码中，`main()` 函数必须在最后声明，否则编译时会产生警告，找不到 `func1()` 或 `func2()` 的声明。

但是，`main()` 是整个程序的入口，也是主要逻辑，放在最前面比较好。另一方面，对于函数较多的程序，保证每个函数的顺序正确，会变得更麻烦。

C 语言提供的解决方法是，只要在程序开头处给出函数原型，函数就可以先使用、后声明。所谓函数原型，就是提前告诉编译器，每个函数的返回类型和参数类型。其他信息都不需要，也不用包括函数体，具体的函数实现可以后面再补上。

```
int twice(int);

int main(int num) {
    return twice(num);
}

int twice(int num) {
    return 2 * num;
}
```

上面示例中，函数 `twice()` 的实现是放在 `main()` 后面，但是代码头部先给出了函数原型，所以可以正确编译。只要提前给出函数原型，函数具体的实现放在哪里，就不重要了。

函数原型包括参数名也可以，虽然这样对于编译器是多余的，但是阅读代码的时候，可能有助于理解函数的意图。

```
int twice(int);

// 等同于
int twice(int num);
```

上面示例中，`twice` 函数的参数名 `num`，无论是否出现在原型里面，都是可以的。

注意，函数原型必须以分号结尾。

一般来说，每个源码文件的头部，都会给出当前脚本使用的所有函数的原型。

## exit()

`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（上例的 `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。

## 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。

## 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。

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

## 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。

## 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) {  
    // ...  
}
```

# 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。

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 9、数组

### 简介

数组是一组相同类型的值，按照顺序储存在一起。数组通过变量名后加方括号表示，方括号里面是数组的成员数量。

```
int scores[100];
```

上面示例声明了一个数组 `scores`，里面包含100个成员，每个成员都是 `int` 类型。

注意，声明数组时，必须给出数组的大小。

数组的成员从 0 开始编号，所以数组 `scores[100]` 就是从第0号成员一直到第99号成员，最后一个成员的编号会比数组长度小 1。

数组名后面使用方括号指定编号，就可以引用该成员。也可以通过该方式，对该位置进行赋值。

```
scores[0] = 13;  
scores[99] = 42;
```

上面示例对数组 `scores` 的第一个位置和最后一个位置，进行了赋值。



注意，如果引用不存在的数组成员（即越界访问数组），并不会报错，所以必须非常小心。

```
int scores[100];

scores[100] = 51;
```

上面示例中，数组 `scores` 只有100个成员，因此 `scores[100]` 这个位置是不存在的。但是，引用这个位置并不会报错，会正常运行，使得紧跟在 `scores` 后面的那块内存区域被赋值，而那实际上是其他变量的区域，因此不知不觉就更改了其他变量的值。这很容易引发错误，而且难以发现。

数组也可以在声明时，使用大括号，同时对每一个成员赋值。

```
int a[5] = {22, 37, 3490, 18, 95};
```

注意，使用大括号赋值时，必须在数组声明时赋值，否则编译时会报错。

```
int a[5];
a = {22, 37, 3490, 18, 95}; // 报错
```

上面代码中，数组 `a` 声明之后再进行大括号赋值，导致报错。

报错的原因是，C 语言规定，数组变量一旦声明，就不得修改变量指向的地址，具体会在后文解释。由于同样的原因，数组赋值之后，再用大括号修改值，也是不允许的。

```
int a[5] = {1, 2, 3, 4, 5};
a = {22, 37, 3490, 18, 95}; // 报错
```

上面代码中，数组 `a` 赋值后，再用大括号重新赋值也是不允许的。

使用大括号赋值时，大括号里面的值不能多于数组的长度，否则编译时会报错。

如果大括号里面的值，少于数组的成员数量，那么未赋值的成员自动初始化为 `0`。

```
int a[5] = {22, 37, 3490};
// 等同于
int a[5] = {22, 37, 3490, 0, 0};
```

如果要将整个数组的每一个成员都设置为零，最简单的写法就是下面这样。

```
int a[100] = {0};
```

数组初始化时，可以指定为哪些位置的成员赋值。

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

上面示例中，数组的2号、9号、14号位置被赋值，其他位置的值都自动设为0。

指定位置的赋值可以不按照顺序，下面的写法与上面的例子是等价的。

```
int a[15] = {[9] = 7, [14] = 48, [2] = 29};
```

指定位置的赋值与顺序赋值，可以结合使用。

```
int a[15] = {1, [5] = 10, 11, [10] = 20, 21};
```

上面示例中，0号、5号、6号、10号、11号被赋值。

C 语言允许省略方括号里面的数组成员数量，这时将根据大括号里面的值的数量，自动确定数组的长度。

```
int a[] = {22, 37, 3490};  
// 等同于  
int a[3] = {22, 37, 3490};
```

上面示例中，数组 a 的长度，将根据大括号里面的值的数量，确定为 3 。

省略成员数量时，如果同时采用指定位置的赋值，那么数组长度将是最大的指定位置再加1。

```
int a[] = {[2] = 6, [9] = 12};
```

上面示例中，数组 a 的最大指定位置是 9，所以数组的长度是10。

## 数组长度

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 。

## 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（ $10 \times 10 = 100$ ）。

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，下面的语句是上面的赋值语句是完全等价的。

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

## 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 = a + b;  
int a[n];
```

上面示例中，数组 `a` 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。

任何长度需要运行时才能确定的数组，都是变长数组。

```
int i = 10;  
  
int a1[i];  
int a2[i + 5];  
int a3[i + k];
```

上面示例中，三个数组的长度都需要运行代码才能直到，所以它们都是变长数组。

变长数组也可以用于多维数组。

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

上面示例中，`c[m][n]` 就是二维变长数组。

## 数组的地址

数组是一连串连续储存的同类型值，只要获得起始地址（首个成员的内存地址），就能推算出其他成员的地址。请看下面的例子。

```
int a[5] = {11, 22, 33, 44, 55};
int* p;

p = &a[0];

printf("%d\n", *p); // Prints "11"
```

上面示例中，`&a[0]` 就是数组 `a` 的首个成员 `11` 的内存地址，也是整个数组的起始地址。反过来，从这个地址（`*p`），可以获得首个成员的值 `11`。

由于数组的起始地址是常用操作，`&array[0]` 的写法有点麻烦，C 语言提供了便利写法，数组名等同于起始地址，也就是说，数组名就是指向第一个成员（`array[0]`）的指针。

```
int a[5] = {11, 22, 33, 44, 55};

int* p = &a[0];
// 等同于
int* p = a;
```

上面示例中，`&a[0]` 和数组名 `a` 是等价的。

这样的话，如果把数组名传入一个函数，就等同于传入一个指针变量。在函数内部，就可以通过这个指针变量获得整个数组。

函数接受数组作为参数，函数原型可以写成下面这样。

```
// 写法一
int sum(int arr[], int len);

// 写法二
int sum(int* arr, int len);
```

上面示例中，传入一个整数数组，与传入一个整数指针是同一回事，数组符号 `[]` 与指针符号 `*` 是可以互换的。下一个例子是通过数组指针对成员求和。

```

int sum(int* arr, int len) {
    int i;
    int total = 0;

    // 假定数组有 10 个成员
    for (i = 0; i < len; i++) {
        total += arr[i];
    }
    return total;
}

```

上面示例中，传入函数的是一个指针 `arr`（也是数组名）和数组长度，通过指针获取数组的每个成员，从而求和。

`*` 和 `&` 运算符也可以用于多维数组。

```

int a[4][2];

// 取出 a[0][0] 的值
*(a[0]);
// 等同于
**a

```

上面示例中，由于 `a[0]` 本身是一个指针，指向二维数组的第一个成员 `a[0][0]`。所以，`*(a[0])` 取出的是 `a[0][0]` 的值。至于 `**a`，就是对 `a` 进行两次 `*` 运算，第一次取出的是 `a[0]`，第二次取出的是 `a[0][0]`。同理，二维数组的 `&a[0][0]` 等同于 `*a`。

注意，数组名指向的地址是不能更改的。声明数组时，编译器自动为数组分配了内存地址，这个地址与数组名是绑定的，不可更改，下面的代码会报错。

```

int ints[100];
ints = NULL; // 报错

```

上面示例中，重新为数组名赋值，改变原来的内存地址，就会报错。

这也导致不能将一个数组名赋值给另外一个数组名。

```
int a[5] = {1, 2, 3, 4, 5};
```

```
// 写法一
```

```
int b[5] = a; // 报错
```

```
// 写法二
```

```
int b[5];
```

```
b = a; // 报错
```

上面两种写法都会更改数组 `b` 的地址，导致报错。

## 数组指针的加减法

C 语言里面，数组名可以进行加法和减法运算，等同于在数组成员之间前后移动，即从一个成员的内存地址移动到另一个成员的内存地址。比如，`a + 1` 返回下一个成员的地址，`a - 1` 返回上一个成员的地址。

```
int a[5] = {11, 22, 33, 44, 55};
```

```
for (int i = 0; i < 5; i++) {  
    printf("%d\n", *(a + i));  
}
```

上面示例中，通过指针的移动遍历数组，`a + i` 的每轮循环每次都会指向下一个成员的地址，`*(a + i)` 取出该地址的值，等同于 `a[i]`。对于数组的第一个成员，`*(a + 0)`（即 `*a`）等同于 `a[0]`。

由于数组名与指针是等价的，所以下面的等式总是成立。

```
a[b] == *(a + b)
```

上面代码给出了数组成员的两种访问方式，一种是使用方括号 `a[b]`，另一种是使用指针 `*(a + b)`。

如果指针变量 `p` 指向数组的一个成员，那么 `p++` 就相当于指向下一个成员，这种方法常用来遍历数组。



```
int a[] = {11, 22, 33, 44, 55, 999};

int* p = a;

while (*p != 999) {
    printf("%d\n", *p);
    p++;
}
```

上面示例中，通过 `p++` 让变量 `p` 指向下一个成员。

注意，数组名指向的地址是不能变的，所以上例中，不能直接对 `a` 进行自增，即 `a++` 的写法是错的，必须将 `a` 的地址赋值给指针变量 `p`，然后对 `p` 进行自增。

遍历数组一般都是通过数组长度的比较来实现，但也可以通过数组起始地址和结束地址的比较来实现。

```
int sum(int* start, int* end) {
    int total = 0;

    while (start < end) {
        total += *start;
        start++;
    }

    return total;
}

int arr[5] = {20, 10, 5, 39, 4};
printf("%i\n", sum(arr, arr + 5));
```

上面示例中，`arr` 是数组的起始地址，`arr + 5` 是结束地址。只要起始地址小于结束地址，就表示还没有到达数组尾部。

反过来，通过数组的减法，可以知道两个地址之间有多少个数组成员，请看下面的例子，自己实现一个计算数组长度的函数。

```
int arr[5] = {20, 10, 5, 39, 88};
int* p = arr;

while (*p != 88)
    p++;

printf("%i\n", p - arr); // 4
```

上面示例中，将某个数组成员的地址，减去数组起始地址，就可以知道，当前成员与起始地址之间有多少个成员。

对于多维数组，数组指针的加减法对于不同维度，含义是不一样的。

```
int arr[4][2];

// 指针指向 arr[1]
arr + 1;

// 指针指向 arr[0][1]
arr[0] + 1
```

上面示例中，arr 是一个二维数组，arr + 1 是将指针移动到第一维数组的下一个成员，即 arr[1]。由于每个第一维的成员，本身都包含另一个数组，即 arr[0] 是一个指向第二维数组的指针，所以 arr[0] + 1 的含义是将指针移动到第二维数组的下一个成员，即 arr[0][1]。

同一个数组的两个成员的指针相减时，返回它们之间的距离。

```
int* p = &a[5];
int* q = &a[1];

printf("%d\n", p - q); // 4
printf("%d\n", q - p); // -4
```

上面示例中，变量 p 和 q 分别是数组5号位置和1号位置的指针，它们相减等于4或-4。

## 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。

## 作为函数的参数

### 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。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



# 10、字符串

## 简介

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on知道字符串结束了。

```
char localString[10];
```

上面示例声明了一个 10 个成员的字符数组，可以当作字符串。由于必须留一个位置给 `\0`，所以最多只能容纳 9 个字符的字符串。

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"
```

上面示例中，第一行尾部的反斜杠，将字符串拆成两行。

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")
```

## 字符串变量的声明

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` 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";
```

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
```

上面代码使用指针，声明了一个字符串变量，然后修改了字符串的第一个字符。这种写法是错的，会导致难以预测的后果，执行时很可能会报错。

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？原因是声明为指针时，字符串是一个保存在内存“栈区”（stack）的常量，“栈区”的值由系统管理，一般都不允许修改；声明为数组时，字符串的副本会被拷贝到内存“堆区”（heap），“堆区”的值由用户管理，是可以修改的。

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";
```

上面示例中，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。

解决方法就是使用 C 语言原生提供的 `strcpy()` 函数，通过字符串拷贝完成赋值。

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

上面示例中，`strcpy()` 函数把字符串 `abc` 拷贝给变量 `s`，这个函数的详细用法会在后面介绍。

## strlen()

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

```
// string.h  
size_t strlen(const char* s);
```

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。下面是一个用法实例。

```
char* str = "hello";
int len = strlen(str); // 5
```

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 。

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char* s = "Hello, world!";
    printf("The string is %zd characters long.\n", strlen(s));
}
```

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
```

上面示例中，字符串长度是5，字符串变量长度是50。

如果不使用这个函数，可以通过判断字符串末尾的 \0，自己计算字符串长度。

```
int my_strlen(char *s) {
    int count = 0;
    while (s[count] != '\0')
        count++;
    return count;
}
```

## strcpy()

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;
```

上面代码可以运行，结果是两个指针变量 s1 和 s2 指向同一字符串，而不是将字符串 s2 的内容复制给 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。

```

#include <stdio.h>
#include <string.h>

int main(void) {
    char s[] = "Hello, world!";
    char t[100];

    strcpy(t, s);

    t[0] = 'z';
    printf("%s\n", s); // "Hello, world!"
    printf("%s\n", t); // "zello, world!"
}

```

上面示例将变量 `s` 的值，拷贝一份放到变量 `t`，变成两个不同的字符串，修改一个不会影响到另一个。另外，变量 `t` 的长度大于 `s`，复制后多余的位置（结束标志 `\0` 后面的位置）都为随机值。

`strcpy()` 也可以用于字符数组的赋值。

```

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

```

上面示例将字符数组变量，赋值为字符串“abcd”。

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

```

char* s1 = "beast";
char s2[40] = "Be the best that you can be.";
char* ps;

ps = strcpy(s2 + 7, s1);

puts(s2); // Be the beast
puts(ps); // beast

```

上面示例中，从 `s2` 的第7个位置开始拷贝字符串 `beast`，前面的位置不变。这导致 `s2` 后面的内容都被截去了，因为会连 `beast` 结尾的空字符一起拷贝。`strcpy()` 返回的是一个指针，指向拷贝开始的位置。

`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。

另外，`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。

## strncpy()

`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(str1, str2, sizeof(str1) - 1);  
str1[sizeof(str1) - 1] = '\0';
```

上面示例中，字符串 `str2` 复制给 `str1`，但是复制长度最多为 `str1` 的长度减去1，`str1` 剩下的最后一位用于写入字符串的结尾标志 `\0`。这是因为 `strncpy()` 不会自己添加 `\0`，如果复制的字符串片段不包含结尾标志，就需要手动添加。

`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
```

上面示例中，指定只拷贝前5个字符。

## strcat()

`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。

## strncat()

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。

```
int 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。

## strcmp()

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。

## strncmp()

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。

## sprintf(), snprintf()

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

## 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]);
}
```

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 沉默王二 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



# 11、C 语言的内存管理

## 简介

C 语言的内存管理，分成两部分。一部分是系统管理的，另一部分是用户手动管理的。

系统管理的内存，主要是函数内部的变量（局部变量）。这部分变量在函数运行时进入内存，函数运行结束后自动从内存卸载。这些变量存放的区域称为“栈”（stack），”栈“所在的内存是系统自动管理的。

用户手动管理的内存，主要是程序运行的整个过程中都存在的变量（全局变量），这些变量需要用户手动从内存释放。如果使用后忘记释放，它就一直占用内存，直到程序退出，这种情况称为”内存泄漏“（memory leak）。这些变量所在的内存称为”堆“（heap），”堆“所在的内存是用户手动管理的。

## void 指针

前面章节已经说过了，每一块内存都有地址，通过指针变量可以获取指定地址的内存块。指针变量必须有类型，否则编译器无法知道，如何解读内存块保存的二进制数据。但是，向系统请求内存的时候，有时不确定会有什么样的数据写入内存，需要先获得内存块，稍后再确定写入的数据类型。

为了满足这种需求，C 语言提供了一种不定类型的指针，叫做 void 指针。它只有内存块的地址信息，没有类型信息，等到使用该块内存的时候，再向编译器补充说明，里面的数据类型是什么。

另一方面，void 指针等同于无类型指针，可以指向任意类型的数据，但是不能解读数据。void 指针与其他所有类型指针之间是互相转换关系，任一类型的指针都可以转为 void 指针，而 void 指针也可以转为任一类型的指针。

```
int x = 10;

void* p = &x; // 整数指针转为 void 指针
int* q = p; // void 指针转为整数指针
```

上面示例中，整数指针和 void 指针可以互相转换。

注意，由于不知道 void 指针指向什么类型的值，所以不能用 \* 运算符取出它指向的值。

```
char a = 'X';
void* p = &a;

printf("%c\n", *p); // 报错
```

上面示例中，p 是一个 void 指针，所以这时无法用 \*p 取出指针指向的值。

void 指针的重要之处在于，很多内存相关函数的返回值就是 void 指针，只给出内存块的地址信息，所以放在最前面进行介绍。

## malloc()

malloc() 函数用于分配内存，该函数向系统要求一段内存，系统就在“堆”里面分配一段连续的内存块给它。它的原型定义在头文件 stdlib.h 。

```
void* malloc(size_t size)
```

它接受一个非负整数作为参数，表示所要分配的内存字节数，返回一个 void 指针，指向分配好的内存块。这是非常合理的，因为 malloc() 函数不知道，将要存储在该块内存的数据是什么类型，所以只能返回一个无类型的 void 指针。

可以使用 malloc() 为任意类型的数据分配内存，常见的做法是先使用 sizeof() 函数，算出某种数据类型所需的字节长度，然后再将这个长度传给 malloc() 。

```
int* p = malloc(sizeof(int));

*p = 12;
printf("%d\n", *p); // 12
```

上面示例中，先为整数类型分配一段内存，然后将整数 12 放入这段内存里面。这个例子其实不需要使用 malloc()，因为 C 语言会自动为整数（本例是 12）提供内存。

有时候为了增加代码的可读性，可以对 malloc() 返回的指针进行一次强制类型转换。

```
int* p = (int*) malloc(sizeof(int));
```

上面代码将 malloc() 返回的 void 指针，强制转换成了整数指针。

由于 sizeof() 的参数可以是变量，所以上面的例子也可以写成下面这样。

```
int* p = (int*) malloc(sizeof(*p));
```

malloc() 分配内存有可能分配失败，这时返回常量 NULL。Null 的值为 0，是一个无法读写的内存地址，可以理解成一个不指向任何地方的指针。它在包括 stdlib.h 等多个头文件里面都有定义，所以只要可以使用 malloc()，就可以使用 NULL。由于存在分配失败的可能，所以最好在使用 malloc() 之后检查一下，是否分配成功。

```
int* p = malloc(sizeof(int));

if (p == NULL) {
    // 内存分配失败
}

// or
if (!p) {
    // ...
}
```

上面示例中，通过判断返回的指针 p 是否为 NULL，确定 malloc() 是否分配成功。

malloc() 最常用的场合，就是为数组和自定义数据结构分配内存。

```
int* p = (int*) malloc(sizeof(int) * 10);

for (int i = 0; i < 10; i++)
    p[i] = i * 5;
```

上面示例中，`p` 是一个整数指针，指向一段可以放置10个整数的内存，所以可以用作数组。

`malloc()` 用来创建数组，有一个好处，就是它可以创建动态数组，即根据成员数量的不同，而创建长度不同的数组。

```
int* p = (int*) malloc(n * sizeof(int));
```

上面示例中，`malloc()` 可以根据变量 `n` 的不同，动态为数组分配不同的大小。

注意，`malloc()` 不会对所分配的内存进行初始化，里面还保存着原来的值。如果没有初始化，就使用这段内存，可能从里面读到以前的值。程序员要自己负责初始化，比如，字符串初始化可以使用 `strcpy()` 函数。

```
char* p = malloc(4);
strcpy(p, "abc");

// or
*p = "abc";
```

上面示例中，字符指针 `p` 指向一段4个字节的内存，`strcpy()` 将字符串“abc”拷贝放入这段内存，完成了这段内存的初始化。

## free()

`free()` 用于释放 `malloc()` 函数分配的内存，将这块内存还给系统以便重新使用，否则这个内存块会一直占用到程序运行结束。该函数的原型定义在头文件 `stdlib.h` 里面。

```
void free(void* block)
```

上面代码中，`free()` 的参数是 `malloc()` 返回的内存地址。下面就是用法实例。

```
int* p = (int*) malloc(sizeof(int));

*p = 12;
free(p);
```

注意，分配的内存块一旦释放，就不应该再次操作已经释放的地址，也不应该再次使用 `free()` 对该地址释放第二次。



一个很常见的错误是，在函数内部分配了内存，但是函数调用结束时，没有使用 `free()` 释放内存。

```
void gobble(double arr[], int n) {  
    double* temp = (double*) malloc(n * sizeof(double));  
    // ...  
}
```

上面示例中，函数 `gobble()` 内部分配了内存，但是没有写 `free(temp)`。这会造成函数运行结束后，占用的内存块依然保留，如果多次调用 `gobble()`，就会留下多个内存块。并且，由于指针 `temp` 已经消失了，也无法访问这些内存块，再次使用。

## `calloc()`

`calloc()` 函数的作用与 `malloc()` 相似，也是分配内存块。该函数的原型定义在头文件 `stdlib.h`。

两者的区别主要有两点：

- (1) `calloc()` 接受两个参数，第一个参数是数据类型的单位字节长度，第二个是该数据类型的数量。

```
void* calloc(size_t n, size_t size);
```

`calloc()` 的返回值也是一个 `void` 指针。分配失败时，返回 `NULL`。

- (2) `calloc()` 会将所分配的内存全部初始化为 `0`。`malloc()` 不会对内存进行初始化，如果想要初始化为 `0`，还要额外调用 `memset()` 函数。

```
int *p = calloc(10, sizeof(int));  
  
// 等同于  
int *q = malloc(sizeof(int) * 10);  
memset(q, 0, sizeof(int) * 10);
```

上面示例中，`calloc()` 相当于 `malloc()` + `memset()`。

`calloc()` 分配的内存块，也要使用 `free()` 释放。

## `realloc()`

`realloc()` 函数用于修改已经分配的内存块的大小，可以放大也可以缩小，返回一个指向新的内存块的指针。如果分配不成功，返回 `NULL`。该函数的原型定义在头文件 `stdlib.h`。

```
void* realloc(void* block, size_t size)
```

它接受两个参数。

- `block`：已经分配好的内存块指针（由 `malloc()` 或 `calloc()` 或 `realloc()` 产生）。
- `size`：该内存块的新大小，单位为字节。

`realloc()` 可能返回一个全新的地址（数据也会自动复制过去），也可能返回跟原来一样的地址。`realloc()` 优先在原有内存块上进行缩减，尽量不移动数据，所以通常是返回原先的地址。如果新内存块小于原来的大小，则丢弃超出的部分；如果大于原来的大小，则不对新增的部分进行初始化（程序员可以自动调用 `memset()`）。

下面是一个例子，`b` 是数组指针，`realloc()` 动态调整它的大小。

```
int* b;

b = malloc(sizeof(int) * 10);
b = realloc(b, sizeof(int) * 2000);
```

上面示例中，指针 `b` 原来指向10个成员的整数数组，使用 `realloc()` 调整为2000个成员的数组。这就是手动分配数组内存的好处，可以在运行时随时调整数组的长度。

`realloc()` 的第一个参数可以是 `NULL`，这时就相当于新建一个指针。

```
char* p = realloc(NULL, 3490);
// 等同于
char* p = malloc(3490);
```

如果 `realloc()` 的第二个参数是 `0`，就会释放掉内存块。

由于有分配失败的可能，所以调用 `realloc()` 以后，最好检查一下它的返回值是否为 `NULL`。分配失败时，原有内存块中的数据不会发生改变。

```
float* new_p = realloc(p, sizeof(*p * 40));

if (new_p == NULL) {
    printf("Error reallocing\n");
    return 1;
}
```

注意，`realloc()` 不会对内存块进行初始化。

## restrict 说明符

声明指针变量时，可以使用 `restrict` 说明符，告诉编译器，该块内存区域只有当前指针一种访问方式，其他指针不能读写该块内存。这种指针称为“受限指针”（`restrict pointer`）。

```
int* restrict p;
p = malloc(sizeof(int));
```

上面示例中，声明指针变量 `p` 时，加入了 `restrict` 说明符，使得 `p` 变成了受限指针。后面，当 `p` 指向 `malloc()` 函数返回的一块内存区域，就意味着，该区域只有通过 `p` 来访问，不存在其他访问方式。

```
int* restrict p;
p = malloc(sizeof(int));

int* q = p;
*q = 0; // 未定义行为
```

上面示例中，另一个指针 `q` 与受限指针 `p` 指向同一块内存，现在该内存有 `p` 和 `q` 两种访问方式。这就违反了对编译器的承诺，后面通过 `*q` 对该内存区域赋值，会导致未定义行为。

## memcpy()

`memcpy()` 用于将一块内存拷贝到另一块内存。该函数的原型定义在头文件 `string.h` 。

```
void* memcpy(
    void* restrict dest,
    void* restrict source,
    size_t n
);
```

上面代码中，`dest` 是目标地址，`source` 是源地址，第三个参数 `n` 是要拷贝的字节数 `n`。如果要拷贝10个 `double` 类型的数组成员，`n` 就等于 `10 * sizeof(double)`，而不是 `10`。该函数会将从 `source` 开始的 `n` 个字节，拷贝到 `dest`。

`dest` 和 `source` 都是 `void` 指针，表示这里不限制指针类型，各种类型的内存数据都可以拷贝。两者都有 `restrict` 关键字，表示这两个内存块不应该有互相重叠的区域。

`memcpy()` 的返回值是第一个参数，即目标地址的指针。

因为 `memcpy()` 只是将一段内存的值，复制到另一段内存，所以不需要知道内存里面的数据是什么类型。下面是复制字符串的例子。

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char s[] = "Goats!";
    char t[100];

    memcpy(t, s, sizeof(s)); // 拷贝7个字节，包括终止符

    printf("%s\n", t); // "Goats!"

    return 0;
}
```

上面示例中，字符串 `s` 所在的内存，被拷贝到字符数组 `t` 所在的内存。

`memcpy()` 可以取代 `strcpy()` 进行字符串拷贝，而且是更好的方法，不仅更安全，速度也更快，它不检查字符串尾部的 `\0` 字符。

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

size_t len = strlen(s) + 1;
char *c = malloc(len);

if (c) {
    // strcpy() 的写法
    strcpy(c, s);

    // memcpy() 的写法
    memcpy(c, s, len);
}
```

上面示例中，两种写法的效果完全一样，但是 `memcpy()` 的写法要好于 `strcpy()`。

使用 `void` 指针，也可以自定义一个复制内存的函数。

```
void* my_memcpy(void* dest, void* src, int byte_count) {
    char* s = src;
    char* d = dest;

    while (byte_count--) {
        *d++ = *s++;
    }

    return dest;
}
```

上面示例中，不管传入的 `dest` 和 `src` 是什么类型的指针，将它们重新定义成一字节的 `Char` 指针，这样就可以逐字节进行复制。`*d++ = *s++` 语句相当于先执行 `*d = *s`（源字节的值复制给目标字节），然后各自移动到下一个字节。最后，返回复制后的 `dest` 指针，便于后续使用。

## memmove()

`memmove()` 函数用于将一段内存数据复制到另一段内存。它跟 `memcpy()` 的主要区别是，它允许目标区域与源区域有重叠。如果发生重叠，源区域的内容会被更改；如果没有重叠，它与 `memcpy()` 行为相同。

该函数的原型定义在头文件 `string.h`。

```
void* memmove(
    void* dest,
    void* source,
    size_t n
);
```

上面代码中，`dest` 是目标地址，`source` 是源地址，`n` 是要移动的字节数。`dest` 和 `source` 都是 `void` 指针，表示可以移动任何类型的内存数据，两个内存区域可以有重叠。

`memmove()` 返回值是第一个参数，即目标地址的指针。

```
int a[100];
// ...

memmove(&a[0], &a[1], 99 * sizeof(int));
```

上面示例中，从数组成员 `a[1]` 开始的99个成员，都向前移动一个位置。

下面是另一个例子。

```
char x[] = "Home Sweet Home";

// 输出 Sweet Home Home
printf("%s\n", (char *) memmove(x, &x[5], 10));
```

上面示例中，从字符串 `x` 的5号位置开始的10个字节，就是“Sweet Home”，`memmove()` 将其前移到0号位置，所以 `x` 就变成了“Sweet Home Home”。

## memcmp()

`memcmp()` 函数用来比较两个内存区域。它的原型定义在 `string.h` 。

```
int memcmp(
    const void* s1,
    const void* s2,
    size_t n
);
```

它接受三个参数，前两个参数是用来比较的指针，第三个参数指定比较的字节数。

它的返回值是一个整数。两块内存区域的每个字节以字符形式解读，按照字典顺序进行比较，如果两者相同，返回 0；如果 s1 大于 s2，返回大于0的整数；如果 s1 小于 s2，返回小于0的整数。

```
char* s1 = "abc";  
char* s2 = "acd";  
int r = memcmp(s1, s2, 3); // 小于 0
```

上面示例比较 s1 和 s2 的前三个字节，由于 s1 小于 s2，所以 r 是一个小于0的整数，一般为-1。

下面是另一个例子。

```
char s1[] = {'b', 'i', 'g', '\\0', 'c', 'a', 'r'};  
char s2[] = {'b', 'i', 'g', '\\0', 'c', 'a', 't'};  
  
if (memcmp(s1, s2, 3) == 0) // true  
if (memcmp(s1, s2, 4) == 0) // true  
if (memcmp(s1, s2, 7) == 0) // false
```

上面示例展示了，memcmp() 可以比较内部带有字符串终止符 \0 的内存区域。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 沉默王二 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



# 12、struct 结构

## 简介

C 语言内置的数据类型，除了最基本的几种原始类型，只有数组属于复合类型，可以同时包含多个值，但是只能包含相同类型的数据，实际使用中并不够用。

实际使用中，主要有下面两种情况，需要更灵活强大的复合类型。

- 复杂的物体需要使用多个变量描述，这些变量都是相关的，最好有某种机制将它们联系起来。
- 某些函数需要传入多个参数，如果一个个按照顺序传入，非常麻烦，最好能组合成一个复合结构传入。

为了解决这些问题，C 语言提供了 `struct` 关键字，允许自定义复合数据类型，将不同类型的值组合在一起。这样不仅为编程提供方便，也有利于增强代码的可读性。C 语言没有其他语言的对象（object）和类（class）的概念，`struct` 结构很大程度上提供了对象和类的功能，可以将它视为只有属性、没有方法的类。

下面是 `struct` 自定义数据类型的一个例子。

```
struct fraction {  
    int numerator;  
    int denominator;  
};
```

上面示例定义了一个分数的数据类型 `struct fraction`，包含两个属性 `numerator` 和 `denominator`。

注意，作为一个自定义的数据类型，它的类型名要包括 `struct` 关键字，比如上例是 `struct fraction`，单独的 `fraction` 没有任何意义，甚至脚本还可以另外定义名为 `fraction` 的变量，虽然这样很容易造成混淆。另外，`struct` 语句结尾的分号不能省略，否则很容易产生错误。

定义了新的数据类型以后，就可以声明该类型的变量，这与声明其他类型变量的写法是一样的。

```
struct fraction f1;  
  
f1.numerator = 22;  
f1.denominator = 7;
```



上面示例中，先声明了一个 `struct fraction` 类型的变量 `f1`，这时编译器就会为 `f1` 分配内存，接着就可以为 `f1` 的不同属性赋值。可以看到，`struct` 结构的属性通过点（.）来表示，比如 `numerator` 属性要写成 `f1.numerator`。

再提醒一下，声明自定义类型的变量时，类型名前面，不要忘记加上 `struct` 关键字。也就是说，必须使用 `struct fraction f1` 声明变量，不能写成 `fraction f1`。

除了逐一对属性赋值，也可以使用大括号，一次性对 `struct` 结构的所有属性赋值。

```
struct car {  
    char* name;  
    float price;  
    int speed;  
};  
  
struct car saturn = {"Saturn SL/2", 16000.99, 175};
```

上面示例中，变量 `saturn` 是 `struct car` 类型，大括号里面同时对它的三个属性赋值。如果大括号里面的值的数量，少于属性的数量，那么缺失的属性自动初始化为 0。

注意，大括号里面的值的顺序，必须与 `struct` 类型声明时属性的顺序一致。否则，必须为每个值指定属性名。

```
struct car saturn = {.speed=172, .name="Saturn SL/2"};
```

上面示例中，初始化的属性少于声明时的属性，这时剩下的那些属性都会初始化为 0。

声明变量以后，可以修改某个属性的值。

```
struct car saturn = {.speed=172, .name="Saturn SL/2"};  
saturn.speed = 168;
```

上面示例将 `speed` 属性的值改成 168。

`struct` 的数据类型声明语句与变量的声明语句，可以合并为一个语句。

```
struct book {
    char title[500];
    char author[100];
    float value;
} b1;
```

上面的语句同时声明了数据类型 `book` 和该类型的变量 `b1` 。如果类型标识符 `book` 只用在这样一个地方，后面不再用到，这里可以将类型名省略。

```
struct {
    char title[500];
    char author[100];
    float value;
} b1;
```

上面示例中，`struct` 声明了一个匿名数据类型，然后又声明了这个类型的变量 `b1` 。

与其他变量声明语句一样，可以在声明变量的同时，对变量赋值。

```
struct {
    char title[500];
    char author[100];
    float value;
} b1 = {"Harry Potter", "J. K. Rowling", 10.0},
   b2 = {"Cancer Ward", "Aleksandr Solzhenitsyn", 7.85};
```

上面示例中，在声明变量 `b1` 和 `b2` 的同时，为它们赋值。

下一章介绍的 `typedef` 命令可以为 `struct` 结构指定一个别名，这样使用起来更简洁。

```
typedef struct cell_phone {
    int cell_no;
    float minutes_of_charge;
} phone;

phone p = {5551234, 5};
```

上面示例中，`phone` 就是 `struct cell_phone` 的别名。

指针变量也可以指向 struct 结构。

```
struct book {
    char title[500];
    char author[100];
    float value;
}* b1;

// 或者写成两个语句
struct book {
    char title[500];
    char author[100];
    float value;
};
struct book* b1;
```

上面示例中，变量 b1 是一个指针，指向的数据是 struct book 类型的实例。

struct 结构也可以作为数组成员。

```
struct fraction numbers[1000];

numbers[0].numerator = 22;
numbers[0].denominator = 7;
```

上面示例声明了一个有1000个成员的数组 numbers，每个成员都是自定义类型 fraction 的实例。

struct 结构占用的存储空间，不是各个属性存储空间的总和。因为为了计算效率，C 语言的内存占用空间一般来说，都必须是 int 类型存储空间的倍数。如果 int 类型的存储是4字节，那么 struct 类型的存储空间就总是4的倍数。

```
struct { char a; int b; } s;
printf("%d\n", sizeof(s)); // 8
```

上面示例中，如果按照属性占据的空间相加，变量 s 的存储空间应该是5个字节。但是，struct 结构的存储空间是 int 类型的倍数，所以最后的结果是占据8个字节，a 属性与 b 属性之间有3个字节的“空洞”。

## struct 的复制

struct 变量可以使用赋值运算符（=），复制给另一个变量，这时会生成一个全新的副本。系统会分配一块新的内存空间，大小与原来的变量相同，把每个属性都复制过去，即原样生成了一份数据。这一点跟数组的复制不一样，务必小心。

```
struct cat { char name[30]; short age; } a, b;

strcpy(a.name, "Hula");
a.age = 3;

b = a;
b.name[0] = 'M';

printf("%s\n", a.name); // Hula
printf("%s\n", b.name); // Mula
```

上面示例中，变量 b 是变量 a 的副本，两个变量的值是各自独立的，修改掉 b.name 不影响 a.name。这一点跟数组完全不同，数组使用赋值运算符，不会复制数据，只会共享地址。

但是，稍作修改，结果就不一样。

```
struct cat { char* name; short age; } a, b;

a.name = "Hula";
a.age = 3;

b = a;
```

上面示例中，name 属性变成了一个字符串指针，这时 a 赋值给 b，导致 b.name 也是同样的字符串指针，指向同一个地址，也就是说两个属性共享同一个地址。因为这时，struct 结构内部保存的是一个指针，而不是上一个例子的数组，这时复制的就不是字符串本身，而是它的指针。

注意，这种赋值要求两个变量是同一个类型，不同类型的 struct 变量无法互相赋值。

另外，C 语言没有提供比较两个自定义数据结构是否相等的方法，无法用比较运算符（比如 == 和 !=）比较两个数据结构是否相等或不等。

## struct 指针

如果将 struct 变量传入函数，函数内部得到的是一个原始值的副本。

```

#include <stdio.h>

struct turtle {
    char* name;
    char* species;
    int age;
};

void happy(struct turtle t) {
    t.age = t.age + 1;
}

int main() {
    struct turtle myTurtle = {"MyTurtle", "sea turtle", 99};
    happy(myTurtle);
    printf("Age is %i\n", myTurtle.age); // 输出 99
    return 0;
}

```

上面示例中，函数 `happy()` 传入的是一个 `struct` 变量 `myTurtle`，函数内部有一个自增操作。但是，执行完 `happy()` 以后，函数外部的 `age` 属性值根本没变。原因就是函数内部得到的是 `struct` 变量的副本，改变副本影响不到函数外部的原始数据。

通常情况下，开发者希望传入函数的是同一份数据，函数内部修改数据以后，会反映在函数外部。而且，传入的是同一份数据，也有利于提高程序性能。这时就需要将 `struct` 变量的指针传入函数，通过指针来修改 `struct` 属性，就可以影响到函数外部。

`struct` 指针传入函数的写法如下。

```

void happy(struct turtle* t) {
}

happy(&myTurtle);

```

上面代码中，`t` 是 `struct` 结构的指针，调用函数时传入的是指针。`struct` 类型跟数组不一样，类型标识符本身并不是指针，所以传入时，指针必须写成 `&myTurtle`。

函数内部也必须使用 `(*t).age` 的写法，从指针拿到 `struct` 结构本身。

```
void happy(struct turtle* t) {
    (*t).age = (*t).age + 1;
}
```

上面示例中，`(*t).age` 不能写成 `*t.age`，因为点运算符 `.` 的优先级高于 `*`。`*t.age` 这种写法会将 `t.age` 看成一个指针，然后取它对应的值，会出现无法预料的结果。

现在，重新编译执行上面的整个示例，`happy()` 内部对 `struct` 结构的操作，就会反映到函数外部。

`(*t).age` 这样的写法很麻烦。C 语言就引入了一个新的箭头运算符（`->`），可以从 `struct` 指针上直接获取属性，大大增强了代码的可读性。

```
void happy(struct turtle* t) {
    t->age = t->age + 1;
}
```

总结一下，对于 `struct` 变量名，使用点运算符（`.`）获取属性；对于 `struct` 变量指针，使用箭头运算符（`->`）获取属性。以变量 `myStruct` 为例，假设 `ptr` 是它的指针，那么下面三种写法是同一回事。

```
// ptr == &myStruct
myStruct.prop == (*ptr).prop == ptr->prop
```

## struct 的嵌套

`struct` 结构的成员可以是另一个 `struct` 结构。

```
struct species {
    char* name;
    int kinds;
};

struct fish {
    char* name;
    int age;
    struct species breed;
};
```

上面示例中，`fish` 的属性 `breed` 是另一个 `struct` 结构 `species`。

赋值的时候有多种写法。

```
// 写法一
struct fish shark = {"shark", 9, {"Selachimorpha", 500}};

// 写法二
struct breck myBreed = {"Selachimorpha", 500};
struct fish shark = {"shark", 9, myBreed};

// 写法三
struct fish shark = {
    .name="shark",
    .age=9,
    .breed={"Selachimorpha", 500}
};

// 写法四
struct fish shark = {
    .name="shark",
    .age=9,
    .breed.name="Selachimorpha",
    .breed.kinds=500
};

printf("Shark's species is %s", shark.breed.name);
```

上面示例展示了嵌套 Struct 结构的四种赋值写法。另外，引用 breed 属性的内部属性，要使用两次点运算符（ shark.breed.name ）。

下面是另一个嵌套 struct 的例子。

```
struct name {
    char first[50];
    char last[50];
};

struct student {
    struct name name;
    short age;
    char sex;
```

```

} student1;

strcpy(student1.name.first, "Harry");
strcpy(student1.name.last, "Potter");

// or
struct name myname = {"Harry", "Potter"};
student1.name = myname;

```

上面示例中，自定义类型 `student` 的 `name` 属性是另一个自定义类型，如果要引用后者的属性，就必须使用两个 `.` 运算符，比如 `student1.name.first`。另外，对字符数组属性赋值，要使用 `strcpy()` 函数，不能直接赋值，因为直接改掉字符数组名的地址会报错。

`struct` 结构内部不仅可以引用其他结构，还可以自我引用，即结构内部引用当前结构。比如，链表结构的节点就可以写成下面这样。

```

struct node {
    int data;
    struct node* next;
};

```

上面示例中，`node` 结构的 `next` 属性，就是指向另一个 `node` 实例的指针。下面，使用这个结构自定义一个数据链表。

```

struct node {
    int data;
    struct node* next;
};

struct node* head;

// 生成一个三个节点的列表 (11)->(22)->(33)
head = malloc(sizeof(struct node));

head->data = 11;
head->next = malloc(sizeof(struct node));

head->next->data = 22;
head->next->next = malloc(sizeof(struct node));

```



```
head->next->next->data = 33;
head->next->next->next = NULL;

// 遍历这个列表
for (struct node *cur = head; cur != NULL; cur = cur->next) {
    printf("%d\n", cur->data);
}
```

上面示例是链表结构的最简单实现，通过 for 循环可以对其进行遍历。

## 位字段

struct 还可以用来定义二进制位组成的数据结构，称为“位字段”（bit field），这对于操作底层的二进制数据非常有用。

```
typedef struct {
    unsigned int ab:1;
    unsigned int cd:1;
    unsigned int ef:1;
    unsigned int gh:1;
} synth;

synth.ab = 0;
synth.cd = 1;
```

上面示例中，每个属性后面的 :1，表示指定这些属性只占用一个二进制位，所以这个数据结构一共是4个二进制位。

注意，定义二进制位时，结构内部的各个属性只能是整数类型。

实际存储的时候，C 语言会按照 int 类型占用的字节数，存储一个位字段结构。如果有剩余的二进制位，可以使用未命名属性，填满那些位。也可以使用宽度为0的属性，表示占满当前字节剩余的二进制位，迫使下一个属性存储在下一个字节。

```
struct {
    unsigned int field1 : 1;
    unsigned int      : 2;
    unsigned int field2 : 1;
    unsigned int      : 0;
    unsigned int field3 : 1;
} stuff;
```

上面示例中，`stuff.field1` 与 `stuff.field2` 之间，有一个宽度为两个二进制位的未命名属性。`stuff.field3` 将存储在下一个字节。

## 弹性数组成员

很多时候，不能事先确定数组到底有多少个成员。如果声明数组的时候，事先给出一个很大的成员数，就会很浪费空间。C 语言提供了一个解决方法，叫做弹性数组成员（flexible array member）。

如果不能事先确定数组成员的数量时，可以定义一个 struct 结构。

```
struct vstring {
    int len;
    char chars[];
};
```

上面示例中，`struct vstring` 结构有两个属性。`len` 属性用来记录数组 `chars` 的长度，`chars` 属性是一个数组，但是没有给出成员数量。

`chars` 数组到底有多少个成员，可以在为 `vstring` 分配内存时确定。

```
struct vstring* str = malloc(sizeof(struct vstring) + n * sizeof(char));
str->len = n;
```

上面示例中，假定 `chars` 数组的成员数量是 `n`，只有在运行时才能知道 `n` 到底是多少。然后，就为 `struct vstring` 分配它需要的内存：它本身占用的内存长度，再加上 `n` 个数组成员占用的内存长度。最后，`len` 属性记录一下 `n` 是多少。

这样就可以让数组 `chars` 有 `n` 个成员，不用事先确定，可以跟运行时的需要保持一致。

弹性数组成员有一些专门的规则。首先，弹性成员的数组，必须是 struct 结构的最后一个属性。另外，除了弹性数组成员，struct 结构必须至少还有一个其他属性。

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 13、typedef 命令

### 简介

typedef 命令用来为某个类型起别名。

```
typedef type name;
```

上面代码中，type 代表类型名，name 代表别名。

```
typedef unsigned char BYTE;
```

```
BYTE c = 'z';
```

上面示例中，typedef 命令为类型 unsigned char 起别名 BYTE，然后就可以使用 BYTE 声明变量。

typedef 可以一次指定多个别名。

```
typedef int antelope, bagel, mushroom;
```

上面示例中，一次性为 int 类型起了三个别名。

typedef 可以为指针起别名。

```
typedef int* intptr;

int a = 10;
intptr x = &a;
```

上面示例中，`intptr` 是 `int*` 的别名。不过，使用的时候要小心，这样不容易看出来，变量 `x` 是一个指针类型。

typedef 也可以用来为数组类型起别名。

```
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

上面示例中，`five_ints` 是一个数组类型，包含5个整数的

typedef 为函数起别名的写法如下。

```
typedef signed char (*fp)(void);
```

上面示例中，类型别名 `fp` 是一个指针，代表函数 `signed char (*)(void)`。

## 主要好处

typedef 为类型起别名的好处，主要有下面几点。

(1) 更好的代码可读性。

```
typedef char* STRING;

STRING name;
```

上面示例为字符指针起别名为 `STRING`，以后使用 `STRING` 声明变量时，就可以轻易辨别该变量是字符串。

(2) 为 `struct`、`union`、`enum` 等命令定义的复杂数据结构创建别名，从而便于引用。

```
struct treeNode {  
    // ...  
};  
  
typedef struct treeNode* Tree;
```

上面示例中，`Tree` 为 `struct treeNode*` 的别名。

`typedef` 也可以与 `struct` 定义数据类型的命令写在一起。

```
typedef struct animal {  
    char* name;  
    int leg_count, speed;  
} animal;
```

上面示例中，自定义数据类型时，同时使用 `typedef` 命令，为 `struct animal` 起了一个别名 `animal`。

这种情况下，C 语言允许省略 `struct` 命令后面的类型名。

```
typedef struct {  
    char *name;  
    int leg_count, speed;  
} animal;
```

上面示例相当于为一个匿名的数据类型起了别名 `animal`。

(3) `typedef` 方便以后为变量改类型。

```
typedef float app_float;  
  
app_float f1, f2, f3;
```

上面示例中，变量 `f1`、`f2`、`f3` 的类型都是 `float`。如果以后需要为它们改类型，只需要修改 `typedef` 语句即可。

```
typedef long double app_float;
```

上面命令将变量 `f1` 、 `f2` 、 `f3` 的类型都改为 `long double` 。

#### (4) 可移植性

某一个值在不同计算机上的类型，可能是不一样的。

```
int i = 100000;
```

上面代码在32位整数的计算机没有问题，但是在16位整数的计算机就会出错。

C 语言的解决办法，就是提供了类型别名，在不同计算机上会解释成不同类型，比如 `int32_t` 。

```
int32_t i = 100000;
```

上面示例将变量 `i` 声明成 `int32_t` 类型，保证它在不同计算机上都是32位宽度，移植代码时就不会出错。

这一类的类型别名都是用 `typedef` 定义的。下面是类似的例子。

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

这些整数类型别名都放在头文件 `stdint.h` ，不同架构的计算机只需修改这个头文件即可，而无需修改代码。

因此，`typedef` 有助于提高代码的可移植性，使其能适配不同架构的计算机。

#### (5) 简化类型声明

C 语言有些类型声明相当复杂，比如下面这个。

```
char ((*x(void))[5])(void);
```

`typedef` 可以简化复杂的类型声明，使其更容易理解。首先，最外面一层起一个类型别名。

```
typedef char (*Func)(void);
Func (*x(void))[5];
```

这个看起来还是有点复杂，就为里面一层也定义一个别名。

```
typedef char (*Func)(void);  
typedef Func Arr[5];  
Arr* x(void);
```

上面代码就比较容易解读了。

- x 是一个函数，返回一个指向 Arr 类型的指针。
- Arr 是一个数组，有5个成员，每个成员是 Func 类型。
- Func 是一个函数指针，指向一个无参数、返回字符值的函数。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 14、Union 结构

有时需要一种数据结构，不同的场合表示不同的数据类型。比如，如果只用一种数据结构表示水果的“量”，这种结构就需要有时是整数（6个苹果），有时是浮点数（1.5公斤草莓）。

C 语言提供了 Union 结构，用来自定义可以灵活变更的数据结构。它内部可以包含各种属性，但同一时间只能有一个属性，因为所有属性都保存在同一个内存地址，后面写入的属性会覆盖前面的属性。这样做的最大好处是节省空间。

```
union quantity {  
    short count;  
    float weight;  
    float volume;  
};
```

上面示例中，`union` 命令定义了一个包含三个属性的数据类型 `quantity`。虽然包含三个属性，但是同一时间只能取到一个属性。最后赋值的属性，就是可以取到值的那个属性。

使用时，声明一个该类型的变量。

```
// 写法一  
union quantity q;  
q.count = 4;  
  
// 写法二  
union quantity q = {.count=4};  
  
// 写法三  
union quantity q = {4};
```

上面代码展示了为 Union 结构赋值的三种写法。最后一种写法不指定属性名，就会赋值给第一个属性。

执行完上面的代码以后，`q.count` 可以取到值，另外两个属性取不到值。

```
printf("count is %i\n", q.count); // count is 4  
printf("weight is %f\n", q.weight); // 未定义行为
```

如果要想 `q.weight` 属性可以取到值，就要先为它赋值。

```
q.weight = 0.5;  
printf("weight is %f\n", q.weight); // weight is 0.5
```

一旦为其他属性赋值，原先可以取到值的 `q.count` 属性就不再有效了。除了这一点，Union 结构的其他用法与 Struct 结构，基本上是一致的。

Union 结构也支持指针运算符 `->`。





```
typedef union {  
    short count;  
    float weight;  
    float volume;  
} quantity;
```

上面示例中，union 命令定义了一个包含三个属性的数据类型，typedef 命令为它起别名为 quantity。

Union 结构的好处，主要是节省空间。它将一段内存空间，重用于不同类型的数据。定义了三个属性，但同一时间只用到一个，使用 Union 结构就可以节省另外两个属性的空间。Union 结构占用的内存长度，等于它内部最长属性的长度。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 15、Enum 类型

如果一种数据类型的取值只有少数几种可能，并且每种取值都有自己的含义，为了提高代码的可读性，可以将它们定义为 Enum 类型，中文名为枚举。

```
enum colors {RED, GREEN, BLUE};  
  
printf("%d\n", RED); // 0  
printf("%d\n", GREEN); // 1  
printf("%d\n", BLUE); // 2
```

上面示例中，假定程序里面需要三种颜色，就可以使用 `enum` 命令，把这三种颜色定义成一种枚举类型 `colors`，它只有三种取值可能 `RED`、`GREEN`、`BLUE`。这时，这三个名字自动成为整数常量，编译器默认将它们的值设为数字 `0`、`1`、`2`。相比之下，`RED` 要比 `0` 的可读性好了许多。

注意，Enum 内部的常量名，遵守标识符的命名规范，但是通常都使用大写。

使用时，可以将变量声明为 Enum 类型。

```
enum colors color;
```

上面代码将变量 `color` 声明为 `enum colors` 类型。这个变量的值就是常量 `RED`、`GREEN`、`BLUE` 之中的一个。

```
color = BLUE;  
printf("%i\n", color); // 2
```

上面代码将变量 `color` 的值设为 `BLUE`，这里 `BLUE` 就是一个常量，值等于 `2`。

`typedef` 命令可以为 Enum 类型起别名。

```
typedef enum {  
    SHEEP,  
    WHEAT,  
    WOOD,  
    BRICK,  
    ORE  
} RESOURCE;  
  
RESOURCE r;
```

上面示例中，`RESOURCE` 是 Enum 类型的别名。声明变量时，使用这个别名即可。

还有一种不常见的写法，就是声明 Enum 类型时，在同一行里面为变量赋值。

```
enum {  
    SHEEP,  
    WHEAT,  
    WOOD,  
    BRICK,  
    ORE  
} r = BRICK, s = WOOD;
```

上面示例中，`r` 的值是 3，`s` 的值是 2。

由于 Enum 的属性会自动声明为常量，所以有时候使用 Enum 的目的，不是为了自定义一种数据类型，而是为了声明一组常量。这时就可以使用下面这种写法，比较简单。

```
enum { ONE, TWO };  
  
printf("%d %d", ONE, TWO); // 0 1
```

上面示例中，`enum` 是一个关键字，后面跟着一个代码块，常量就在代码内声明。`ONE` 和 `TWO` 就是两个 Enum 常量。

常量之间使用逗号分隔。最后一个常量后面的尾逗号，可以省略，也可以保留。

```
enum { ONE, TWO, };
```

由于Enum 会自动编号，因此可以不必为常量赋值。C 语言会自动从0开始递增，为常量赋值。但是，C 语言也允许为 ENUM 常量指定值，不过只能指定为整数，不能是其他类型。因此，任何可以使用整数的场合，都可以使用 Enum 常量。

```
enum { ONE = 1, TWO = 2 };  
  
printf("%d %d", ONE, TWO); // 1 2
```

Enum 常量可以是不连续的值。

```
enum { X = 2, Y = 18, Z = -2 };
```

Enum 常量也可以是同一个值。

```
enum { X = 2, Y = 2, Z = 2 };
```

如果一组常量之中，有些指定了值，有些没有指定。那么，没有指定值的常量会从上一个指定了值的常量，开始自动递增赋值。

```
enum {  
    A,      // 0  
    B,      // 1  
    C = 4,  // 4  
    D,      // 5  
    E,      // 6  
    F = 3,  // 3  
    G,      // 4  
    H      // 5  
}
```

Enum 的作用域与变量相同。如果是在顶层声明，那么在整个文件内都有效；如果是在代码块内部声明，则只对该代码块有效。如果与使用 int 声明的常量相比，Enum 的好处是更清晰地表示代码意图。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的話，欢迎转发分享！



## 16、预处理器 (Preprocessor)

# 简介

C 语言编译器在编译程序之前，会先使用预处理器（preprocessor）处理代码。

预处理器首先会清理代码，进行删除注释、多行的语句合成一个逻辑行等等。然后，执行 # 开头的预处理指令。本章介绍 C 语言的预处理指令。

预处理指令可以出现在程序的任何地方，但是习惯上，往往放在代码的开头部分。

每个预处理指令都以 # 开头，放在一行的行首，指令前面可以有空白字符（比如空格或制表符）。# 和指令的其余部分之间也可以有空格，但是为了兼容老的编译器，一般不留空格。

所有预处理指令都是一行的，除非在行尾使用反斜杠，将其折行。指令结尾处不需要分号。

## #define

#define 是最常见的预处理指令，用来将指定的词替换成另一个词。它的参数分成两个部分，第一个参数就是要被替换的部分，其余参数是替换后的内容。每条替换规则，称为一个宏（macro）。

```
#define MAX 100
```

上面示例中，#define 指定将源码里面的 MAX，全部替换成 100。MAX 就称为一个宏。

宏的名称不允许有空格，而且必须遵守 C 语言的变量命名规则，只能使用字母、数字与下划线（\_），且首字符不能是数字。

宏是原样替换，指定什么内容，就一模一样替换成什么内容。

```
#define HELLO "Hello, world"

// 相当于 printf("%s", "Hello, world");
printf("%s", HELLO);
```

上面示例中，宏 HELLO 会被原样替换成 "Hello, world"。

#define 指令可以出现在源码文件的任何地方，从指令出现的地方到该文件末尾都有效。习惯上，会将#define 放在源码文件的头部。它的主要好处是，会使得程序的可读性更好，也更容易修改。

#define 指令从 # 开始，一直到换行符为止。如果整条指令过长，可以在折行处使用反斜杠，延续到下一行。

```
#define OW "C programming language is invented \
in 1970s."
```

上面示例中，第一行结尾的反斜杠将 `#define` 指令拆成两行。

`#define` 允许多重替换，即一个宏可以包含另一个宏。

```
#define TWO 2
#define FOUR TWO*TWO
```

上面示例中，`FOUR` 会被替换成 `2*2`。

注意，如果宏出现在字符串里面（即出现在双引号中），或者是其他标识符的一部分，就会失效，并不会发生替换。

```
#define TWO 2

// 输出 TWO
printf("TWO\n");

// 输出 22
const TWOs = 22;
printf("%d\n", TWOs);
```

上面示例中，双引号里面的 `TWO`，以及标识符 `TWOs`，都不会被替换。

同名的宏可以重复定义，只要定义是相同的，就没有问题。如果定义不同，就会报错。

```
// 正确
#define FOO hello
#define FOO hello

// 报错
#define BAR hello
#define BAR world
```

上面示例中，宏 `FOO` 没有变化，所以可以重复定义，宏 `BAR` 发生了变化，就报错了。

# 带参数的宏

## 基本用法

宏的强大之处在于，它的名称后面可以使用括号，指定接受一个或多个参数。

```
#define SQUARE(X) X*X
```

上面示例中，宏 `SQUARE` 可以接受一个参数 `X`，替换成 `X*X`。

注意，宏的名称与左边圆括号之间，不能有空格。

这个宏的用法如下。

```
// 替换成 z = 2*2;  
z = SQUARE(2);
```

这种写法很像函数，但又不是函数，而是完全原样的替换，会跟函数有不一样的行为。

```
#define SQUARE(X) X*X  
  
// 输出19  
printf("%d\n", SQUARE(3 + 4));
```

上面示例中，`SQUARE(3 + 4)` 如果是函数，输出的应该是49（`7*7`）；宏是原样替换，所以替换成 `3 + 4*3 + 4`，最后输出19。

可以看到，原样替换可能导致意料之外的行为。解决办法就是在定义宏的时候，尽量多使用圆括号，这样可以避免很多意外。

```
#define SQUARE(X) ((X) * (X))
```

上面示例中，`SQUARE(X)` 替换后的形式，有两层圆括号，就可以避免很多错误的发生。

宏的参数也可以是空的。

```
#define getchar() getc(stdin)
```



上面示例中，宏 `getchar()` 的参数就是空的。这种情况其实可以省略圆括号，但是加上了，会让它看上去更像函数。

一般来说，带参数的宏都是一行的。下面是两个例子。

```
#define MAX(x, y) ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

如果宏的长度过长，可以使用反斜杠（\）折行，将宏写成多行。

```
#define PRINT_NUMS_TO_PRODUCT(a, b) { \
    int product = (a) * (b); \
    for (int i = 0; i < product; i++) { \
        printf("%d\n", i); \
    } \
}
```

上面示例中，替换文本放在大括号里面，这是为了创建一个块作用域，避免宏内部的变量污染外部。

带参数的宏也可以嵌套，一个宏里面包含另一个宏。

```
#define QUADP(a, b, c) ((- (b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((- (b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

上面示例是一元二次方程组求解的宏，由于存在正负两个解，所以宏 `QUAD` 先替换成另外两个宏 `QUADP` 和 `QUADM`，后者再各自替换成一个解。

那么，什么时候使用带参数的宏，什么时候使用函数呢？

一般来说，应该首先使用函数，它的功能更强、更容易理解。宏有时候会产生意想不到的替换结果，而且往往只能写成一行，除非对换行符进行转义，但是可读性就变得很差。

宏的优点是相对简单，本质上是字符串替换，不涉及数据类型，不像函数必须定义数据类型。而且，宏将每一处都替换成实际的代码，省掉了函数调用的开销，所以性能会好一些。另外，以前的代码大量使用宏，尤其是简单的数学运算，为了读懂前人的代码，需要对它有所了解。

## # 运算符，## 运算符

由于宏不涉及数据类型，所以替换以后可能为各种类型的值。如果希望替换后的值为字符串，可以在替换文本的参数前面加上 # 。

```
#define STR(x) #x

// 等同于 printf("%s\n", "3.14159");
printf("%s\n", STR(3.14159));
```

上面示例中，STR(3.14159) 会被替换成 3.14159 。如果 x 前面没有 # ，这会被解释成一个浮点数，有了 # 以后，就会被转换成字符串。

下面是另一个例子。

```
#define XNAME(n) "x"#n

// 输出 x4
printf("%s\n", XNAME(4));
```

上面示例中，#n 指定参数输出为字符串，再跟前面的字符串结合，最终输出为 "x4" 。如果不加 # ，这里实现起来就很麻烦了。

如果替换后的文本里面，参数需要跟其他标识符连在一起，组成一个新的标识符，可以使用 ## 运算符。它起到粘合作用，将参数“嵌入”一个标识符之中。

```
#define MK_ID(n) i##n
```

上面示例中，n 是宏 MK\_ID 的参数，这个参数需要跟标识符 i 粘合在一起，这时 i 和 n 之间就要使用 ## 运算符。下面是这个宏的用法示例。

```
int MK_ID(1), MK_ID(2), MK_ID(3);
// 替换成
int i1, i2, i3;
```

上面示例中，替换后的文本 i1 、 i2 、 i3 是三个标识符，参数 n 是标识符的一部分。从这个例子可以看到，## 运算符的一个主要用途是批量生成变量名和标识符。

## 不定参数的宏

宏的参数还可以是不定数量的（即不确定有多少个参数），`...` 表示剩余的参数。

```
#define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__
```

上面示例中，`X(a, b, ...)` 表示 `X()` 至少有两个参数，多余的参数使用 `...` 表示。在替换文本中，`__VA_ARGS__` 代表多余的参数（每个参数之间使用逗号分隔）。下面是用法示例。

```
X(5, 4, 3.14, "Hi!", 12)
// 替换成
(10*(5) + 20*(4)), 3.14, "Hi!", 12
```

注意，`...` 只能替代宏的尾部参数，不能写成下面这样。

```
// 报错
#define WRONG(X, ..., Y) #X #__CA_ARGS__ #Y
```

上面示例中，`...` 替代中间部分的参数，这是不允许的，会报错。

`__VA_ARGS__` 前面加上一个 `#` 号，可以让输出变成一个字符串。

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3)); // Prints "1, 2, 3"
```

## #undef

`#undef` 指令用来取消已经使用 `#define` 定义的宏。

```
#define LIMIT 400
#undef LIMIT
```

上面示例的 `undef` 指令取消已经定义的宏 `LIMIT`，后面就可以重新用 `LIMIT` 定义一个宏。

有时候想重新定义一个宏，但不确定是否以前定义过，就可以先用 `#undef` 取消，然后再定义。因为同名的宏如果两次定义不一样，会报错，而 `#undef` 的参数如果是不存在的宏，并不会报错。

GCC 的 `-U` 选项可以在命令行取消宏的定义，相当于 `#undef`。

```
$ gcc -ULIMIT foo.c
```

上面示例中的 `-U` 参数，取消了宏 `LIMIT`，相当于源文件里面的 `#undef LIMIT`。

## #include

`#include` 指令用于编译时将其他源码文件，加载进入当前文件。它有两种形式。

```
// 形式一
#include <foo.h> // 加载系统提供的文件

// 形式二
#include "foo.h" // 加载用户提供的文件
```

形式一，文件名写在尖括号里面，表示该文件是系统提供的，通常是标准库的库文件，不需要写路径。因为编译器会到系统指定的安装目录里面，去寻找这些文件。

形式二，文件名写在双引号里面，表示该文件由用户提供，具体的路径取决于编译器的设置，可能是当前目录，也可能是项目的工作目录。如果所要包含的文件在其他位置，就需要指定路径，下面是一个例子。

```
#include "/usr/local/lib/foo.h"
```

GCC 编译器的 `-I` 参数，也可以用来指定 `include` 命令中用户文件的加载路径。

```
$ gcc -Iinclude/ -o code code.c
```

上面命令中，`-Iinclude/` 指定从当前目录的 `include` 子目录里面，加载用户自己的文件。

`#include` 最常见的用途，就是用来加载包含函数原型的头文件（后缀名为 `.h`），参见《多文件编译》一章。多个 `#include` 指令的顺序无关紧要，多次包含同一个头文件也是合法的。

## #if...#endif

`#if...#endif` 指令用于预处理器的条件判断，满足条件时，内部的行会被编译，否则就被编译器忽略。

```
#if 0
    const double pi = 3.1415; // 不会执行
#endif
```

上面示例中，`#if` 后面的 `0`，表示判断条件不成立。所以，内部的变量定义语句会被编译器忽略。`#if 0` 这种写法常用来当作注释使用，不需要的代码就放在 `#if 0` 里面。

`#if` 后面的判断条件，通常是一个表达式。如果表达式的值不等于 `0`，就表示判断条件为真，编译内部的语句；如果表达式的值等于 `0`，表示判断条件为伪，则忽略内部的语句。

`#if...#end` 之间还可以加入 `#else` 指令，用于指定判断条件不成立时，需要编译的语句。

```
#define FOO 1

#if FOO
    printf("defined\n");
#else
    printf("not defined\n");
#endif
```

上面示例中，宏 `FOO` 如果定义过，会被替换成 `1`，从而输出 `defined`，否则输出 `not defined`。

如果有多个判断条件，还可以加入 `#elif` 命令。

```
#if HAPPY_FACTOR == 0
    printf("I'm not happy!\n");
#elif HAPPY_FACTOR == 1
    printf("I'm just regular\n");
#else
    printf("I'm extra happy!\n");
#endif
```

上面示例中，通过 `#elif` 指定了第二重判断。注意，`#elif` 的位置必须在 `#else` 之前。如果多个判断条件皆不满足，则执行 `#else` 的部分。

没有定义过的宏，等同于 `0`。因此如果 `UNDEFINED` 是一个没有定义过的宏，那么 `#if UNDEFINED` 为伪，而 `#if !UNDEFINED` 为真。

`#if` 的常见应用就是打开（或关闭）调试模式。

```
#define DEBUG 1

#if DEBUG
printf("value of i : %d\n", i);
printf("value of j : %d\n", j);
#endif
```

上面示例中，通过将 `DEBUG` 设为 `1`，就打开了调试模式，可以输出调试信息。

`GCC` 的 `-D` 参数可以在编译时指定宏的值，因此可以很方便地打开调试开关。

```
$ gcc -DDEBUG=1 foo.c
```

上面示例中，`-D` 参数指定宏 `DEBUG` 为 `1`，相当于在代码中指定 `#define DEBUG 1`。

## #ifdef...#endif

`#ifdef...#endif` 指令用于判断某个宏是否定义过。

有时源码文件可能会重复加载某个库，为了避免这种情况，可以在库文件里使用 `#define` 定义一个空的宏。通过这个宏，判断库文件是否被加载了。

```
#define EXTRA_HAPPY
```

上面示例中，`EXTRA_HAPPY` 就是一个空的宏。

然后，源码文件使用 `#ifdef...#endif` 检查这个宏是否定义过。

```
#ifdef EXTRA_HAPPY
printf("I'm extra happy!\n");
#endif
```

上面示例中，`#ifdef` 检查宏 `EXTRA_HAPPY` 是否定义过。如果已经存在，表示加载过库文件，就会打印一行提示。

`#ifdef` 可以与 `#else` 指令配合使用。

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#else
    printf("I'm just regular\n");
#endif
```

上面示例中，如果宏 EXTRA\_HAPPY 没有定义过，就会执行 #else 的部分。

#ifdef...#else...#endif 可以用来实现条件加载。

```
#ifdef MAVIS
    #include "foo.h"
    #define STABLES 1
#else
    #include "bar.h"
    #define STABLES 2
#endif
```

上面示例中，通过判断宏 MAVIS 是否定义过，实现加载不同的头文件。

## defined 运算符

上一节的 #ifdef 指令，等同于 #if defined 。

```
#ifdef F00
// 等同于
#if defined F00
```

上面示例中，defined 是一个预处理运算符，如果它的参数是一个定义过的宏，就会返回1，否则返回0。

使用这种语法，可以完成多重判断。

```
#if defined F00
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

这个运算符的一个应用，就是对于不同架构的系统，加载不同的头文件。

```
#if defined IBMPCL
    #include "ibmpc.h"
#elif defined MAC
    #include "mac.h"
#else
    #include "general.h"
#endif
```

上面示例中，不同架构的系统需要定义对应的宏。代码根据不同的宏，加载对应的头文件。

## #ifndef...#endif

`#ifndef...#endif` 指令跟 `#ifdef...#endif` 正好相反。它用来判断，如果某个宏没有被定义过，则执行指定的操作。

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

#ifndef EXTRA_HAPPY
    printf("I'm just regular\n");
#endif
```

上面示例中，针对宏 `EXTRA_HAPPY` 是否被定义过，`#ifdef` 和 `#ifndef` 分别指定了两种情况各自需要编译的代码。

`#ifndef` 常用于防止重复加载。举例来说，为了防止头文件 `myheader.h` 被重复加载，可以把它放在 `#ifndef...#endif` 里面加载。

```
#ifndef MYHEADER_H
    #define MYHEADER_H
    #include "myheader.h"
#endif
```

上面示例中，宏 `MYHEADER_H` 对应文件名 `myheader.h` 的大写。只要 `#ifndef` 发现这个宏没有被定义过，就说明该头文件没有加载过，从而加载内部的代码，并会定义宏 `MYHEADER_H`，防止被再次加载。



`#ifndef` 等同于 `#if !defined`。

```
#ifndef F00
// 等同于
#if !defined F00
```

## 预定义宏

C 语言提供一些预定义的宏，可以直接使用。

- `__DATE__`：编译日期，格式为“Mmm dd yyyy”的字符串（比如 Nov 23 2021）。
- `__TIME__`：编译时间，格式为“hh:mm:ss”。
- `__FILE__`：当前文件名。
- `__LINE__`：当前行号。
- `__func__`：当前正在执行的函数名。该预定义宏必须在函数作用域使用。
- `__STDC__`：如果被设为1，表示当前编译器遵循 C 标准。
- `__STDC_HOSTED__`：如果被设为1，表示当前编译器可以提供完整的标准库；否则被设为0（嵌入式系统的标准库常常是不完整的）。
- `__STDC_VERSION__`：编译所使用的 C 语言版本，是一个格式为 `yyyymmL` 的长整数，C99 版本为“199901L”，C11 版本为“201112L”，C17 版本为“201710L”。

下面示例打印这些预定义宏的值。

```
#include <stdio.h>

int main(void) {
    printf("This function: %s\n", __func__);
    printf("This file: %s\n", __FILE__);
    printf("This line: %d\n", __LINE__);
    printf("Compiled on: %s %s\n", __DATE__, __TIME__);
    printf("C Version: %ld\n", __STDC_VERSION__);
}

/* 输出如下

This function: main
This file: test.c
This line: 7
Compiled on: Mar 29 2021 19:19:37
C Version: 201710
```

```
*/
```

## #line

`#line` 指令用于覆盖预定义宏 `__LINE__`，将其改为自定义的行号。后面的行将从 `__LINE__` 的新值开始计数。

```
// 将下一行的行号重置为 300
#line 300
```

上面示例中，紧跟在 `#line 300` 后面一行的行号，将被改成300，其后的行会在300的基础上递增编号。

`#line` 还可以改掉预定义宏 `__FILE__`，将其改为自定义的文件名。

```
#line 300 "newfilename"
```

上面示例中，下一行的行号重置为 `300`，文件名重置为 `newfilename`。

## #error

`#error` 指令用于让预处理器抛出一个错误，终止编译。

```
#if __STDC_VERSION__ != 201112L
    #error Not C11
#endif
```

上面示例指定，如果编译器不使用 C11 标准，就中止编译。GCC 编译器会像下面这样报错。

```
$ gcc -std=c99 newish.c
newish.c:14:2: error: #error Not C11
```

上面示例中，GCC 使用 C99 标准编译，就报错了。

```
#if INT_MAX < 100000
    #error int type is too small
#endif
```

上面示例中，编译器一旦发现 INT 类型的最大值小于 100,000，就会停止编译。

#error 指令也可以用在 #if...#elif...#else 的部分。

```
#if defined WIN32
    // ...
#elif defined MAC_OS
    // ...
#elif defined LINUX
    // ...
#else
    #error NOT support the operating system
#endif
```

## #pragma

#pragma 指令用来修改编译器属性。

```
// 使用 C99 标准
#pragma c9x on
```

上面示例让编译器以 C99 标准进行编译。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



# 17、I/O 函数

C 语言提供了一些函数，用于与外部设备通信，称为输入输出函数，简称 I/O 函数。输入（import）指的是获取外部数据，输出（export）指的是向外部传递数据。

## 缓存和字节流

严格地说，输入输出函数并不是直接与外部设备通信，而是通过缓存（buffer）进行间接通信。这个小节介绍缓存是什么。

普通文件一般都保存在磁盘上面，跟 CPU 相比，磁盘读取或写入数据是一个很慢的操作。所以，程序直接读写磁盘是不可行的，可能每执行一行命令，都必须等半天。C 语言的解决方案，就是只要打开一个文件，就在内存里面为这个文件设置一个缓存区。

程序向文件写入数据时，程序先把数据放入缓存，等到缓存满了，再把里面的数据会一次性写入磁盘文件。这时，缓存区就空了，程序再把新的数据放入缓存，重复整个过程。

程序从文件读取数据时，文件先把一部分数据放到缓存里面，然后程序从缓存获取数据，等到缓存空了，磁盘文件再把新的数据放入缓存，重复整个过程。

内存的读写速度比磁盘快得多，缓存的设计减少了读写磁盘的次数，大大提高了程序的执行效率。另外，一次性移动大块数据，要比多次移动小块数据快得多。

这种读写模式，对于程序来说，就有点像水流（stream），不是一次性读取或写入所有数据，而是一个持续不断的过程。先操作一部分数据，等到缓存吞吐完这部分数据，再操作下一部分数据。这个过程就叫做字节流操作。

由于缓存读完就空了，所以字节流读取都是只能读一次，第二次就读不到了。这跟读取文件很不一样。

C 语言的输入输出函数，凡是涉及读写文件，都是属于字节流操作。输入函数从文件获取数据，操作的是输入流；输出函数向文件写入数据，操作的是输出流。

## printf()

printf() 是最常用的输出函数，用于屏幕输出，原型定义在头文件 `stdio.h`，详见《基本语法》一章。

## scanf()

### 基本用法

`scanf()` 函数用于读取用户的键盘输入。程序运行到这个语句时，会停下来，等待用户从键盘输入。用户输入数据、按下回车键后，`scanf()` 就会处理用户的输入，将其存入变量。它的原型定义在头文件 `stdio.h` 。

`scanf()` 的语法跟 `printf()` 类似。

```
scanf("%d", &i);
```

它的第一个参数是一个格式字符串，里面会放置占位符（与 `printf()` 的占位符基本一致），告诉编译器如何解读用户的输入，需要提取的数据是什么类型。这是因为 C 语言的数据都是有类型的，`scanf()` 必须提前知道用户输入的数据类型，才能处理数据。它的其余参数就是存放用户输入的变量，格式字符串里面有多少个占位符，就有多少个变量。

上面示例中，`scanf()` 的第一个参数 `%d`，表示用户输入的应该是一个整数。`%d` 就是一个占位符，`%` 是占位符的标志，`d` 表示整数。第二个参数 `&i` 表示，将用户从键盘输入的整数存入变量 `i`。

注意，变量前面必须加上 `&` 运算符（指针变量除外），因为 `scanf()` 传递的不是值，而是地址，即将变量 `i` 的地址指向用户输入的值。如果这里的变量是指针变量（比如字符串变量），那就不用加 `&` 运算符。

下面是一次将键盘输入读入多个变量的例子。

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

上面示例中，格式字符串 `%d%d%f%f`，表示用户输入的前两个是整数，后两个是浮点数，比如 `1 -20 3.4 -4.0e3`。这四个值依次放入 `i`、`j`、`x`、`y` 四个变量。

`scanf()` 处理数值占位符时，会自动过滤空白字符，包括空格、制表符、换行符等。所以，用户输入的数据之间，有一个或多个空格不影响 `scanf()` 解读数据。另外，用户使用回车键，将输入分成几行，也不影响解读。

```
1
-20
3.4
-4.0e3
```

上面示例中，用户分成四行输入，得到的结果与一行输入是完全一样的。每次按下回车键以后，`scanf()` 就会开始解读，如果第一行匹配第一个占位符，那么下次按下回车键时，就会从第二个占位符开始解读。

scanf() 处理用户输入的原理是，用户的输入先放入缓存，等到按下回车键后，按照占位符对缓存进行解读。解读用户输入时，会从上一次解读遗留的第一个字符开始，直到读完缓存，或者遇到第一个不符合条件的字符为止。

```
int x;
float y;

// 用户输入 "    -13.45e12# 0"
scanf("%d", &x);
scanf("%f", &y);
```

上面示例中，scanf() 读取用户输入时，%d 占位符会忽略起首的空格，从 - 处开始获取数据，读取到 -13 停下来，因为后面的 . 不属于整数的有效字符。这就是说，占位符 %d 会读到 -13 。

第二次调用 scanf() 时，就会从上一次停止解读的地方，继续往下读取。这一次读取的首字符是 .，由于对应的占位符是 %f，会读取到 -13.45e12，这是采用科学计数法的浮点数格式。后面的 # 不属于浮点数的有效字符，所以会停在这里。

由于 scanf() 可以连续处理多个占位符，所以上面的例子也可以写成下面这样。

```
scanf("%d%f", &x, &y);
```

scanf() 的返回值是一个整数，表示成功读取的变量个数。如果没有读取任何项，或者匹配失败，则返回 0。如果读取到文件结尾，则返回常量 EOF。

## 占位符

scanf() 常用的占位符如下，与 printf() 的占位符基本一致。

- %c：字符。
- %d：整数。
- %f：float 类型浮点数。
- %lf：double 类型浮点数。
- %Lf：long double 类型浮点数。
- %s：字符串。
- %[]：在方括号中指定一组匹配的字符（比如 %[0-9]），遇到不在集合之中的字符，匹配将会停止。

上面所有占位符之中，除了 %c 以外，都会自动忽略起首的空白字符。%c 不忽略空白字符，总是返回当前第一个字符，无论该字符是否为空格。如果要强制跳过字符前的空白字符，可以写成 scanf(" %c", &ch)，即 %c 前加上一个空格，表示跳过零个或多个空白字符。

下面要特别说一下占位符 `%s`，它其实不能简单地等同于字符串。它的规则是，从当前第一个非空白字符开始读起，直到遇到空白字符（即空格、换行符、制表符等）为止。因为 `%s` 不会包含空白字符，所以无法用来读取多个单词，除非多个 `%s` 一起使用。这也意味着，`scanf()` 不适合读取可能包含空格的字符串，比如书名或歌曲名。另外，`scanf()` 遇到 `%s` 占位符，会在字符串变量末尾存储一个空字符 `\0`。

`scanf()` 将字符串读入字符数组时，不会检测字符串是否超过了数组长度。所以，储存字符串时，很可能会超过数组的边界，导致预想不到的结果。为了防止这种情况，使用 `%s` 占位符时，应该指定读入字符串的最长长度，即写成 `%[m]s`，其中的 `[m]` 是一个整数，表示读取字符串的最大长度，后面的字符将被丢弃。

```
char name[11];
scanf("%10s", name);
```

上面示例中，`name` 是一个长度为11的字符数组，`scanf()` 的占位符 `%10s` 表示最多读取用户输入的10个字符，后面的字符将被丢弃，这样就不会有数组溢出的风险了。

## 赋值忽略符

有时，用户的输入可能不符合预定的格式。

```
scanf("%d-%d-%d", &year, &month, &day);
```

上面示例中，如果用户输入 `2020-01-01`，就会正确解读出年、月、日。问题是用户可能输入其他格式，比如 `2020/01/01`，这种情况下，`scanf()` 解析数据就会失败。

为了避免这种情况，`scanf()` 提供了一个赋值忽略符（assignment suppression character）`*`。只要把 `*` 加在任何占位符的百分号后面，该占位符就不会返回值，解析后将被丢弃。

```
scanf("%d%*c%d%*c%d", &year, &month, &day);
```

上面示例中，`%*c` 就是在占位符的百分号后面，加入了赋值忽略符 `*`，表示这个占位符没有对应的变量，解读后不必返回。

## sscanf()

`sscanf()` 函数与 `scanf()` 很类似，不同之处是 `sscanf()` 从字符串里面，而不是从用户输入获取数据。它的原型定义在头文件 `stdio.h` 里面。

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

sscanf() 的第一个参数是一个字符串指针，用来从其中获取数据。其他参数都与 scanf() 相同。

sscanf() 主要用来处理其他输入函数读入的字符串，从其中提取数据。

```
fgets(str, sizeof(str), stdin);
sscanf(str, "%d%d", &i, &j);
```

上面示例中，fgets() 先从标准输入获取了一行数据（fgets() 的介绍详见下一章），存入字符数组 str。然后，sscanf() 再从字符串 str 里面提取两个整数，放入变量 i 和 j。

sscanf() 的一个好处是，它的数据来源不是流数据，所以可以反复使用，不像 scanf() 的数据来源是流数据，只能读取一次。

sscanf() 的返回值是成功赋值的变量的数量，如果提取失败，返回常量 EOF。

## getchar(), putchar()

### (1) getchar()

getchar() 函数返回用户从键盘输入的一个字符，使用时不带有任何参数。程序运行到这个命令就会暂停，等待用户从键盘输入，等同于使用 scanf() 方法读取一个字符。它的原型定义在头文件 stdio.h。

```
char ch;
ch = getchar();

// 等同于
scanf("%c", &ch);
```

getchar() 不会忽略起首的空白字符，总是返回当前读取的第一个字符，无论是否为空格。如果读取失败，返回常量 EOF，由于 EOF 通常是 -1，所以返回值的类型要设为 int，而不是 char。

由于 getchar() 返回读取的字符，所以可以用在循环条件之中。

```
while (getchar() != '\n')
    ;
```



上面示例中，只有读到的字符等于换行符（`\n`），才会退出循环，常用来跳过某行。`while` 循环的循环体没有任何语句，表示对该行不执行任何操作。

下面的例子是计算某一行的字符长度。

```
int len = 0;
while(getchar() != '\n')
    len++;
```

上面示例中，`getchar()` 每读取一个字符，长度变量 `len` 就会加1，直到读取到换行符为止，这时 `len` 就是该行的字符长度。

下面的例子是跳过空格字符。

```
while ((ch = getchar()) == ' ');
```

上面示例中，结束循环后，变量 `ch` 等于第一个非空格字符。

## (2) putchar()

`putchar()` 函数将它的参数字符输出到屏幕，等同于使用 `printf()` 输出一个字符。它的原型定义在头文件 `stdio.h`。

```
putchar(ch);
// 等同于
printf("%c", ch);
```

操作成功时，`putchar()` 返回输出的字符，否则返回常量 `EOF`。

## (3) 小结

由于 `getchar()` 和 `putchar()` 这两个函数的用法，要比 `scanf()` 和 `printf()` 更简单，而且通常是用宏来实现，所以要比 `scanf()` 和 `printf()` 更快。如果操作单个字符，建议优先使用这两个函数。

## puts()

`puts()` 函数用于将参数字符串显示在屏幕（`stdout`）上，并且自动在字符串末尾添加换行符。它的原型定义在头文件 `stdio.h`。

```
puts("Here are some messages:");  
puts("Hello World");
```

上面示例中，`puts()` 在屏幕上输出两行内容。

写入成功时，`puts()` 返回一个非负整数，否则返回常量 `EOF`。

## gets()

`gets()` 函数以前用于从 `stdin` 读取整行输入，现在已经被废除了，仍然放在这里介绍一下。

该函数读取用户的一行输入，不会跳过起始处的空白字符，直到遇到换行符为止。这个函数会丢弃换行符，将其余字符放入参数变量，并在这些字符的末尾添加一个空字符 `\0`，使其成为一个字符串。

它经常与 `puts()` 配合使用。

```
char words[81];  
  
puts("Enter a string, please");  
gets(words);
```

上面示例使用 `puts()` 在屏幕上输出提示，然后使用 `gets()` 获取用户的输入。

由于 `gets()` 获取的字符串，可能超过字符数组变量的最大长度，有安全风险，建议不要使用，改为使用 `fgets()`。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 18、文件操作

本章介绍 C 语言如何操作文件。

### 文件指针

C 语言提供了一个 FILE 数据结构，记录了操作一个文件所需要的信息。该结构定义在头文件 `stdio.h`，所有文件操作函数都要通过这个数据结构，获取文件信息。

开始操作一个文件之前，就要定义一个指向该文件的 FILE 指针，相当于获取一块内存区域，用来保存文件信息。

```
FILE* fp;
```

上面示例定义了一个 FILE 指针 `fp`。

下面是一个读取文件的完整示例。

```
#include <stdio.h>

int main(void) {
    FILE* fp;
    fp = fopen("hello.txt", "r");

    char c = fgetc(fp);
    printf("%c\n", c);

    fclose(fp);
}
```

上面示例中，新建文件指针 `fp` 以后，依次使用了下面三个文件操作函数，分成三个步骤。其他的文件操作，大致上也是这样的步骤。

第一步，使用 `fopen()` 打开指定文件，返回一个 `File` 指针。如果出错，返回 `NULL`。

它相当于将指定文件的信息与新建的文件指针 `fp` 相关联，在 `FILE` 结构内部记录了这样一些信息：文件内部的当前读写位置、读写报错的记录、文件结尾指示器、缓冲区开始位置的指针、文件标识符、一个计数器（统计拷贝进缓冲区的字节数）等等。后继的操作就可以使用这个指针（而不是文件名）来处理指定文件。

同时，它还为文件建立一个缓存区。由于存在缓存区，也可以说 `fopen()` 函数“打开一个流”，后继的读写文件都是流模式。

第二步，使用读写函数，从文件读取数据，或者向文件写入数据。上例使用了 `fgetc()` 函数，从已经打开的文件里面，读取一个字符。

`fgetc()` 一调用，文件的数据块先拷贝到缓冲区。不同的计算机有不同的缓冲区大小，一般是512字节或是它的倍数，如4096或16384。随着计算机硬盘容量越来越大，缓冲区也越来越大。

`fgetc()` 从缓冲区读取数据，同时将文件指针内部的读写位置指示器，指向所读取字符的下一个字符。所有的文件读取函数都使用相同的缓冲区，后面再调用任何一个读取函数，都将从指示器指向的位置，即上一次读取函数停止的位置开始读取。

当读取函数发现已读完缓冲区里面的所有字符时，会请求把下一个缓冲区大小的数据块，从文件拷贝到缓冲区中。读取函数就以这种方式，读完文件的所有内容，直到文件结尾。不过，上例是只从缓存区读取一个字符。当函数在缓冲区里面，读完文件的最后一个字符时，就把 `FILE` 结构里面的文件结尾指示器设置为真。于是，下一次再调用读取函数时，会返回常量 `EOF`。`EOF` 是一个整数值，代表文件结尾，一般是 `-1`。

第三步，`fclose()` 关闭文件，同时清空缓存区。

上面是文件读取的过程，文件写入也是类似的方式，先把数据写入缓冲区，当缓冲区填满后，缓存区的数据将被转移到文件中。

## fopen()

fopen() 函数用来打开文件。所有文件操作的第一步，都是使用 fopen() 打开指定文件。这个函数的原型定义在头文件 `stdio.h`。

```
FILE* fopen(char* filename, char* mode);
```

它接受两个参数。第一个参数是文件名(可以包含路径)，第二个参数是模式字符串，指定对文件执行的操作，比如下面的例子中，`r` 表示以读取模式打开文件。

```
fp = fopen("in.dat", "r");
```

成功打开文件以后，fopen() 返回一个 FILE 指针，其他函数可以用这个指针操作文件。如果无法打开文件（比如文件不存在或没有权限），会返回空指针 NULL。所以，执行 fopen() 以后，最好判断一下，有没有打开成功。

```
fp = fopen("hello.txt", "r");

if (fp == NULL) {
    printf("Can't open file!\n");
    exit(EXIT_FAILURE);
}
```

上面示例中，如果 fopen() 返回一个空指针，程序就会报错。

fopen() 的模式字符串有以下几种。

- `r`：读模式，只用来读取数据。如果文件不存在，返回 NULL 指针。
- `w`：写模式，只用来写入数据。如果文件存在，文件长度会被截为0，然后再写入；如果文件不存在，则创建该文件。
- `a`：写模式，只用来在文件尾部追加数据。如果文件不存在，则创建该文件。
- `r+`：读写模式。如果文件存在，指针指向文件开始处，可以在文件头部添加数据。如果文件不存在，返回 NULL 指针。
- `w+`：读写模式。如果文件存在，文件长度会被截为0，然后再写入数据。这种模式实际上读不到数据，反而会擦掉数据。如果文件不存在，则创建该文件。
- `a+`：读写模式。如果文件存在，指针指向文件结尾，可以在现有文件末尾添加内容。如果文件不存

在，则创建该文件。

上一小节说过，`fopen()` 函数会为打开的文件创建一个缓冲区。读模式下，创建的是读缓存区；写模式下，创建的是写缓存区；读写模式下，会同时创建两个缓冲区。C 语言通过缓存区，以流的形式，向文件读写数据。

数据在文件里面，都是以二进制形式存储。但是，读取的时候，有不同的解读方法：以原本的二进制形式解读，叫做“二进制流”；将二进制数据转成文本，以文本形式解读，叫做“文本流”。写入操作也是如此，分成以二进制写入和以文本写入，后者会多一个文本转二进制的步骤。

`fopen()` 的模式字符串，默认是以文本流读写。如果添加 `b` 后缀（表示 binary），就会以“二进制流”进行读写。比如，`rb` 是读取二进制数据模式，`wb` 是写入二进制数据模式。

模式字符串还有一个 `x` 后缀，表示独占模式（exclusive）。如果文件已经存在，则打开文件失败；如果文件不存在，则新建文件，打开后不再允许其他程序或线程访问当前文件。比如，`wx` 表示以独占模式写入文件，如果文件已经存在，就会打开失败。

## 标准流

Linux 系统默认提供三个已经打开的文件，它们的文件指针如下。

- `stdin`（标准输入）：默认来源为键盘，文件指针编号为 0。
- `stdout`（标准输出）：默认目的地为显示器，文件指针编号为 1。
- `stderr`（标准错误）：默认目的地为显示器，文件指针编号为 2。

Linux 系统的文件，不一定是数据文件，也可以是设备文件，即文件代表一个可以读或写的设备。文件指针 `stdin` 默认是把键盘看作一个文件，读取这个文件，就能获取用户的键盘输入。同理，`stdout` 和 `stderr` 默认是把显示器看作一个文件，将程序的运行结果写入这个文件，用户就能看到运行结果了。它们的区别是，`stdout` 写入的是程序的正常运行结果，`stderr` 写入的是程序的报错信息。

这三个输入和输出渠道，是 Linux 默认提供的，所以分别称为标准输入（`stdin`）、标准输出（`stdout`）和标准错误（`stderr`）。因为它们的实现是一样的，都是文件流，所以合称为“标准流”。

Linux 允许改变这三个文件指针（文件流）指向的文件，这称为重定向（`redirection`）。

如果标准输入不绑定键盘，而是绑定其他文件，可以在文件名前面加上小于号 `<`，跟在程序名后面。这叫做“输入重定向”（`input redirection`）。

```
$ demo < in.dat
```

上面示例中，`demo` 程序代码里面的 `stdin`，将指向文件 `in.dat`，即从 `in.dat` 获取数据。

如果标准输出绑定其他文件，而不是显示器，可以在文件名前加上大于号 >，跟在程序名后面。这叫做“输出重定向”（output redirection）。

```
$ demo > out.dat
```

上面示例中，demo 程序代码里面的 stdout，将指向文件 out.dat，即向 out.dat 写入数据。

输出重定向 > 会先擦去 out.dat 的所有原有的内容，然后再写入。如果希望写入的信息追加在 out.dat 的结尾，可以使用 >> 符号。

```
$ demo >> out.dat
```

上面示例中，demo 程序代码里面的 stdout，将向文件 out.dat 写入数据。与 > 不同的是，写入的开始位置是 out.dat 的文件结尾。

标准错误的重定向符号是 2>。其中的 2 代表文件指针的编号，即 2> 表示将2号文件指针的写入，重定向到 err.txt。2号文件指针就是标准错误 stderr。

```
$ demo > out.dat 2> err.txt
```

上面示例中，demo 程序代码里面的 stderr，会向文件 err.txt 写入报错信息。而 stdout 向文件 out.dat 写入。

输入重定向和输出重定向，也可以结合在一条命令里面。

```
$ demo < in.dat > out.dat
```

```
// or
```

```
$ demo > out.dat < in.dat
```

重定向还有另一种情况，就是将一个程序的标准输出 stdout，指向另一个程序的标准输入 stdin，这时要使用 | 符号。

```
$ random | sum
```

上面示例中，random 程序代码里面的 stdout 的写入，会从 sum 程序代码里面的 stdin 被读取。

## fclose()

`fclose()` 用来关闭已经使用 `fopen()` 打开的文件。它的原型定义在 `stdio.h` 。

```
int fclose(FILE* stream);
```

它接受一个文件指针 `fp` 作为参数。如果成功关闭文件，`fclose()` 函数返回整数 `0`；如果操作失败（比如磁盘已满，或者出现 I/O 错误），则返回一个特殊值 `EOF`（详见下一小节）。

```
if (fclose(fp) != 0)
    printf("Something wrong.");
```

不再使用的文件，都应该使用 `fclose()` 关闭，否则无法释放资源。一般来说，系统对同时打开的文件数量有限制，及时关闭文件可以避免超过这个限制。

## EOF

C 语言文件操作函数的设计是，如果遇到文件结尾，就返回一个特殊值。程序接收到这个特殊值，就知道已经到达文件结尾了。

头文件 `stdio.h` 为这个特殊值定义了一个宏 `EOF`（end of file 的缩写），它的值一般是 `-1`。这是因为从文件读取的值，不管是二进制形式，还是 ASCII 码的形式，都不可能是负值，所以可以很安全地返回 `-1`，不会跟文件本身的数据相冲突。

需要注意的是，不像字符串结尾真的存储了 `\0` 这个值，`EOF` 并不存储在文件结尾，文件中并不存在这个值，完全是文件操作函数发现到达了文件结尾，而返回这个值。

## freopen()

`freopen()` 用于新打开一个文件，直接关联到某个已经打开的文件指针。这样可以复用文件指针。它的原型定义在头文件 `stdio.h` 。

```
FILE* freopen(char* filename, char* mode, FILE stream);
```

它跟 `fopen()` 相比，就是多出了第三个参数，表示要复用的文件指针。其他两个参数都一样，分别是文件名和打开模式。



```
freopen("output.txt", "w", stdout);
printf("hello");
```

上面示例将文件 `output.txt` 关联到 `stdout`，此后向 `stdout` 写入的内容，都会写入 `foo.txt`。由于 `printf()` 默认就是输出到 `stdout`，所以运行上面的代码以后，文件 `output.txt` 会被写入 `hello`。

`freopen()` 的返回值是它的第三个参数（文件指针）。如果打开失败（比如文件不存在），会返回空指针 `NULL`。

`freopen()` 会自动关闭原先已经打开的文件，如果文件指针并没有指向已经打开的文件，则 `freopen()` 等同于 `fopen()`。

下面是 `freopen()` 关联 `scanf()` 的例子。

```
int i, i2;

scanf("%d", &i);

freopen("someints.txt", "r", stdin);
scanf("%d", &i2);
```

上面例子中，一共调用了两次 `scanf()`，第一次调用是从键盘读取，然后使用 `freopen()` 将 `stdin` 指针关联到某个文件，第二次调用就会从该文件读取。

某些系统允许使用 `freopen()`，改变文件的打开模式。这时，`freopen()` 的第一个参数应该是 `NULL`。

```
freopen(NULL, "wb", stdout);
```

上面示例将 `stdout` 的打开模式从 `w` 改成了 `wb`。

## fgetc(), getc()

`fgetc()` 和 `getc()` 用于从文件读取一个字符。它们的用法跟 `getchar()` 类似，区别是 `getchar()` 只用来从 `stdin` 读取，而这两个函数是从任意指定的文件读取。它们的原型定义在头文件 `stdio.h`。

```
int fgetc(FILE *stream)
int getc(FILE *stream);
```

`fgetc()` 与 `getc()` 的用法是一样的，都只有文件指针一个参数。两者的区别是，`getc()` 一般用宏来实现，而 `fgetc()` 是函数实现，所以前者的性能可能更好一些。注意，虽然这两个函数返回的是一个字符，但是它们的返回值类型却不是 `char`，而是 `int`，这是因为读取失败的情况下，它们会返回 `EOF`，这个值一般是 `-1`。

```
#include <stdio.h>

int main(void) {
    FILE *fp;
    fp = fopen("hello.txt", "r");

    int c;
    while ((c = getc(fp)) != EOF)
        printf("%c", c);

    fclose(fp);
}
```

上面示例中，`getc()` 依次读取文件的每个字符，将其放入变量 `c`，直到读到文件结尾，返回 `EOF`，循环终止。变量 `c` 的类型是 `int`，而不是 `char`，因为有可能等于负值，所以设为 `int` 更好一些。

## fputc(), putc()

`fputc()` 和 `putc()` 用于向文件写入一个字符。它们的用法跟 `putchar()` 类似，区别是 `putchar()` 是向 `stdout` 写入，而这两个函数是向文件写入。它们的原型定义在头文件 `stdio.h`。

```
int fputc(int char, FILE *stream);
int putc(int char, FILE *stream);
```

`fputc()` 与 `putc()` 的用法是一样，都接受两个参数，第一个参数是待写入的字符，第二个参数是文件指针。它们的区别是，`putc()` 通常是使用宏来实现，而 `fputc()` 只作为函数来实现，所以理论上，`putc()` 的性能会好一点。

写入成功时，它们返回写入的字符；写入失败时，返回 `EOF`。

## fprintf()

`fprintf()` 用于向文件写入格式化字符串，用法与 `printf()` 类似。区别是 `printf()` 总是写入 `stdout`，而 `fprintf()` 则是写入指定的文件，它的第一个参数必须是一个文件指针。它的原型定义在头文件 `stdio.h`。

```
int fprintf(FILE* stream, const char* format, ...)
```

`fprintf()` 可以替代 `printf()`。

```
printf("Hello, world!\n");  
fprintf(stdout, "Hello, world!\n");
```

上面例子中，指定 `fprintf()` 写入 `stdout`，结果就等同于调用 `printf()`。

```
fprintf(fp, "Sum: %d\n", sum);
```

上面示例是向文件指针 `fp` 写入指定格式的字符串。

下面是向 `stderr` 输出错误信息的例子。

```
fprintf(stderr, "Something number.\n");
```

## fscanf()

`fscanf()` 用于按照给定的模式，从文件中读取内容，用法跟 `scanf()` 类似。区别是 `scanf()` 总是从 `stdin` 读取数据，而 `fscanf()` 是从文件读入数据，它的原因定义在头文件 `stdio.h`，第一个参数必须是文件指针。

```
int fscanf(FILE* stream, const char* format, ...);
```

下面是一个例子。

```
fscanf(fp, "%d%d", &i, &j);
```

上面示例中，`fscanf()` 从文件 `fp` 里面，读取两个整数，放入变量 `i` 和 `j`。

使用 `fscanf()` 的前提是知道文件的结构，它的占位符解析规则与 `scanf()` 完全一致。由于 `fscanf()` 可以连续读取，直到读到文件尾，或者发生错误（读取失败、匹配失败），才会停止读取，所以 `fscanf()` 通常放在循环里面。

```
while(fscanf(fp, "%s", words) == 1)
    puts(words);
```

上面示例中，`fscanf()` 依次读取文件的每个词，将它们一行打印一个，直到文件结束。

`fscanf()` 的返回值是赋值成功的变量数量，如果赋值失败会返回 EOF。

## fgets()

`fgets()` 用于从文件读取指定长度的字符串，它名字的第一个字符是 `f`，就代表 `file`。它的原型定义在头文件 `stdio.h`。

```
char* fgets(char* str, int STRLEN, File* fp);
```

它的第一个参数 `str` 是一个字符串指针，用于存放读取的内容。第二个参数 `STRLEN` 指定读取的长度，第三个参数是一个 `FILE` 指针，指向要读取的文件。

`fgets()` 读取 `STRLEN - 1` 个字符之后，或者遇到换行符与文件结尾，就会停止读取，然后在已经读取的内容末尾添加一个空字符 `\0`，使之成为一个字符串。注意，`fgets()` 会将换行符（`\n`）存储进字符串。

如果 `fgets` 的第三个参数是 `stdin`，就可以读取标准输入，等同于 `scanf()`。

```
fgets(str, sizeof(str), stdin);
```

读取成功时，`fgets()` 的返回值是它的第一个参数，即指向字符串的指针，否则返回空指针 `NULL`。

`fgets()` 可以用来读取文件的每一行，下面是读取文件所有行的例子。

```
#include <stdio.h>

int main(void) {
    FILE* fp;
    char s[1024]; // 数组必须足够大，足以放下一行
    int linecount = 0;
```

```

fp = fopen("hello.txt", "r");

while (fgets(s, sizeof s, fp) != NULL)
    printf("%d: %s", ++linecount, s);

fclose(fp);
}

```

上面示例中，每读取一行，都会输出行号和该行的内容。

下面的例子是循环读取用户的输入。

```

char words[10];

puts("Enter strings (q to quit):");

while (fgets(words, 10, stdin) != NULL) {
    if (words[0] == 'q' && words[1] == '\n')
        break;

    puts(words);
}

puts("Done.");

```

上面的示例中，如果用户输入的字符串大于9个字符，`fgets()` 会多次读取。直到遇到 `q` + 回车键，才会退出循环。

## fputs()

`fputs()` 函数用于向文件写入字符串，和 `puts()` 函数只有一点不同，那就是它不会在字符串末尾添加换行符。这是因为 `fgets()` 保留了换行符，所以 `fputs()` 就不添加了。`fputs()` 函数通常与 `fgets()` 配对使用。

它的原型定义在 `stdio.h` 。

```

int fputs(const char* str, FILE* stream);

```

它接受两个参数，第一个参数是字符串指针，第二个参数是要写入的文件指针。如果第二个参数为 `stdout`（标准输出），就是将内容输出到计算机屏幕，等同于 `printf()`。

```
char words[14];

puts("Enter a string, please.");
fgets(words, 14, stdin);

puts("This is your string:");
fputs(words, stdout);
```

上面示例中，先用 `fgets()` 从 `stdin` 读取用户输入，然后用 `fputs()` 输出到 `stdout`。

写入成功时，`fputs()` 返回一个非负整数，否则返回 `EOF`。

## fwrite()

`fwrite()` 用来一次性写入较大的数据块，主要用途是将数组数据一次性写入文件，适合写入二进制数据。它的原型定义在 `stdio.h`。

```
size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* fp);
```

它接受四个参数。

- `ptr`：数组指针。
- `size`：每个数组成员的大小，单位字节。
- `nmemb`：数组成员的数量。
- `fp`：要写入的文件指针。

注意，`fwrite()` 原型的第一个参数类型是 `void*`，这是一个无类型指针，编译器会自动将参数指针转成 `void*` 类型。正是由于 `fwrite()` 不知道数组成员的类型，所以才需要知道每个成员的大小（第二个参数）和成员数量（第三个参数）。

`fwrite()` 函数的返回值是成功写入的数组成员的数量（注意不是字节数）。正常情况下，该返回值就是第三个参数 `nmemb`，但如果出现写入错误，只写入了一部分成员，返回值会比 `nmemb` 小。

要将整个数组 `arr` 写入文件，可以采用下面的写法。

```
fwrite(  
    arr,  
    sizeof(arr[0]),  
    sizeof(arr) / sizeof(arr[0]),  
    fp  
);
```

上面示例中，`sizeof(a[0])` 是每个数组成员占用的字节，`sizeof(a) / sizeof(a[0])` 是整个数组的成员数量。

下面的例子是将一个大小为256字节的字符串写入文件。

```
char buffer[256];  
  
fwrite(buffer, 1, 256, fp);
```

上面示例中，数组 `buffer` 每个成员是1个字节，一共有256个成员。由于 `fwrite()` 是连续内存复制，所以写成 `fwrite(buffer, 256, 1, fp)` 也能达到目的。

`fwrite()` 没有规定一定要写入整个数组，只写入数组的一部分也是可以的。

任何类型的数据都可以看成是1字节数据组成的数组，或者是一个成员的数组，所以 `fwrite()` 实际上可以写入任何类型的数据，而不仅仅是数组。比如，`fwrite()` 可以将一个 Struct 结构写入文件保存。

```
fwrite(&s, sizeof(s), 1, fp);
```

上面示例中，`s` 是一个 Struct 结构指针，可以看成是一个成员的数组。注意，如果 `s` 的属性包含指针，存储时需要小心，因为保存指针可能没意义，还原出来的时候，并不能保证指针指向的数据还存在。

`fwrite()` 以及后面要介绍的 `fread()`，比较适合读写二进制数据，因为它们不会对写入的数据进行解读。二进制数据可能包含空字符 `\0`，这是 C 语言的字符串结尾标记，所以读写二进制文件，不适合使用文本读写函数（比如 `fprintf()` 等）。

下面是一个写入二进制文件的例子。

```
#include <stdio.h>

int main(void) {
    FILE* fp;
    unsigned char bytes[] = {5, 37, 0, 88, 255, 12};

    fp = fopen("output.bin", "wb");
    fwrite(bytes, sizeof(char), sizeof(bytes), fp);
    fclose(fp);
    return 0;
}
```

上面示例中，写入二进制文件时，`fopen()` 要使用 `wb` 模式打开，表示二进制写入。`fwrite()` 可以把数据解释成单字节数组，因此它的第二个参数是 `sizeof(char)`，第三个参数是数组的总字节数 `sizeof(bytes)`。

上面例子写入的文件 `output.bin`，使用十六进制编辑器打开，会是下面的内容。

```
05 25 00 58 ff 0c
```

`fwrite()` 还可以连续向一个文件写入数据。

```
struct clientData myClient = {1, 'foo bar'};

for (int i = 1; i <= 100; i++) {
    fwrite(&myClient, sizeof(struct clientData), 1, cfPtr);
}
```

上面示例中，`fwrite()` 连续将100条数据写入文件。

## fread()

`fread()` 函数用于一次性从文件读取较大的数据块，主要用途是将文件内容读入一个数组，适合读取二进制数据。它的原型定义在头文件 `stdio.h`。

```
size_t fread(void* ptr, size_t size, size_t nmem, FILE* fp);
```

它接受四个参数，与 `fwrite()` 完全相同。



- ptr：数组地址。
- size：数组的成员数量。
- nmemb：每个数组成员的大小。
- fp：文件指针。

要将文件内容读入数组 arr，可以采用下面的写法。

```
fread(
    arr,
    sizeof(arr[0]),
    sizeof(arr) / sizeof(arr[0]),
    fp
);
```

上面示例中，数组长度（第二个参数）和每个成员的大小（第三个参数）的乘积，就是数组占用的内存空间的大小。fread() 会从文件（第四个参数）里面读取相同大小的内容，然后将 ptr（第一个参数）指向这些内容的内存地址。

下面的例子是将文件内容读入一个10个成员的双精度浮点数数组。

```
double earnings[10];
fread(earnings, sizeof(double), 10, fp);
```

上面示例中，每个数组成员的大小是 sizeof(double)，一个有10个成员，就会从文件 fp 读取 sizeof(double) \* 10 大小的内容。

fread() 函数的返回值是成功读取的数组成员的数量。正常情况下，该返回值就是第三个参数 nmemb，但如果出现读取错误或读到文件结尾，该返回值就会比 nmemb 小。所以，检查 fread() 的返回值是非常重要的。

fread() 和 fwrite() 可以配合使用。在程序终止之前，使用 fwrite() 将数据保存进文件，下次运行时再用 fread() 将数据还原进入内存。

下面是读取上一节生成的二进制文件 output.bin 的例子。

```
#include <stdio.h>

int main(void) {
    FILE* fp;
    unsigned char c;

    fp = fopen("output.bin", "rb");
    while (fread(&c, sizeof(char), 1, fp) > 0)
        printf("%d\n", c);
    return 0;
}
```

运行后，得到如下结果。

```
5
37
0
88
255
12
```

## feof()

`feof()` 函数判断文件的内部指针是否指向文件结尾。它的原型定义在头文件 `stdio.h` 。

```
int feof(FILE *fp);
```

`feof()` 接受一个文件指针作为参数。如果已经到达文件结尾，会返回一个非零值（表示 `true`），否则返回 `0`（表示 `false`）。

诸如 `fgetc()` 这样的文件读取函数，如果返回 `EOF`，有两种可能，一种可能是已读取到文件结尾，另一种可能是出现读取错误。`feof()` 可以用来判断到底是那一种情况。

下面是通过 `feof()` 判断是否到达文件结尾，从而循环读取整个文件的例子。

```

int num;
char name[50];

FILE* cfPtr = fopen("clients.txt", "r");

while (!feof(cfPtr)) {
    fscanf(cfPtr, "%d%s\n", &num, name);
    printf("%d %s\n", num, name);
}

fclose(cfPtr);

```

上面示例通过循环判断 `feof()` 是否读到文件结尾，从而实现读出整个文件内容。

`feof()` 为真时，可以通过 `fseek()`、`rewind()`、`fsetpos()` 函数改变文件内部读写位置的指示器，从而清除这个函数的状态。

## fseek()

每个文件指针都有一个内部指示器（内部指针），记录当前打开的文件的读写位置（file position），即下一次读写从哪里开始。文件操作函数（比如 `getc()`、`fgets()`、`fscanf()` 和 `fread()` 等）都从这个指示器指定的位置开始按顺序读写文件。

如果希望改变这个指示器，将它移到文件的指定位置，可以使用 `fseek()` 函数。它的原型定义在头文件 `stdio.h`。

```

int fseek(FILE* stream, long int offset, int whence);

```

`fseek()` 接受3个参数。

- `stream`：文件指针。
- `offset`：距离基准（第三个参数）的字节数。类型为 `long int`，可以为正值（向文件末尾移动）、负值（向文件开始处移动）或 0（保持不动）。
- `whence`：位置基准，用来确定计算起点。它的值是以下三个宏（定义在 `stdio.h`）：`SEEK_SET`（文件开始处）、`SEEK_CUR`（内部指针的当前位置）、`SEEK_END`（文件末尾）

请看下面的例子。

```

// 定位到文件开始处

```

```
fseek(fp, 0L, SEEK_SET);

// 定位到文件末尾
fseek(fp, 0L, SEEK_END);

// 从当前位置前移2个字节
fseek(fp, 2L, SEEK_CUR);

// 定位到文件第10个字节
fseek(fp, 10L, SEEK_SET);

// 定位到文件倒数第10个字节
fseek(fp, -10L, SEEK_END);
```

上面示例中，`fseek()` 的第二个参数为 `long` 类型，所以移动距离必须加上后缀 `L`，将其转为 `long` 类型。

下面的示例逆向输出文件的所有字节。

```
for (count = 1L; count <= size; count++) {
    fseek(fp, -count, SEEK_END);
    ch = getc(fp);
}
```

注意，`fseek()` 最好只用来操作二进制文件，不要用来读取文本文件。因为文本文件的字符有不同的编码，某个位置的准确字节位置不容易确定。

正常情况下，`fseek()` 的返回值为0。如果发生错误（如移动的距离超出文件的范围），返回值为非零值（比如 -1）。

## ftell()

`ftell()` 函数返回文件内部指示器的当前位置。它的原型定义在头文件 `stdio.h`。

```
long int ftell(FILE* stream);
```

它接受一个文件指针作为参数。返回值是一个 `long` 类型的整数，表示内部指示器的当前位置，即文件开始处到当前位置的字节数，`0` 表示文件开始处。如果发生错误，`ftell()` 返回 `-1L`。

`ftell()` 可以跟 `fseek()` 配合使用，先记录内部指针的位置，一系列操作过后，再用 `fseek()` 返回原来的位置。

```
long file_pos = ftell(fp);

// 一系列文件操作之后
fseek(fp, file_pos, SEEK_SET);
```

下面的例子先将指示器定位到文件结尾，然后得到文件开始处到结尾的字节数。

```
fseek(fp, 0L, SEEK_END);
size = ftell(fp);
```

## rewind()

`rewind()` 函数可以让文件的内部指示器回到文件开始处。它的原型定义在 `stdio.h` 。

```
void rewind(file* stream);
```

它接受一个文件指针作为参数。

`rewind(fp)` 基本等价于 `fseek(fp, 0L, seek_set)`，唯一的区别是 `rewind()` 没有返回值，而且会清除当前文件的错误指示器。

## fgetpos(), fsetpos()

`fseek()` 和 `ftell()` 有一个潜在的问题，那就是它们都把文件大小限制在 `long int` 类型能表示的范围内。这看起来相当大，但是在32位计算机上，`long int` 的长度为4个字节，能够表示的范围最大为 4GB。随着存储设备的容量迅猛增长，文件也越来越大，往往会超出这个范围。鉴于此，C 语言新增了两个处理大文件的新定位函数：`fgetpos()` 和 `fsetpos()`。

它们的原型都定义在头文件 `stdio.h` 。

```
int fgetpos(FILE* stream, fpos_t* pos);
int fsetpos(FILE* stream, const fpos_t* pos);
```

`fgetpos()` 函数会将文件内部指示器的当前位置，存储在指针变量 `pos`。该函数接受两个参数，第一个是文件指针，第二个存储指示器位置的变量。

`fsetpos()` 函数会将文件内部指示器的位置，移动到指针变量 `pos` 指定的地址。注意，变量 `pos` 必须是通过调用 `fgetpos()` 方法获得的。`fsetpos()` 的两个参数与 `fgetpos()` 必须是一样的。

记录文件内部指示器位置的指针变量 `pos`，类型为 `fpos_t*`（file position type 的缩写，意为文件定位类型）。它不一定是整数，也可能是一个 Struct 结构。

下面是用法示例。

```
fpos_t file_pos;
fgetpos(fp, &file_pos);

// 一系列文件操作之后
fsetpos(fp, &file_pos);
```

上面示例中，先用 `fgetpos()` 获取内部指针的位置，后面再用 `fsetpos()` 恢复指针的位置。

执行成功时，`fgetpos()` 和 `fsetpos()` 都会返回 `0`，否则返回非零值。

## `ferror()`，`clearerr()`

所有的文件操作函数如果执行失败，都会在文件指针里面记录错误状态。后面的操作只要读取错误指示器，就知道前面的操作出错了。

`ferror()` 函数用来返回错误指示器的状态。可以通过这个函数，判断前面的文件操作是否成功。它的原型定义在头文件 `stdio.h`。

```
int ferror(FILE *stream);
```

它接受一个文件指针作为参数。如果前面的操作出现错误，`ferror()` 就会返回一个非零整数（表示 true），否则返回 `0`。

`clearerr()` 函数用来重置出错指示器。它的原型定义在头文件 `stdio.h`。

```
void clearerr(FILE* fp);
```

它接受一个文件指针作为参数，没有返回值。

下面是一个例子。

```
FILE* fp = fopen("file.txt", "w");
char c = fgetc(fp);

if (ferror(fp)) {
    printf("读取文件: file.txt 时发生错误\n");
}

clearerr(fp);
```

上面示例中，`fgetc()` 尝试读取一个以“写模式”打开的文件，读取失败就会返回 EOF。这时调用 `ferror()` 就可以知道上一步操作出错了。处理完以后，再用 `clearerr()` 清除出错状态。

文件操作函数如果正常执行，`ferror()` 和 `feof()` 都会返回零。如果执行不正常，就要判断到底是哪里出了问题。

```
if (fscanf(fp, "%d", &n) != 1) {
    if (ferror(fp)) {
        printf("io error\n");
    }
    if (feof(fp)) {
        printf("end of file\n");
    }

    clearerr(fp);

    fclose(fp);
}
```

上面示例中，当 `fscanf()` 函数报错时，通过检查 `ferror()` 和 `feof()`，确定到底发生什么问题。这两个指示器改变状态后，会保持不变，所以要用 `clearerr()` 清除它们，`clearerr()` 可以同时清除两个指示器。

## remove()

`remove()` 函数用于删除指定文件。它的原型定义在头文件 `stdio.h` 。

```
int remove(const char* filename);
```

它接受文件名作为参数。如果删除成功，`remove()` 返回 0，否则返回非零值。

```
remove("foo.txt");
```

上面示例删除了 `foo.txt` 文件。

注意，删除文件必须是在文件关闭的状态下。如果是用 `fopen()` 打开的文件，必须先用 `fclose()` 关闭后再删除。

## rename()

`rename()` 函数用于文件改名，也用于移动文件。它的原型定义在头文件 `stdio.h` 。

```
int rename(const char* old_filename, const char* new_filename);
```

它接受两个参数，第一个参数是现在的文件名，第二个参数是新的文件名。如果改名成功，`rename()` 返回 `0`，否则返回非零值。

```
rename("foo.txt", "bar.txt");
```

上面示例将 `foo.txt` 改名为 `bar.txt` 。

注意，改名后的文件不能与现有文件同名。另外，如果要改名的文件已经打开了，必须先关闭，然后再改名，对打开的文件进行改名会失败。

下面是移动文件的例子。

```
rename("/tmp/evidence.txt", "/home/beej/nothing.txt");
```

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！





## 19、变量说明符

C 语言允许声明变量的时候，加上一些特定的说明符（specifier），为编译器提供变量行为的额外信息。它的主要作用是帮助编译器优化代码，有时会对程序行为产生影响。

### const

const 说明符表示变量是只读的，不得被修改。

```
const double PI = 3.14159;  
PI = 3; // 报错
```

上面示例里面的 const，表示变量 PI 的值不应改变。如果改变的话，编译器会报错。

对于数组，const 表示数组成员不能修改。

```
const int arr[] = {1, 2, 3, 4};  
arr[0] = 5;
```

上面示例中，const 使得数组 arr 的成员无法修改。

对于指针变量，const 有两种写法，含义是不一样的。如果 const 在 \* 前面，表示指针指向的值不可修改。

```
// const 表示指向的值 *x 不能修改
int const * x
# 或者
const int * x
```

下面示例中，对 x 指向的值进行修改导致报错。

```
int p = 1
const int* x = &p;

(*x)++; // 报错
```

如果 const 在 \* 后面，表示指针包含的地址不可修改。

```
// const 表示地址 x 不能修改
int* const x
```

下面示例中，对 x 进行修改导致报错。

```
int p = 1
int* const x = &p;

x++; // 报错
```

这两者可以结合起来。

```
const char* const x;
```

上面示例中，指针变量 x 指向一个字符串。两个 const 意味着，x 包含的内存地址以及 x 指向的字符串，都不能修改。

const 的一个用途，就是防止函数体内修改函数参数。如果某个参数在函数体内不会被修改，可以在函数声明时，对该参数添加 const 说明符。这样的话，使用这个函数的人看到原型里面的 const，就知道调用函数前后，参数数组保持不变。

```
void find(const int* arr, int n);
```

上面示例中，函数 `find` 的参数数组 `arr` 有 `const` 说明符，就说明该数组在函数内部将保持不变。

有一种情况需要注意，如果一个指针变量指向 `const` 变量，那么该指针变量也不应该被修改。

```
const int i = 1;
int* j = &i;
*j = 2; // 报错
```

上面示例中，`j` 是一个指针变量，指向变量 `i`，即 `j` 和 `i` 指向同一个地址。`j` 本身没有 `const` 说明符，但是 `i` 有。这种情况下，`j` 指向的值也不能被修改。

## static

`static` 说明符对于全局变量和局部变量有不同的含义。

(1) 用于局部变量（位于块作用域内部）。

`static` 用于函数内部声明的局部变量时，表示该变量的值会在函数每次执行后得到保留，下次执行时不会进行初始化，就类似于一个只用于函数内部的全局变量。由于不必每次执行函数时，都对该变量进行初始化，这样可以提高函数的执行速度，详见《函数》一章。

(2) 用于全局变量（位于块作用域外部）。

`static` 用于函数外部声明的全局变量时，表示该变量只用于当前文件，其他源码文件不可以引用该变量，即该变量不会被链接（link）。

`static` 修饰的变量，初始化时，值不能等于变量，必须是常量。

```
int n = 10;
static m = n; // 报错
```

上面示例中，变量 `m` 有 `static` 修饰，它的值如果等于变量 `n`，就会报错，必须等于常量。

只在当前文件里面使用的函数，也可以声明为 `static`，表明该函数只在当前文件使用，其他文件可以定义同名函数。

```
static int g(int i);
```

## auto

`auto` 说明符表示该变量的存储，由编译器自主分配内存空间，且只存在于定义时所在的作用域，退出作用域时会自动释放。

由于只要不是 `extern` 的变量（外部变量），都是由编译器自主分配内存空间的，这属于默认行为，所以该说明符没有实际作用，一般都省略不写。

```
auto int a;  
// 等同于  
int a;
```

## extern

`extern` 说明符表示，该变量在其他文件里面声明，没有必要在当前文件里面为它分配空间。通常用来表示，该变量是多个文件共享的。

```
extern int a;
```

上面代码中，`a` 是 `extern` 变量，表示该变量在其他文件里面定义和初始化，当前文件不必为它分配存储空间。

但是，变量声明时，同时进行初始化，`extern` 就会无效。

```
// extern 无效  
extern int i = 0;  
  
// 等同于  
int i = 0;
```

上面代码中，`extern` 对变量初始化的声明是无效的。这是为了防止多个 `extern` 对同一个变量进行多次初始化。

函数内部使用 `extern` 声明变量，就相当于该变量是静态存储，每次执行时都要从外部获取它的值。

函数本身默认是 `extern`，即该函数可以被外部文件共享，通常省略 `extern` 不写。如果只希望函数在当前文件可用，那就需要在函数前面加上 `static`。

```
extern int f(int i);  
// 等同于  
int f(int i);
```

## register

`register` 说明符向编译器表示，该变量是经常使用的，应该提供最快的读取速度，所以应该放进寄存器。但是，编译器可以忽略这个说明符，不一定按照这个指示行事。

```
register int a;
```

上面示例中，`register` 提示编译器，变量 `a` 会经常用到，要为其提供最快的读取速度。

`register` 只对声明在代码块内部的变量有效。

设为 `register` 的变量，不能获取它的地址。

```
register int a;  
int *p = &a; // 编译器报错
```

上面示例中，`&a` 会报错，因为变量 `a` 可能放在寄存器里面，无法获取内存地址。

如果数组设为 `register`，也不能获取整个数组或任一数组成员的地址。

```
register int a[] = {11, 22, 33, 44, 55};  
  
int p = a; // 报错  
int a = *(a + 2); // 报错
```

历史上，CPU 内部的缓存，称为寄存器（register）。与内存相比，寄存器的访问速度快得多，所以使用它们可以提高速度。但是它们不在内存之中，所以没有内存地址，这就是为什么不能获取指向它们的指针地址。现代编译器已经有巨大的进步，不管是否使用 `register` 关键字，都会尽可能使用寄存器，所以不保证一定会把这些变量放到寄存器。

## volatile

`volatile` 说明符表示所声明的变量，可能会预想不到地发生变化（即其他程序可能会更改它的值），不受当前程序控制，因此编译器不要对这类变量进行优化，每次使用时都应该查询一下它的值。硬件设备的编程中，这个说明符很常用。

```
volatile int foo;
volatile int* bar;
```

`volatile` 的目的是阻止编译器对变量行为进行优化，请看下面的例子。

```
int foo = x;
// 其他语句，假设没有改变 x 的值
int bar = x;
```

上面代码中，由于变量 `foo` 和 `bar` 都等于 `x`，而且 `x` 的值也没有发生变化，所以编译器可能会把 `x` 放入缓存，直接从缓存读取值（而不是从 `x` 的原始内存位置读取），然后对 `foo` 和 `bar` 进行赋值。如果 `x` 被设定为 `volatile`，编译器就不会把它放入缓存，每次都从原始位置去取 `x` 的值，因为在两次读取之间，其他程序可能会改变 `x`。

## restrict

`restrict` 说明符允许编译器优化某些代码。它只能用于指针，表明该指针是访问数据的唯一方式。

```
int* restrict pt = (int*) malloc(10 * sizeof(int));
```

上面示例中，`restrict` 表示变量 `pt` 是访问 `malloc` 所分配内存的唯一方式。

下面例子的变量 `foo`，就不能使用 `restrict` 修饰符。

```
int foo[10];
int* bar = foo;
```

上面示例中，变量 `foo` 指向的内存，可以用 `foo` 访问，也可以用 `bar` 访问，因此就不能将 `foo` 设为 `restrict`。

如果编译器知道某块内存只能用一个方式访问，可能可以更好地优化代码，因为不用担心其他地方会修改值。

`restrict` 用于函数参数时，表示参数的内存地址之间没有重叠。

```
void swap(int* restrict a, int* restrict b) {  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```

上面示例中，函数参数声明里的 `restrict` 表示，参数 `a` 和参数 `b` 的内存地址没有重叠。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 20、多文件项目

### 简介

一个软件项目往往包含多个源码文件，编译时需要将这些文件一起编译，生成一个可执行文件。

假定一个项目有两个源码文件 `foo.c` 和 `bar.c`，其中 `foo.c` 是主文件，`bar.c` 是库文件。所谓“主文件”，就是包含了 `main()` 函数的项目入口文件，里面会引用库文件定义的各种函数。

```
// File foo.c
#include <stdio.h>

int main(void) {
    printf("%d\n", add(2, 3)); // 5!
}
```

上面代码中，主文件 `foo.c` 调用了函数 `add()`，这个函数是在库文件 `bar.c` 里面定义的。

```
// File bar.c

int add(int x, int y) {
    return x + y;
}
```

现在，将这两个文件一起编译。

```
$ gcc -o foo foo.c bar.c

# 更省事的写法
$ gcc -o foo *.c
```

上面命令中，`gcc` 的 `-o` 参数指定生成的二进制可执行文件的文件名，本例是 `foo`。

这个命令运行后，编译器会发出警告，原因是在编译 `foo.c` 的过程中，编译器发现一个不认识的函数 `add()`，`foo.c` 里面没有这个函数的原型或者定义。因此，最好修改一下 `foo.c`，在文件头部加入 `add()` 的原型。

```
// File foo.c
#include <stdio.h>

int add(int, int);

int main(void) {
    printf("%d\n", add(2, 3)); // 5!
}
```

现在再编译就没有警告了。



你可能马上就会想到，如果有多个文件都使用这个函数 `add()`，那么每个文件都需要加入函数原型。一旦需要修改函数 `add()`（比如改变参数的数量），就会非常麻烦，需要每个文件逐一改动。所以，通常的做法是新建一个专门的头文件 `bar.h`，放置所有在 `bar.c` 里面定义的函数的原型。

```
// File bar.h

int add(int, int);
```

然后使用 `include` 命令，在用到这个函数的源码文件里面加载这个头文件 `bar.h`。

```
// File foo.c

#include <stdio.h>
#include "bar.h"

int main(void) {
    printf("%d\n", add(2, 3)); // 5!
}
```

上面代码中，`#include "bar.h"` 表示加入头文件 `bar.h`。这个文件没有放在尖括号里面，表示它是用户提供的；它没有写路径，就表示与当前源码文件在同一个目录。

然后，最好在 `bar.c` 里面也加载这个头文件，这样可以让编译器验证，函数原型与函数定义是否一致。

```
// File bar.h
#include "bar.h"

int add(int, int);
```

现在重新编译，就可以顺利得到二进制可执行文件。

```
$ gcc -o foo foo.c bar.c
```

## 重复加载

头文件里面还可以加载其他头文件，因此有可能产生重复加载。比如，`a.h` 和 `b.h` 都加载了 `c.h`，然后 `foo.c` 同时加载了 `a.h` 和 `b.h`，这意味着 `foo.c` 会编译两次 `c.h`。

最好避免这种重复加载，虽然多次定义同一个函数原型并不会报错，但是有些语句重复使用会报错，比如多次重复定义同一个 Struct 数据结构。解决重复加载的常见方法是，在头文件里面设置一个专门的宏，加载时一旦发现这个宏存在，就不再继续加载当前文件了。

```
// File bar.h
#ifndef BAR_H
#define BAR_H
int add(int, int);
#endif
```

上面示例中，头文件 bar.h 使用 `#ifndef` 和 `#endif` 设置了一个条件判断。每当加载这个头文件时，就会执行这个判断，查看有没有设置过宏 `BAR_H`。如果设置过了，表明这个头文件已经加载过了，就不再重复加载了，反之就先设置一下这个宏，然后加载函数原型。

## extern 说明符

当前文件还可以使用其他文件定义的变量，这时要使用 `extern` 说明符，在当前文件中声明，这个变量是其他文件定义的。

```
extern int myVar;
```

上面示例中，`extern` 说明符告诉编译器，变量 `myvar` 是其他脚本文件声明的，不需要在这里为它分配内存空间。

由于不需要分配内存空间，所以 `extern` 声明数组时，不需要给出数组长度。

```
extern int a[];
```

这种共享变量的声明，可以直接写在源码文件里面，也可以放在头文件中，通过 `#include` 指令加载。

## static 说明符

正常情况下，当前文件内部的全局变量，可以被其他文件使用。有时候，不希望发生这种情况，而是希望某个变量只局限在当前文件内部使用，不要被其他文件引用。

这时可以在声明变量的时候，使用 `static` 关键字，使得该变量变成当前文件的私有变量。

```
static int foo = 3;
```

上面示例中，变量 `foo` 只能在当前文件里面使用，其他文件不能引用。

## 编译策略

多个源码文件的项目，编译时需要所有文件一起编译。哪怕只是修改了一行，也需要从头编译，非常耗费时间。

为了节省时间，通常的做法是将编译拆分成两个步骤。第一步，使用 GCC 的 `-c` 参数，将每个源码文件单独编译为对象文件（object file）。第二步，将所有对象文件链接在一起，合并生成一个二进制可执行文件。

```
$ gcc -c foo.c # 生成 foo.o
$ gcc -c bar.c # 生成 bar.o

# 更省事的写法
$ gcc -c *.c
```

上面命令为源码文件 `foo.c` 和 `bar.c`，分别生成对象文件 `foo.o` 和 `bar.o`。

对象文件不是可执行文件，只是编译过程中的一个阶段性产物，文件名与源码文件相同，但是后缀名变成了 `.o`。

得到所有的对象文件以后，再次使用 `gcc` 命令，将它们通过链接，合并生成一个可执行文件。

```
$ gcc -o foo foo.o bar.o

# 更省事的写法
$ gcc -o foo *.o
```

以后，修改了哪一个源文件，就将这个文件重新编译成对象文件，其他文件不用重新编译，可以继续使用原来的对象文件，最后再将所有对象文件重新链接一次就可以了。由于链接的耗时大大短于编译，这样做就节省了大量时间。

## make 命令

大型项目的编译，如果全部手动完成，是非常麻烦的，容易出错。一般会使用专门的自动化编译工具，比如 `make`。

make 是一个命令行工具，使用时会自动在当前目录下搜索配置文件 makefile（也可以写成 Makefile）。该文件定义了所有的编译规则，每个编译规则对应一个编译产物。为了得到这个编译产物，它需要知道两件事。

- 依赖项（生成该编译产物，需要用到哪些文件）
- 生成命令（生成该编译产物的命令）

比如，对象文件 `foo.o` 是一个编译产物，它的依赖项是 `foo.c`，生成命令是 `gcc -c foo.c`。对应的编译规则如下：

```
foo.o: foo.c
    gcc -c foo.c
```

上面示例中，编译规则由两行组成。第一行首先是编译产物，冒号后面是它的依赖项，第二行则是生成命令。

注意，第二行的缩进必须使用 Tab 键，如果使用空格键会报错。

完整的配置文件 makefile 由多个编译规则组成，可能是下面的样子。

```
foo: foo.o bar.o
    gcc -o foo foo.o bar.o

foo.o: bar.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```

上面是 makefile 的一个示例文件。它包含三个编译规则，对应三个编译产物（`foo.o`、`bar.o` 和 `foo`），每个编译规则之间使用空行分隔。

有了 makefile，编译时，只要在 make 命令后面指定编译目标（编译产物的名字），就会自动调用对应的编译规则。

```
$ make foo.o
```

```
# or
```

```
$ make bar.o
```

```
# or
```

```
$ make foo
```

上面示例中，make 命令会根据不同的命令，生成不同的编译产物。

如果省略了编译目标，make 命令会执行第一条编译规则，构建相应的产物。

```
$ make
```

上面示例中，make 后面没有编译目标，所以会执行 makefile 的第一条编译规则，本例是 make foo 。由于用户期望执行 make 后得到最终的可执行文件，所以建议总是把最终可执行文件的编译规则，放在 makefile 文件的第一条。makefile 本身对编译规则没有顺序要求。

make 命令的强大之处在于，它不是每次执行命令，都会进行编译，而是会检查是否有必要重新编译。具体方法是，通过检查每个源码文件的时间戳，确定在上次编译之后，哪些文件发生过变动。然后，重新编译那些受到影响的编译产物（即编译产物直接或间接依赖于那些发生变动的源码文件），不受影响的编译产物，就不会重新编译。

举例来说，上次编译之后，修改了 foo.c ，没有修改 bar.c 和 bar.h 。于是，重新运行 make foo 命令时，Make 就会发现 bar.c 和 bar.h 没有变动过，因此不用重新编译 bar.o ，只需要重新编译 foo.o 。有了新的 foo.o 以后，再跟 bar.o 一起，重新编译成新的可执行文件 foo 。

Make 这样设计的最大好处，就是自动处理编译过程，只重新编译变动过的文件，因此大大节省了时间。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 21、命令行环境

### 命令行参数

C 语言程序可以从命令行接收参数。

```
$ ./foo hello world
```

上面示例中，程序 `foo` 接收了两个命令行参数 `hello` 和 `world`。

程序内部怎么拿到命令行参数呢？C 语言会把命令行输入的内容，放在一个数组里面。`main()` 函数的参数可以接收到这个数组。

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }
}
```

上面示例中，`main()` 函数有两个参数 `argc`（argument count）和 `argv`（argument variable）。这两个参数的名字可以任意取，但是一般来说，约定俗成就是使用这两个词。

第一个参数 `argc` 是命令行参数的数量，由于程序名也被计算在内，所以严格地说 `argc` 是参数数量 + 1。

第二个参数 `argv` 是一个数组，保存了所有的命令行输入，它的每个成员是一个字符串指针。

以 `./foo hello world` 为例，`argc` 是3，表示命令行输入有三个组成部分：`./foo`、`hello`、`world`。数组 `argv` 用来获取这些输入，`argv[0]` 是程序名 `./foo`，`argv[1]` 是 `hello`，`argv[2]` 是 `world`。一般来说，`argv[1]` 到 `argv[argc - 1]` 依次是命令行的所有参数。`argv[argc]` 则是一个空指针 `NULL`。

由于字符串指针可以看成是字符数组，所以下面三种写法是等价的。

```
// 写法一
int main(int argc, char* argv[])

// 写法二
int main(int argc, char** argv)

// 写法三
int main(int argc, char argv[][])
```

另一方面，每个命令行参数既可以写成数组形式 `argv[i]`，也可以写成指针形式 `*(argv + i)`。

利用 `argc`，可以限定函数只能有多少个参数。

```
#include <stdio.h>

int main(int argc, char** argv) {
    if (argc != 3) {
        printf("usage: mult x y\n");
        return 1;
    }

    printf("%d\n", atoi(argv[1]) * atoi(argv[2]));
    return 0;
}
```

上面示例中，`argc` 不等于 3 就会报错，这样就限定了程序必须有两个参数，才能运行。

另外，`argv` 数组的最后一个成员是 `NULL` 指针（`argv[argc] == NULL`）。所以，参数的遍历也可以写成下面这样。

```
for (char** p = argv; *p != NULL; p++) {  
    printf("arg: %s\n", *p);  
}
```

上面示例中，指针 `p` 依次移动，指向 `argv` 的每个成员，一旦移到空指针 `NULL`，就表示遍历结束。由于 `argv` 的地址是固定的，不能执行自增运算（`argv++`），所以必须通过一个中间变量 `p`，完成遍历操作。

## 退出状态

C 语言规定，如果 `main()` 函数没有 `return` 语句，那么结束运行的时候，默认会添加一句 `return 0`，即返回整数 `0`。这就是为什么 `main()` 语句通常约定返回一个整数值，并且返回整数 `0` 表示程序运行成功。如果返回非零值，就表示程序运行出了问题。

Bash 的环境变量 `$?` 可以用来读取上一个命令的返回值，从而知道是否运行成功。

```
$ ./foo hello world  
$ echo $?  
0
```

上面示例中，`echo $?` 用来打印环境变量 `$?` 的值，该值为 `0`，就表示上一条命令运行成功，否则就是运行失败。

注意，只有 `main()` 会默认添加 `return 0`，其他函数都没有这个机制。

## 环境变量

C 语言提供了 `getenv()` 函数（原型在 `stdlib.h`）用来读取命令行环境变量。

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void) {  
    char* val = getenv("HOME");  
  
    if (val == NULL) {  
        printf("Cannot find the HOME environment variable\n");  
        return 1;  
    }  
}
```



```
printf("Value: %s\n", val);  
return 0;  
}
```

上面示例中，`getenv("HOME")`用来获取命令行的环境变量 `$HOME`，如果这个变量为空（`NULL`），则程序报错返回。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！



## 22、多字节字符

本章介绍 C 语言如何处理非英语字符。

### Unicode 简介

C 语言诞生时，只考虑了英语字符，使用7位的 ASCII 码表示所有字符。ASCII 码的范围是0到127，也就是100多个字符，所以 `char` 类型只占用一个字节，

但是，如果处理非英语字符，一个字节就不够了，单单是中文，就至少有几万个字符，字符集就势必使用多个字节表示。

最初，不同国家有自己的字符编码方式，这样不便于多种字符的混用。因此，后来就逐渐统一到 Unicode 编码，将所有字符放入一个字符集。

Unicode 为每个字符提供一个号码，称为码点（code point），其中0到127的部分，跟 ASCII 码是重合的。通常使用“U+十六进制码点”表示一个字符，比如 U+0041 表示字母 A。

Unicode 编码目前一共包含了100多万个字符，码点范围是 U+0000 到 U+10FFFF。完整表达整个 Unicode 字符集，至少需要三个字节。但是，并不是所有文档都需要那么多字符，比如对于 ASCII 码就够用的英语文档，如果每个字符使用三个字节表示，就会比单字节表示的文件体积大出三倍。

为了适应不同的使用需求，Unicode 标准委员会提供了三种不同的表示方法，表示 Unicode 码点。

- UTF-8：使用1个到4个字节，表示一个码点。不同的字符占用的字节数不一样。
- UTF-16：对于U+0000 到 U+FFFF 的字符（称为基本平面），使用2个字节表示一个码点。其他字符使用4个字节。
- UTF-32：统一使用4个字节，表示一个码点。

其中，UTF-8 的使用最为广泛，因为对于 ASCII 字符（U+0000 到 U+007F），它只使用一个字节表示，这就跟 ASCII 的编码方式完全一样。

C 语言提供了两个宏，表示当前系统支持的编码字节长度。这两个宏都定义在头文件 limits.h。

- MB\_LEN\_MAX：任意支持地区的最大字节长度，定义在 limits.h。
- MB\_CUR\_MAX：当前语言的最大字节长度，总是小于或等于 MB\_LEN\_MAX，定义在 stdlib.h。

## 字符的表示方法

字符表示法的本质，是将每个字符映射为一个整数，然后从编码表获得该整数对应的字符。

C 语言提供了不同的写法，用来表示字符的整数号码。

- \123：以八进制值表示一个字符，斜杠后面需要三个数字。
- \x4D：以十六进制表示一个字符，\x 后面是十六进制整数。
- \u2620：以 Unicode 码点表示一个字符（不适用于 ASCII 字符），码点以十六进制表示，\u 后面需要4个字符。
- \U0001243F：以 Unicode 码点表示一个字符（不适用于 ASCII 字符），码点以十六进制表示，\U 后面需要8个字符。

```
printf("ABC\n");  
printf("\101\102\103\n");  
printf("\x41\x42\x43\n");
```

上面三行都会输出“ABC”。

```
printf("\u2022 Bullet 1\n");  
printf("\U00002022 Bullet 1\n");
```

上面两行都会输出“• Bullet 1”。

## 多字节字符的表示

C 语言预设只有基本字符，才能使用字面量表示，其它字符都应该使用码点表示，并且当前系统还必须支持该码点的编码方法。

所谓基本字符，指的是所有可打印的 ASCII 字符，但是有三个字符除外：@、\$、`。

因此，遇到非英语字符，应该将其写成 Unicode 码点形式。

```
char* s = "\u6625\u5929";  
printf("%s\n", s); // 春天
```

上面代码会输出中文“春天”。

如果当前系统是 UTF-8 编码，可以直接用字面量表示多字节字符。

```
char* s = "春天";  
printf("%s\n", s);
```

注意，`\u + 码点` 和 `\U + 码点` 的写法，不能用来表示 ASCII 码字符（码点小于 0xA0 的字符），只有三个字符除外：0x24（\$），0x40（@）和 0x60（`）。

```
char* s = "\u0024\u0040\u0060";  
printf("%s\n", s); // @$`
```

上面代码会输出三个 Unicode 字符“@\$”，但是其它 ASCII 字符都不能用这种表示法表示。

为了保证程序执行时，字符能够正确解读，最好将程序环境切换到本地化环境。

```
set_locale(LC_ALL, "");
```

上面代码中，使用 `set_locale()` 切换到执行环境切换到系统的本地化语言。`set_locale()` 的原型定义在头文件 `locale.h`，详见标准库部分的《`locale.h`》章节。

像下面这样，指定编码语言也可以。

```
setlocale(LC_ALL, "zh_CN.UTF-8");
```

上面代码将程序执行环境，切换到中文环境的 UTF-8 编码。

C 语言允许使用 `u8` 前缀，对多字节字符串指定编码方式为 UTF-8。

```
char* s = u8"春天";  
printf("%s\n", s);
```

一旦字符串里面包含多字节字符，就意味着字符串的字节数与字符数不再一一对应了。比如，字符串的长度为10字节，就不再是包含10个字符，而可能只包含7个字符、5个字符等等。

```
set_locale(LC_ALL, "");  
  
char* s = "春天";  
printf("%d\n", strlen(s)); // 6
```

上面示例中，字符串 `s` 只包含两个字符，但是 `strlen()` 返回的结果却是6，表示这两个字符一共占据了6个字节。

C 语言的字符串函数只针对单字节字符有效，对于多字节字符都会失效，比如

`strtok()`、`strchr()`、`strspn()`、`toupper()`、`tolower()`、`isalpha()` 等不会得到正确结果。

## 宽字符

上一小节的多字节字符串，每个字符的字节宽度是可变的。这种编码方式虽然使用起来方便，但是很不利于字符串处理，因此必须逐一检查每个字符占用的字节数。所以除了这种方式，C 语言还提供了确定宽度的多字节字符存储方式，称为宽字符（wide character）。

所谓“宽字符”，就是每个字符占用的字节数是固定的，要么是2个字节，要么是4个字节。这样的话，就很容易快速处理。

宽字符有一个单独的数据类型 `wchar_t`，每个宽字符都是这个类型。它属于整数类型的别名，可能是有符号的，也可能是无符号的，由当前实现决定。该类型的长度为16位（2个字节）或32位（4个字节），足以容纳当前系统的所有字符。它定义在头文件 `wchar.h` 里面。

宽字符的字面量必须加上前缀“L”，否则 C 语言会把字面量当作窄字符类型处理。

```
set_locale(LC_ALL, "");

wchar_t c = L'牛';
printf("%lc\n", c);

wchar_t* s = L"春天";
printf("%ls\n", s);
```

上面示例中，前缀“L”在单引号前面，表示宽字符，对应 `printf()` 的占位符为 `%lc`；在双引号前面，表示宽字符串，对应 `printf()` 的占位符为 `%ls`。

宽字符串的结尾也有一个空字符，不过是宽空字符，占用多个字节。

处理宽字符，需要使用宽字符专用的函数，绝大部分都定义在头文件 `wchar.h`。

## 多字节字符处理函数

### `mblen()`

`mblen()` 函数返回一个多字节字符占用的字符数。它的原型定义在头文件 `stdlib.h`。

```
int mblen(const char* mbstr, size_t n);
```

它接受两个参数，第一个参数是多字节字符串指针，一般会检查该字符串的第一个字符；第二个参数是需要检查的字节数，这个数字不能大于当前系统单个字符占用的最大字节，一般使用 `MB_CUR_MAX`。

它的返回值是该字符占用的字节数。如果当前字符是空的宽字符，则返回 0；如果当前字符不是有效的多字节字符，则返回 -1。

```

setlocale(LC_ALL, "");

char* mbs1 = "春天";
printf("%d\n", mblen(mbs1, MB_CUR_MAX)); // 3

char* mbs2 = "abc";
printf("%d\n", mblen(mbs2, MB_CUR_MAX)); // 1

```

上面示例中，字符串“春天”的第一个字符“春”，占用3个字节；字符串“abc”的第一个字符“a”，占用1个字节。

## wctomb()

wctomb() 函数（wide character to multibyte）用于将宽字符转为多字节字符。它的原型定义在头文件 `stdlib.h` 。

```
int wctomb(char* s, wchar_t wc);
```

wctomb() 接受两个参数，第一个参数是作为目标的多字节字符数组，第二个参数是需要转换的一个宽字符。它的返回值是多字节字符存储占用的字节数量，如果无法转换，则返回 -1 。

```

setlocale(LC_ALL, "");

wchar_t wc = L'牛';
char mbStr[10] = "";

int nBytes = 0;
nBytes = wctomb(mbStr, wc);

printf("%s\n", mbStr); // 牛
printf("%d\n", nBytes); // 3

```

上面示例中，wctomb() 将宽字符“牛”转为多字节字符，wctomb() 的返回值表示转换后的多字节字符占用3个字节。

## mbtowc()

mbtowc() 用于将多字节字符转为宽字符。它的原型定义在头文件 `stdlib.h` 。

```
int mbtowc(  
    wchar_t* wchar,  
    const char* mbchar,  
    size_t count  
);
```

它接受3个参数，第一个参数是作为目标的宽字符指针，第二个参数是待转换的多字节字符指针，第三个参数是多字节字符的字节数。

它的返回值是多字节字符的字节数，如果转换失败，则返回 -1 。

```
setlocale(LC_ALL, "");  
  
char* mbchar = "牛";  
wchar_t wc;  
wchar_t* pwc = &wc;  
  
int nBytes = 0;  
nBytes = mbtowc(pwc, mbchar, 3);  
  
printf("%d\n", nBytes); // 3  
printf("%lc\n", *pwc); // 牛
```

上面示例中，`mbtowc()` 将多字节字符“牛”转为宽字符 `wc`，返回值是 `mbchar` 占用的字节数（占用3个字节）。

## `wcstombs()`

`wcstombs()` 用来将宽字符串转换为多字节字符串。它的原型定义在头文件 `stdlib.h` 。

```
size_t wcstombs(  
    char* mbstr,  
    const wchar_t* wcstr,  
    size_t count  
);
```

它接受三个参数，第一个参数 `mbstr` 是目标的多字节字符串指针，第二个参数 `wcstr` 是待转换的宽字符串指针，第三个参数 `count` 是用来存储多字节字符串的最大字节数。

如果转换成功，它的返回值是成功转换后的多字节字符串的字节数，不包括尾部的字符串终止符；如果转换失败，则返回 -1 。

下面是一个例子。

```
setlocale(LC_ALL, "");

char mbs[20];
wchar_t* wcs = L"春天";

int nBytes = 0;
nBytes = wcstombs(mbs, wcs, 20);

printf("%s\n", mbs); // 春天
printf("%d\n", nBytes); // 6
```

上面示例中，`wcstombs()` 将宽字符串 `wcs` 转为多字节字符串 `mbs`，返回值 6 表示写入 `mbs` 的字符串占用 6 个字节，不包括尾部的字符串终止符。

如果 `wcstombs()` 的第一个参数是 `NULL`，则返回转换成功所需要的目标字符串的字节数。

## `mbstowcs()`

`mbstowcs()` 用来将多字节字符串转换为宽字符串。它的原型定义在头文件 `stdlib.h` 。

```
size_t mbstowcs(
    wchar_t* wctr,
    const char* mbstr,
    size_t count
);
```

它接受三个参数，第一个参数 `wctr` 是目标宽字符串，第二个参数 `mbstr` 是待转换的多字节字符串，第三个参数是待转换的多字节字符串的最大字符数。

转换成功时，它的返回值是成功转换的多字节字符的数量；转换失败时，返回 -1。如果返回值与第三个参数相同，那么转换后的宽字符串不是以 `NULL` 结尾的。

下面是一个例子。



```
setlocale(LC_ALL, "");

char* mbs = "天气不错";
wchar_t wcs[20];

int nBytes = 0;
nBytes = mbstowcs(wcs, mbs, 20);

printf("%ls\n", wcs); // 天气不错
printf("%d\n", nBytes); // 4
```

上面示例中，多字节字符串 `mbs` 被 `mbstowcs()` 转为宽字符串，成功转换了4个字符，所以该函数的返回值为4。

如果 `mbstowcs()` 的第一个参数为 `NULL`，则返回目标宽字符串会包含的字符数量。

---

《C语言入门教程》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“08”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！

