

C

面向过程

目录

初识 C 语言.....	5
C 语言概述	6
简单的 C 程序示例.....	6
示例解释.....	6
#include 指令和头文件	7
main() 函数.....	7
注释.....	7
花括号、函数体和块.....	7
声明.....	7
赋值.....	7
printf() 函数.....	7
return 语句	7
关键字和保留标识符.....	8
数据和 C	10
数据：数据类型关键字.....	10
位、字节和字.....	11
转义序列.....	11
基本数据类型.....	11
整数类型.....	11
浮点类型.....	12
void 类型	13
变量.....	14
常量.....	16
使用数据类型.....	18
强制类型转化 (type)	18
本章小结.....	20
字符串和格式化输入/输出.....	20
字符串简介.....	20

常量和 C 预处理器.....	21
printf() 和 scanf()	21
转换说明.....	21
sacnf() 函数示例.....	26
关键概念.....	28
本章小结.....	28
运算符.....	29
算术运算符.....	29
关系运算符.....	30
逻辑运算符.....	30
位运算符.....	31
赋值运算符.....	33
杂项运算符.....	33
运算符优先级.....	35
逗号运算符.....	36
判断.....	37
if 语句.....	37
if...else 语句	38
if...else if...else 语句	41
嵌套 if 语句.....	43
switch 语句	49
嵌套 switch 语句.....	51
? : 运算符.....	52
循环.....	53
while 循环	53
do...while 循环.....	57
for 循环	59
如何选择循环.....	61
嵌套循环.....	61

循环控制语句.....	66
break 语句.....	66
continue 语句	67
goto 语句	69
字符输入/输出和输入验证.....	72
函数.....	72
数组和指针.....	73
字符串和字符串函数.....	73

初识 C 语言

1972

丹尼斯里奇

(1) 优点:

- 设计特性：自顶向下规划、结构化编程、模块化设计
- 高效性
- 可移植性
- 强大而灵活
- 面向程序员

(2) 缺点:

- 指针危险！

(3) 应用范围:

- 操作系统
- 嵌入式系统

(4) 语言标准

C11 标准：

(5) 使用 C 语言的 7 个步骤

- 定义程序目标
- 设计程序
- 编写代码
- 编译
- 运行程序
- 测试和调试程序
- 维护和修改代码

本章小结

- C 是强大而简洁的编程语言。很好的控制硬件，容易移植。
- C 是编译型语言。C 编译器和链接器是把 C 语言源代码转换成可执行代码的程序

C 语言概述

简单的 C 程序示例

```
#include <stdio.h>

int main(void)
{
    int num;
    num = 1;

    printf("I am a simple ");
    printf("computer.\n");
    printf("My favorite number is %d because it is
first.\n",num);

    return 0;
}
```

示例解释

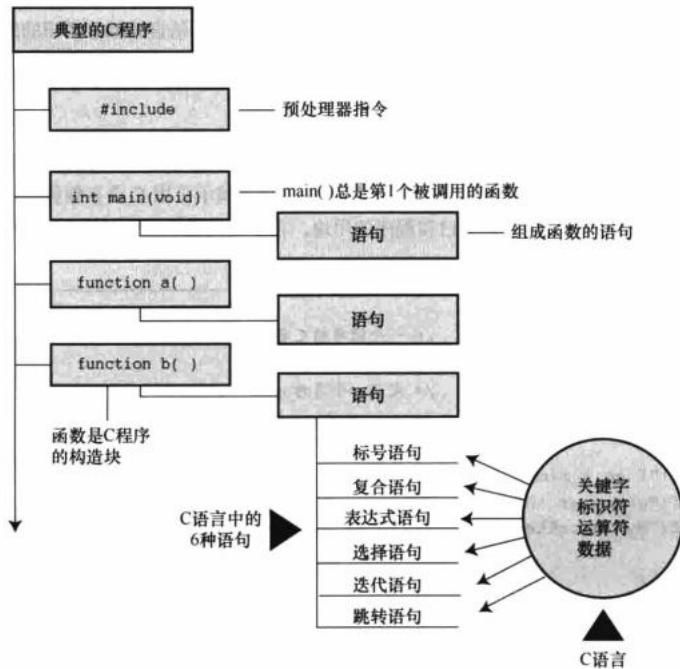


图 2.1 C 程序解剖

程序细节

#include 指令和头文件

main()函数

注释

```
//单行注释  
/*  
多行注释  
多行注释  
多行注释  
*/
```

花括号、函数体和块

声明

声明多个变量用逗号隔开

关键字

标识符（变量名、函数名等）基本要求不报错

字母、数字、下划线；开头为字母或下划线

1. 见名知意
2. 下划线命名法
3. 驼峰命名法
4. 匈牙利命名法

赋值

printf()函数

return 语句

关键字和保留标识符

表 2.2 ISO C 关键字

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

关键字	说明
auto	声明自动变量
break	跳出当前循环
case	开关语句分支
char	声明字符型变量或函数返回值类型
const	定义常量，如果一个变量被 const 修饰，那么它的值就不能再被改变
continue	结束当前循环，开始下一轮循环
default	开关语句中的“其它”分支
do	循环语句的循环体
double	声明双精度浮点型变量或函数返回值类型
else	条件语句否定分支（与 if 连用）
enum	声明枚举类型
extern	声明变量或函数是在其它文件或本文件的其他位置定义
float	声明浮点型变量或函数返回值类型
for	一种循环语句
goto	无条件跳转语句

if	条件语句
int	声明整型变量或函数
long	声明长整型变量或函数返回值类型
register	声明寄存器变量
return	子程序返回语句（可以带参数，也可不带参数）
short	声明短整型变量或函数
signed	声明有符号类型变量或函数
sizeof	计算数据类型或变量长度（即所占字节数）

static	声明静态变量
struct	声明结构体类型
switch	用于开关语句
typedef	用以给数据类型取别名
unsigned	声明无符号类型变量或函数
union	声明共用体类型
void	声明函数无返回值或无参数，声明无类型指针
volatile	说明变量在程序执行中可被隐含地改变
while	循环语句的循环条件

C99 新增关键字

_Bool	_Complex	_Imaginary	inline	restrict
-------	----------	------------	--------	----------

C11 新增关键字

_Alignas	_Alignof	_Atomic	_Generic	_Noreturn
_Static_assert	_Thread_local			

本章小结

C 程序由一个或多个 C 函数组成。每个 C 程序必须包含一个 main() 函数，这是 C 程序要调用的第一个函数。简单的函数由函数头和后面的一对花括号组成，花括号中是由声明、语句组成的函数体。

在 C 语言中，大部分语句都以分号结尾。声明为变量创建变量名和标识该变量中储存的数据类型。变量名是一种标识符。赋值表达式语句把值赋给变量，或者更一般地说，把值赋给存储空间。函数表达式语句用于调用指定的已命名函数。调用函数执行完毕后，程序会返回到函数调用后面的语句继续执行。

printf() 函数用于输出想要表达的内容和变量的值。

一门语言的语法是一套规则，用于管理语言中各有效语句组合在一起的方式。语句的语义是语句要表达的意思。编译器可以检测出语法错误，但是程序里的语义错误只有在编译完之后才能从程序的行为中表现出来。检查程序是否有语义错误要跟踪程序的状态，即程序每执行一步后所有变量的值。

最后，关键字是 C 语言的词汇。

数据和 C

在 C 语言中，数据类型指的是用于声明不同类型的变量或函数的一个广泛的系统。变量的类型决定了变量存储占用的空间，以及如何解释存储的位模式

数据：数据类型关键字

表 3.1 C 语言的数据类型关键字

最初 K&R 给出的关键字	C90 标准添加的关键字	C99 标准添加的关键字
int	signed	_Bool
long	void	_Complex
short		_Imaginary
unsigned		
char		
float		
double		

C 中的类型可分为以下几种：

序号	类型与描述
1	基本类型： 它们是算术类型，包括两种类型：整数类型和浮点类型。
2	枚举类型： 它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值的变量。
3	void 类型： 类型说明符 void 表明没有可用的值。
4	派生类型： 它们包括：指针类型、数组类型、结构类型、共用体类型和函数类型。

在 C 语言中，用 int 关键字来表示基本的整数类型。后 3 个关键字（long、short 和 unsigned）和 C90 新增的 signed 用于提供基本整数类型的变式，例如 unsigned short int 和 long long int。char 关键字用于指定字母和其他字符（如，#、\$、% 和 *）。另外，char 类型也可以表示较小的整数。float、double 和 long double 表示带小数点的数。_Bool 类型表示布尔值（true 或 false），_complex 和 _Imaginary 分别表示复数和虚数。

通过这些关键字创建的类型，按计算机的储存方式可分为两大基本类型：整数类型和浮点数类型。

位、字节和字

1 位 (开和关)

1 字节均为 8 位

1 字 (8 位-16 位-32 位-64 位)

转义序列

表 3.2 转义序列

转义序列	含义
\a	警报 (ANSI C)
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\\\	反斜杠 (\)
\'	单引号
\"	双引号
\?	问号
\ooo	八进制值 (oo 必须是有效的八进制数, 即每个 o 可表示 0~7 中的一个数)
\xhh	十六进制值 (hh 必须是有效的十六进制数, 即每个 h 可表示 0~f 中的一个数)

基本数据类型

整数类型

类型	存储大小	值范围
char	1 字节	-128 到 127 或 0 到 255
unsigned char	1 字节	0 到 255
signed char	1 字节	-128 到 127
int	2 或 4 字节	-32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
unsigned int	2 或 4 字节	0 到 65,535 或 0 到 4,294,967,295
short	2 字节	-32,768 到 32,767
unsigned short	2 字节	0 到 65,535
long	4 字节	-2,147,483,648 到 2,147,483,647
unsigned long	4 字节	0 到 4,294,967,295

注意，各种类型的存储大小与系统位数有关，但目前通用的以64位系统为主。

以下列出了32位系统与64位系统的存储大小的差别（windows 相同）：

Windows vc12	Linux gcc-5.3.1	Compiler		
win32	x64	i686	x86_64	Target
1		1	1	char
1		1	1	unsigned char
2		2	2	short
2		2	2	unsigned short
4		4	4	int
4		4	4	unsigned int
4		8	8	long
4		8	8	unsigned long
4		4	4	float
8		8	8	double
4		4	8	long int
8		8	8	long long
8	12	16		long double

电脑位数与内存的关系!!!

浮点类型

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

类型	存储大小	值范围	精度
float	4 字节	1.2E-38 到 3.4E+38	6 位有效位
double	8 字节	2.3E-308 到 1.7E+308	15 位有效位
long double	16 字节	3.4E-4932 到 1.1E+4932	19 位有效位

头文件 float.h 定义了宏，在程序中可以使用这些值和其他有关实数二进制表示的细节。下面的实例将输出浮点类型占用的存储空间以及它的范围值：

实例

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

int main()
{
    printf("float 存储最大字节数 : %lu \n", sizeof(float));
    printf("float 最小值: %E\n", FLT_MIN );
    printf("float 最大值: %E\n", FLT_MAX );
    printf("精度值: %d\n", FLT_DIG );

    return 0;
}
```

%E 为以指数形式输出单、双精度实数，详细说明查看 [C 库函数 - printf\(\)](#)。

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

```
float 存储最大字节数 : 4
float 最小值: 1.175494E-38
float 最大值: 3.402823E+38
精度值: 6
```

void 类型

void 类型指定没有可用的值。它通常用于以下三种情况下：

序号	类型与描述
1	函数返回为空 C 中有各种函数都不返回值，或者您可以说它们返回空。不返回值的函数的返回类型为空。例如 <code>void exit(int status);</code>
2	函数参数为空 C 中有各种函数不接受任何参数。不带参数的函数可以接受一个 void。例如 <code>int rand(void);</code>
3	指针指向 void 类型为 <code>void *</code> 的指针代表对象的地址，而不是类型。例如，内存分配函数 <code>void *malloc(size_t size);</code> 返回指向 void 的指针，可以转换为任何数据类型。

小结：基本数据类型

关键字：

基本数据类型由 11 个关键字组成：int、long、short、unsigned、char、float、double、signed、_Bool、_Complex 和 _Imaginary。

有符号整型：

有符号整型可用于表示正整数和负整数。

- int——系统给定的基本整数类型。C 语言规定 int 类型不小于 16 位。

- short 或 short int ——最大的 short 类型整数小于或等于最大的 int 类型整数。C 语言规定 short 类型至少占 16 位。
- long 或 long int ——该类型可表示的整数大于或等于最大的 int 类型整数。C 语言规定 long 类型至少占 32 位。
- long long 或 long long int ——该类型可表示的整数大于或等于最大的 long 类型整数。Long long 类型至少占 64 位。

一般而言，long 类型占用的内存比 short 类型大，int 类型的宽度要么和 long 类型相同，要么和 short 类型相同。例如，旧 DOS 系统的 PC 提供 16 位的 short 和 int，以及 32 位的 long；Windows 95 系统提供 16 位的 short 以及 32 位的 int 和 long。

无符号整型：

无符号整型只能用于表示零和正整数，因此无符号整型可表示的正整数比有符号整型的大。在整型类型前加上关键字 unsigned 表明该类型是无符号整型：unsigned int、unsigned long、unsigned short。单独的 unsigned 相当于 unsigned int。

字符类型：

可打印出来的符号（如 A、& 和 +）都是字符。根据定义，char 类型表示一个字符要占用 1 字节内存。出于历史原因，1 字节通常是 8 位，但是如果要表示基本字符集，也可以是 16 位或更大。

- char ——字符类型的关键字。有些编译器使用有符号的 char，而有些则使用无符号的 char。
在需要时，可在 char 前面加上关键字 signed 或 unsigned 来指明具体使用哪一种类型。

布尔类型：

布尔值表示 true 和 false。C 语言用 1 表示 true，0 表示 false。

- _Bool ——布尔类型的关键字。布尔类型是无符号 int 类型，所占用的空间只要能储存 0 或 1 即可。

实浮点类型：

实浮点类型可表示正浮点数和负浮点数。

- float ——系统的基本浮点类型，可精确表示至少 6 位有效数字。
- double ——储存浮点数的范围（可能）更大，能表示比 float 类型更多的有效数字（至少 10 位，通常会更多）和更大的指数。
- long double ——储存浮点数的范围（可能）比 double 更大，能表示比 double 更多的有效数字和更大的指数。

复数和虚数浮点数：

虚数类型是可选的类型。复数的实部和虚部类型都基于实浮点类型来构成：

- float _Complex
- double _Complex
- long double _Complex
- float _Imaginary
- double _Imaginary
- long long _Imaginary

变量

变量只是程序可操作的存储区的名称。C 中每个变量都有特定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可用于变量上。

变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C 是大小写敏感的。基于上小节讲解的基本类型，有以下几种基本的变量类型：

C 语言也允许定义各种其他类型的变量，比如枚举、指针、数组、结构、共用体等等，这将会在后续的章节中进行讲解，本章节我们先讲解基本变量类型。

变量定义与声明

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

type variable list;

在这里, **type** 必须是一个有效的 C 数据类型, 可以是 `char`、`w_char`、`int`、`float`、`double` 或任何用户自定义的对象, **variable list** 可以由一个或多个标识符名称组成, 多个标识符之间用逗号分隔。下面列出几个有效的声明:

```
int    i, j, k;  
char   c, ch;  
float  f, salary;  
double d;
```

变量声明向编译器保证变量以指定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

变量的声明有两种情况：

- 1、一种是需要建立存储空间的。例如：int a 在声明的时候就已经建立了存储空间。
 - 2、另一种是不需要建立存储空间的，通过使用extern关键字声明变量名而不定义它。例如：extern int a 其中变量 a 可以在别的文件中定义的。
 - 除非有extern关键字，否则都是变量的定义。

```
extern int i; //声明，不是定义  
int i; //声明，也是定义
```

小结：如何声明简单变量

1. 选择需要的类型。
2. 使用有效的字符给变量起一个变量名。
3. 按以下格式进行声明：
 类型说明符 变量名；
 类型说明符由一个或多个关键字组成。下面是一些示例：
 int erest;
 unsigned short cash;
4. 可以同时声明相同类型的多个变量，用逗号分隔各变量名，如下所示：
 char ch, init, ans;
5. 在声明的同时还可以初始化变量：
 float mass = 6.0E24;

声明多个变量用逗号隔开

关键字

标识符（变量名、函数名等）基本要求不报错

1. 见名知意
2. 下划线命名法
3. 驼峰命名法
4. 匈牙利命名法

常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做**字面量**。

常量可以是任何的基本数据类型，比如整数常量、浮点常量、字符常量，或字符串字面值，也有枚举常量。

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数 (unsigned)，L 表示长整数 (long)。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212      /* 合法的 */  
215u     /* 合法的 */  
0xFeeL   /* 合法的 */  
078      /* 非法的：8 不是八进制的数字 */  
032UU    /* 非法的：不能重复后缀 */
```

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时，必须包含小数点、指数，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：

```
3.14159      /* 合法的 */
314159E-5L   /* 合法的 */
510E          /* 非法的：不完整的指数 */
210f          /* 非法的：没有小数或指数 */
.e55          /* 非法的：缺少整数或分数 */
```

.e55 既没有整数也没有小数，不可以!!!

字符常量

字符常量是括在单引号中，例如，'x' 可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

转义序列	含义
\\	\字符
'	'字符
"	"字符
\?	?字符
\a	警报铃声
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh ...	一个或多个数字的十六进制数

字符串常量

字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

```
"hello, dear"  
"hello, \  
  dear"  
"hello, " "d" "ear"
```

定义常量

使用#define 预处理器（宏定义）

```
#define identifier value
```

```
#define LENGTH 10  
#define WIDTH 5  
#define NEWLINE '\n'
```

注意常量名一般大写，在函数之外写

使用 const 关键字

```
const type variable = value;
```

```
const int LENGTH = 10;  
const int WIDTH = 5;  
const char NEWLINE = '\n';
```

在函数内写

把常量定义为大写字母形式，是一个很好的编程习惯

使用数据类型

编写程序时，应注意合理选择所需的变量及其类型。通常，用 int 或 float 类型表示数字，char 类型表示字符。在使用变量之前必须先声明，并选择有意义的变量名。初始化变量应使用与变量类型匹配的常数类型。例如：

```
int apples = 3; /* 正确 */  
int oranges = 3.0; /* 不好的形式 */
```

许多程序员和公司内部都有系统化的命名约定，在变量名中体现其类型。例如，用 i_ 前缀表示 int 类型，us_ 前缀表示 unsigned short 类型。这样，一眼就能看出来 i_smart 是 int 类型的变量，us_versmart 是 unsigned short 类型的变量。

强制类型转化 (type)

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

```
void pound(int n);

int main(void)
{
    int times = 5;
    char ch = '!';
    float f = 6.2;

    pound(times);
    pound((int)ch);
    pound((int)f);

    return 0;
}

void pound(int n)
{
    while (n-- > 0)
        printf("#");
    printf("\n");
}
```

本章小结

C 有多种的数据类型。基本数据类型分为两大类：整数类型和浮点数类型。通过为类型分配的储存量以及是有符号还是无符号，区分不同的整数类型。最小的整数类型是 `char`，因实现不同，可以是有符号的 `char` 或无符号的 `char`，即 `unsigned char` 或 `signed char`。但是，通常用 `char` 类型表示小整数时才这样显示说明。其他整数类型有 `short`、`int`、`long` 和 `long long` 类型。C 规定，后面的类型不能小于前面的类型。上述都是有符号类型，但也可以使用 `unsigned` 关键字创建相应的无符号类型：`unsigned short`、`unsigned int`、`unsigned long` 和 `unsigned long long`，或者，在类型名前加上 `signed` 修饰符显式表明该类型是有符号类型。最后，`_Bool` 类型是一种无符号类型，可储存 0 或 1，分别代表 `false` 和 `true`。

浮点类型有 3 种：`float`、`double` 和 C90 新增的 `long double`。后面的类型应大于或等于前面的类型。有些实现可选择支持复数类型和虚数类型，通过关键字 `_Complex` 和 `_Imaginary` 与浮点类型的关键词组合（如，`double _Complex` 类型和 `float _Imaginary` 类型）来表示这些类型。

整数可以表示为十进制、八进制或十六进制。0 前缀表示八进制数，0x 或 0X 前缀表示十六进制数。例如，32、040、0x20 分别以十进制、八进制、十六进制表示同一个值。l 或 L 前缀表明该值是 `long` 类型，ll 或 LL 前缀表明该值是 `long long` 类型。

在 C 语言中，直接表示一个字符常量的方法是：把该字符用单引号括起来，如 'Q'、'8' 和 '\$'。C 语言的转义序列（如，'\n'）表示某些非打印字符。另外，还可以在八进制或十六进制数前加上一个反斜杠（如，'\007'），表示 ASCII 码中的一个字符。

浮点数可写成固定小数点的形式（如，9393.912）或指数形式（如，7.38E10）。C99 和 C11 提供了第 3 种指数表示法，即用十六进制数 2 的幂来表示（如，0xa.1fp10）。

`printf()` 函数根据转换说明打印各种类型的值。转换说明最简单的形式由一个百分号 (%) 和一个转换字符组成，如 %d 或 %f。

字符串和格式化输入/输出

字符串简介

字符串（*character string*）是一个或多个字符的序列，如下所示：

"Zing went the strings of my heart!"

双引号不是字符串的一部分。双引号仅告知编译器它括起来的是字符串，正如单引号用于标识单个字符一样。

C 语言没有专门用于储存字符串的变量类型，字符串都被储存在 `char` 类型的数组中。数组由连续的存储单元组成，字符串中的字符被储存在相邻的存储单元中，每个单元储存一个字符（见图 4.1）。

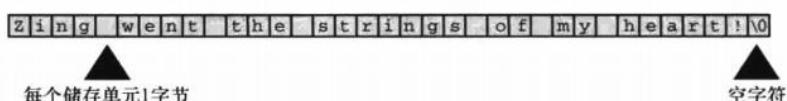


图 4.1 数组中的字符串

注意图 4.1 中数组末尾位置的字符 \0。这是空字符（*null character*），C 语言用它标记字符串的结束。空字符不是数字 0，它是非打印字符，其 ASCII 码值是（或等价于）0。C 中的字符串一定以空字符结束，这意味着数组的容量必须至少比待存储字符串中的字符数多 1。因此，程序清单 4.1 中有 40 个存储单元的字符串，只能储存 39 个字符，剩下一个字节留给空字符。

一般而言，C 把函数库中相关的函数归为一类，并为每类函数提供一个头文件。例如，`printf()` 和 `scanf()` 都隶属标准输入和输出函数，使用 `stdio.h` 头文件。`string.h` 头文件中包含了 `strlen()` 函数和其他一些与字符串相关的函数（如拷贝字符串的函数和字符串查找函数）。

常量和 C 预处理器

请注意格式，首先是#define，接着是符号常量名（TAXRATE），然后是符号常量的值（0.015）（注意，其中并没有=符号）。所以，其通用格式如下：

```
#define NAME value
```

C90 标准新增了 const 关键字，用于限定一个变量为只读¹。其声明如下：

```
const int MONTHS = 12; // MONTHS 在程序中不可更改，值为 12
```

这使得 MONTHS 成为一个只读值。也就是说，可以在计算中使用 MONTHS，可以打印 MONTHS，但是不能更改 MONTHS 的值。const 用起来比#define 更灵活，第 12 章将讨论与 const 相关的内容。

printf() 和 scanf()

printf() 函数和 scanf() 函数能让用户可以与程序交流，它们是输入/输出函数，或简称为 I/O 函数。它们不仅是 C 语言中的 I/O 函数，而且是最才多艺的函数。过去，这些函数和 C 库的一些其他函数一样，并不是 C 语言定义的一部分。最初，C 把输入/输出的实现留给了编译器的作者，这样可以针对特殊的机器更好地匹配输入/输出。后来，考虑到兼容性的问题，各编译器都提供不同版本的 printf() 和 scanf()。尽管如此，各版本之间偶尔有一些差异。C90 和 C99 标准规定了这些函数的标准版本，本书亦遵循这一标准。

虽然 printf() 是输出函数，scanf() 是输入函数，但是它们的工作原理几乎相同。两个函数都使用格式字符串和参数列表。我们先介绍 printf()，再介绍 scanf()。

转换说明

表 4.3 转换说明及其打印的输出结果

转换说明	输出
%a	浮点数、十六进制数和 p 记数法 (C99/C11)
%A	浮点数、十六进制数和 p 记数法 (C99/C11)
%c	单个字符
%d	有符号十进制整数
%e	浮点数，e 记数法
%E	浮点数，e 记数法
%f	浮点数，十进制记数法
%g	根据值的不同，自动选择%f 或%e。%e 格式用于指数小于-4 或者大于或等于精度时
%G	根据值的不同，自动选择%f 或%E。%E 格式用于指数小于-4 或者大于或等于精度时
%i	有符号十进制整数（与%d 相同）
%o	无符号八进制整数
%p	指针
%s	字符串
%u	无符号十进制整数
%x	无符号十六进制整数，使用十六进制数 0f
%X	无符号十六进制整数，使用十六进制数 0F
%%	打印一个百分号

补充

%lf----double

表 4.5 printf() 中的标记

标记	含义
-	待打印项左对齐。即，从字段的左侧开始打印该项 示例："%-20s"
+	有符号值若为正，则在值前面显示加号；若为负，则在值前面显示减号 示例："%+6.2f"
空格	有符号值若为正，则在值前面显示前导空格（不显示任何符号）；若为负，则在值前面显示减号 +标记覆盖一个空格 示例："%6.2f"
#	把结果转换为另一种形式。如果是%o 格式，则以 0 开始；如果是%x 或%X 格式，则以 0x 或 0X 开始；对于所有的浮点格式，#保证了即使后面没有任何数字，也打印一个小数点字符。对于%g 和%G 格式，#防止结果后面的 0 被删除 示例："%#o"、">%#8.0f"、">%#10.3e"
0	对于数值格式，用前导 0 代替空格填充字段宽度。对于整数格式，如果出现-标记或指定精度，则忽略该标记

```
#include <stdio.h>

int main()
{
    const double RENET = 3582.99;

    printf("%f\n", RENET);
    printf("%e\n", RENET);
    printf("%4.2f\n", RENET);
    printf("%3.1f\n", RENET);
    printf("%10.3f\n", RENET);
    printf("%10.3E\n", RENET);
    printf("%+4.2f\n", RENET);
    printf("%010.2f\n", RENET);

    return 0;
}
```

```
3582.990000
3.582990e+03
3582.99
3583.0
 3582.990
 3.583E+03
+3582.99
0003582.99
```

```
请按任意键继续. . . |
```

本例的第 1 个转换说明是%*f*。在这种情况下，字段宽度和小数点后面的位数均为系统默认设置，即字段宽度是容纳带打印数字所需的位数和小数点后打印 6 位数字。

第 2 个转换说明是%*e*。默认情况下，编译器在小数点的左侧打印 1 个数字，在小数点的右侧打印 6 个数字。这样打印的数字太多！解决方案是指定小数点右侧显示的位数，程序中接下来的 4 个例子就是这样做的。请注意，第 4 个和第 6 个例子对输出结果进行了四舍五入。另外，第 6 个例子用*E*代替了*e*。

第 7 个转换说明中包含了+标记，这使得打印的值前面多了一个代数符号(+)。0 标记使得打印的值前面以 0 填充以满足字段要求。注意，转换说明%010.2*f*的第一个 0 是标记，句点(.)之前、标记之后的数字(本例为 10)是指定的字段宽度。

```
#include <stdio.h>

#define BLURB "Authentic imitation!"

int main(void)
{
    printf("[%2s]\n", BLURB);
    printf("[%24s]\n", BLURB);
    printf("[%24.5s]\n", BLURB);
    printf("[%-.24s]\n", BLURB);

    return 0;
}
```

```
[Authentic imitation!]
[      Authentic imitation!]
[                      Authe]
[                           ]
```

```
请按任意键继续. . . |
```

注意，虽然第 1 个转换说明是%2s，但是字段被扩大为可容纳字符串中的所有字符。还需注意，精度限制了待打印字符的个数。.5 告诉 printf() 只打印 5 个字符。另外，- 标记使得文本左对齐输出。

程序清单 4.14 longstrg.c 程序

```
/* longstrg.c --打印较长的字符串 */
#include <stdio.h>
int main(void)
{
    printf("Here's one way to print a ");
    printf("long string.\n");
    printf("Here's another way to print a \
long string.\n");
    printf("Here's the newest way to print a "
           "long string.\n"); /* ANSI C */

    return 0;
}
```

该程序的输出如下：

```
Here's one way to print a long string.
Here's another way to print a long string.
Here's the newest way to print a long string.
```

方法 1：使用多个 printf() 语句。因为第 1 个字符串没有以\n 字符结束，所以第 2 个字符串紧跟第 1 个字符串末尾输出。

方法 2：用反斜杠 (\) 和 Enter (或 Return) 键组合来断行。这使得光标移至下一行，而且字符串中不会包含换行符。其效果是在下一行继续输出。但是，下一行代码必须和程序清单中的代码一样从最左边开始。如果缩进该行，比如缩进 5 个空格，那么这 5 个空格就会成为字符串的一部分。

方法 3：ANSI C 引入的字符串连接。在两个用双引号括起来的字符串之间用空白隔开，C 编译器会把

多个字符串看作是一个字符串。因此，以下 3 种形式是等效的：

```
printf("Hello, young lovers, wherever you are.");
printf("Hello, young "      "lovers" ", wherever you are.");
printf("Hello, young lovers"
      ", wherever you are.");
```

上述方法中，要记得在字符串中包含所需的空格。如，“young”“lovers”会成为“younglovers”，而“young ” “lovers”才是“young lovers”。

!!! 数组与指针!!!

- 如果用 scanf() 读取基本变量类型的值，在变量名前加上一个&；
- 如果用 scanf() 把字符串读入字符数组中，不要使用&。

程序清单 4.15 中的小程序演示了这两条规则。

```
#include <stdio.h>

int main(void)
{
    int age;
    float assets;
    char pet[30] = {0};

    printf("Enter your age, assets, and favorite pet.\n");
    scanf("%d %f", &age, &assets);
    scanf("%s", pet);
```

```

    printf("%d %.2f %s\n", age, assets, pet);

    return 0;
}

```

输入是空格回车都可最后要回车!!!

空格可以一次性读取数据!!! (另外注意与循环的搭配)

```

Enter your age, assets, and favorite pet.
23
0
dog
23 $0.00 dog

请按任意键继续. . .

```

表 4.6 ANSI C 中 `scanf()` 的转换说明

转换说明	含义
%c	把输入解释成字符
%d	把输入解释成有符号十进制整数
%e、%f、%g、%a	把输入解释成浮点数 (C99 标准新增了%a)
%E、%F、%G、%A	把输入解释成浮点数 (C99 标准新增了%A)
%i	把输入解释成有符号十进制整数
%o	把输入解释成有符号八进制整数
%p	把输入解释成指针 (地址)
%s	把输入解释成字符串。从第 1 个非空白字符开始，到下一个空白字符之前的所有字符都是输入
%u	把输入解释成无符号十进制整数
%x、%X	把输入解释成有符号十六进制整数

表 4.7 `scanf()` 转换说明中的修饰符

转换说明	含义
*	抑制赋值 (详见后面解释) 示例: "%*d"
数字	最大字段宽度。输入达到最大字段宽度处，或第 1 次遇到空白字符时停止 示例: "%10s"
hh	把整数作为 <code>signed char</code> 或 <code>unsigned char</code> 类型读取 示例: "%hd"、"%hu"
ll	把整数作为 <code>long long</code> 或 <code>unsigned long long</code> 类型读取 (C99) 示例: "%lld"、"%llu"

转换说明	含义
h、l 或 L	"%hd"和"%hi"表明把对应的值储存为 short int 类型 "%ho"、"%hx"和"%hu"表明把对应的值储存为 unsigned short int 类型 "%ld"和"%li"表明把对应的值储存为 long 类型 "%lo"、"%lx"和"%lu"表明把对应的值储存为 unsigned long 类型 "%le"、"%lf"和"%lg"表明把对应的值储存为 double 类型 在 e、f 和 g 前面使用 L 而不是 l，表明把对应的值被储存为 long double 类型。 如果没有修饰符，d、i、o 和 x 表明对应的值被储存为 int 类型，f 和 g 表明把对应的值储存为 float 类型
j	在整型转换说明后面时，表明使用 intmax_t 或 uintmax_t 类型 (C99) 示例："%zd"、"%zo"
z	在整型转换说明后面时，表明使用 sizeof 的返回类型 (C99)
t	在整型转换说明后面时，表明使用表示两个指针差值的类型 (C99) 示例："%td"、"%tx"

sacnf() 函数示例

```
#include <stdio.h>

int main(void)
{
    int num;

    scanf("%d",&num); //里面啥也没有的！！！
    printf("您输入的数字为%d\n",num);

    return 0;
}
```

高级示例!!! 注意理解特性

```
// 找出 0°C 以下的天数占总天数的百分比
#include <stdio.h>

int main(void)
{
    const int FREEZING = 0;
    float temperature;
    int cold_days = 0;
    int all_days = 0;

    printf("Enter the list of daily low temperatures.\n");
    printf("Use Celsius, and enter q to quit.\n");
    while (scanf("%f", &temperature) == 1)
    {
        all_days++;
        if (temperature < FREEZING)
```

```
{  
    cold_days++;  
}  
}  
if (all_days != 0)  
{  
    printf("%d daays total: %.1f%% were below freezing.\n",  
           all_days, 100.0 * (float)cold_days / all_days);  
}  
if (all_days == 0)  
{  
    printf("No data entered!\n");  
}  
return 0;  
}
```

printf() 和 scanf() 的*修饰符

printf() 和 scanf() 都可以使用*修饰符来修改转换说明的含义。但是，它们的用法不太一样。首先，我们来看 printf() 的*修饰符。

如果你不想预先指定字段宽度，希望通过程序来指定，那么可以用*修饰符代替字段宽度。但还是要用一个参数告诉函数，字段宽度应该是多少。也就是说，如果转换说明是%*d，那么参数列表中应包含*和 d 对应的值。这个技巧也可用于浮点值指定精度和字段宽度。程序清单 4.16 演示了相关用法。

scanf() 中*的用法与此不同。把*放在%和转换字符之间时，会使得 scanf() 跳过相应的输出项。程序清单 4.17 就是一个例子。

关键概念

C 语言用 `char` 类型表示单个字符，用字符串表示字符序列。字符常量是一种字符串形式，即用双引号把字符括起来：“Good luck, my friend”。可以把字符串储存在字符数组（由内存中相邻的字节组成）中。字符串，无论是表示成字符常量还是储存在字符数组中，都以一个叫做空字符的隐藏字符结尾。

在程序中，最好用 `#define` 定义数值常量，用 `const` 关键字声明的变量为只读变量。在程序中使用符号常量（明示常量），提高了程序的可读性和可维护性。

C 语言的标准输入函数（`scanf()`）和标准输出函数（`printf()`）都使用一种系统。在该系统中，第 1 个参数中的转换说明必须与后续参数中的值相匹配。例如，`int` 转换说明 `%d` 与一个浮点值匹配会产生奇怪的结果。必须格外小心，确保转换说明的数量和类型与函数的其余参数相匹配。对于 `scanf()`，一定要记得在变量名前加上地址运算符（`&`）。

空白字符（制表符、空格和换行符）在 `scanf()` 处理输入时起着至关重要的作用。除了 `%c` 模式（读取下一个字符），`scanf()` 在读取输入时会跳过非空白字符前的所有空白字符，然后一直读取字符，直至遇到空白字符或与正在读取字符不匹配的字符。考虑一下，如果 `scanf()` 根据不同的转换说明读取相同的输入行，会发生什么情况。假设有如下输入行：

```
-13.45e12# 0
```

如果其对应的转换说明是 `%d`，`scanf()` 会读取 3 个字符（-13）并停在小数点处，小数点将被留在输入中作为下一次输入的首字符。如果其对应的转换说明是 `%f`，`scanf()` 会读取 -13.45e12，并停在 `#` 符号处，而 `#` 将被留在输入中作为下一次输入的首字符；然后，`scanf()` 把读取的字符序列 -13.45e12 转换成相应的浮点值，并储存在 `float` 类型的目标变量中。如果其对应的转换说明是 `%s`，`scanf()` 会读取 -13.45e12`#`，并停在空格处，空格将被留在输入中作为下一次输入的首字符；然后，`scanf()` 把这 10 个字符的字符码储存在目标字符数组中，并在末尾加上一个空字符。如果其对应的转换说明是 `%c`，`scanf()` 只会读取并储存第 1 个字符，该例中是一个空格¹。

本章小结

字符串是一系列被视为一个处理单元的字符。在 C 语言中，字符串是以空字符（ASCII 码是 0）结尾的一系列字符。可以把字符串储存在字符数组中。数组是一系列同类型的项或元素。下面声明了一个名为 `name`、有 30 个 `char` 类型元素的数组：

```
char name[30];
```

要确保有足够的元素来储存整个字符串（包括空字符）。

字符串常量是用双引号括起来的字符序列，如：“This is an example of a string”。

`scanf()` 函数（声明在 `string.h` 头文件中）可用于获得字符串的长度（末尾的空字符不计算在内）。`scanf()` 函数中的转换说明是 `%s` 时，可读取一个单词。

C 预处理器为预处理器指令（以 `#` 符号开始）查找源代码程序，并在开始编译程序之前处理它们。处理器根据 `#include` 指令把另一个文件中的内容添加到该指令所在的位置。`#define` 指令可以创建明示常量（符号常量），即代表常量的符号。`limits.h` 和 `float.h` 头文件用 `#define` 定义了一组表示整型和浮点型不同属性的符号常量。另外，还可以使用 `const` 限定符创建定义后就不能修改的变量。

`printf()` 和 `scanf()` 函数对输入和输出提供多种支持。两个函数都使用格式字符串，其中包含的转换说明表明待读取或待打印数据项的数量和类型。另外，可以使用转换说明控制输出的外观：字段宽度、小数位和字段内的布局。

运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。

算术运算符

关系运算符

逻辑运算符

位运算符

赋值运算符

其他运算符

算术运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

```
#include <stdio.h>

int main()
{
    int c;
    int a = 10;
    c = a++;
    printf("先赋值后运算: \n");
    printf("Line 1 - c 的值是 %d\n", c );
    printf("Line 2 - a 的值是 %d\n", a );
    a = 10;
    c = a--;
    printf("Line 3 - c 的值是 %d\n", c );
    printf("Line 4 - a 的值是 %d\n", a );

    printf("先运算后赋值: \n");
    a = 10;
    c = ++a;
    printf("Line 5 - c 的值是 %d\n", c );
    printf("Line 6 - a 的值是 %d\n", a );
    a = 10;
    c = --a;
    printf("Line 7 - c 的值是 %d\n", c );
    printf("Line 8 - a 的值是 %d\n", a );
}
```

以上程序执行输出结果为：

```
先赋值后运算：  
Line 1 - c 的值是 10  
Line 2 - a 的值是 11  
Line 3 - c 的值是 10  
Line 4 - a 的值是 9  
先运算后赋值：  
Line 5 - c 的值是 11  
Line 6 - a 的值是 11  
Line 7 - c 的值是 9  
Line 8 - a 的值是 9
```

关系运算符

下表显示了 C 语言支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
<code>==</code>	检查两个操作数的值是否相等，如果相等则条件为真。	$(A == B)$ 为假。
<code>!=</code>	检查两个操作数的值是否相等，如果不相等则条件为真。	$(A != B)$ 为真。
<code>></code>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	$(A > B)$ 为假。
<code><</code>	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	$(A < B)$ 为真。
<code>>=</code>	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	$(A >= B)$ 为假。
<code><=</code>	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	$(A <= B)$ 为真。

逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 **A** 的值为 1，变量 **B** 的值为 0，则：

运算符	描述	实例
<code>&&</code>	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	$(A \&& B)$ 为假。
<code> </code>	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	$(A B)$ 为真。
<code>!</code>	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	$!(A \&& B)$ 为真。

小结：逻辑运算符和表达式

逻辑运算符：

逻辑运算符的运算对象通常是关系表达式。!运算符只需要一个运算对象，其他两个逻辑运算符都需要两个运算对象，左侧一个，右侧一个。

逻辑运算符	含义
&&	与
	或
!	非

逻辑表达式：

当且仅当 expression1 和 expression2 都为真，expression1 && expression2 才为真。如果 expression1 或 expression2 为真，expression1 || expression2 为真。如果 expression 为假，!expression 则为真，反之亦然。

求值顺序：

逻辑表达式的求值顺序是从左往右。一旦发现有使整个表达式为假的因素，立即停止求值。

示例：

```
6 > 2 && 3 == 3      真  
!(6 > 2 && 3 == 3)    假  
x != 0 && (20 / x) < 5 只有当 x 不等于 0 时，才会对第 2 个表达式求值
```

位运算符

位运算符作用于位，并逐位执行操作。&、| 和 ^ 的真值表如下所示：

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

&	按位与操作，按二进制位进行"与"运算。运算规则：	(A & B) 将得到 12，即为 0000 1100
	按位或运算符，按二进制位进行"或"运算。运算规则：	(A B) 将得到 61，即为 0011 1101
^	异或运算符，按二进制位进行"异或"运算。运算规则：	(A ^ B) 将得到 49，即为 0011 0001
~	取反运算符，按二进制位进行"取反"运算。运算规则：	(~A) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。	A >> 2 将得到 15，即为 0000 1111

赋值运算符

下表列出了 C 语言支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
=	按位或且赋值运算符	C = 2 等同于 C = C 2

杂项运算符

下表列出了 C 语言支持的其他一些重要的运算符，包括 **sizeof** 和 **? :**

运算符	描述	实例
sizeof()	返回变量的大小。	sizeof(a) 将返回 4，其中 a 是整数。
&	返回变量的地址。	&a; 将给出变量的实际地址。
*	指向一个变量。	*a; 将指向一个变量。
? :	条件表达式	如果条件为真？则值为 X；否则值为 Y

实例

```
#include <stdio.h>

int main()
{
    int a = 4;
    short b;
    double c;
    int* ptr;

    /* sizeof 运算符实例 */
    printf("Line 1 - 变量 a 的大小 = %lu\n", sizeof(a));
    printf("Line 2 - 变量 b 的大小 = %lu\n", sizeof(b));
    printf("Line 3 - 变量 c 的大小 = %lu\n", sizeof(c));

    /* & 和 * 运算符实例 */
    ptr = &a;      /* 'ptr' 现在包含 'a' 的地址 */
    printf("a 的值是 %d\n", a);
    printf("*ptr 是 %d\n", *ptr);

    /* 三元运算符实例 */
    a = 10;
    b = (a == 1) ? 20: 30;
    printf( "b 的值是 %d\n", b );

    b = (a == 10) ? 20: 30;
    printf( "b 的值是 %d\n", b );
}
```

&返回变量的地址

*指向一个变量

? : 条件表达式；如果条件为真则为前值否则为后值

我们来看程序清单 7.8 中的油漆程序，该程序计算刷给定平方英尺的面积需要多少罐油漆。基本算法很简单：用平方英尺数除以每罐油漆能刷的面积。但是，商店只卖整罐油漆，不会拆分来卖，所以如果计算结果是 1.7 罐，就需要两罐。因此，该程序计算得到带小数的结果时应该进 1。条件运算符常用于处理这种情况，而且还要根据单复数分别打印 can 和 cans。

```
// 使用条件运算符
#include <stdio.h>
#define COVERAGE 350

int main(void)
{
    int sq_feet;
    int cans;

    printf("Enter number of square feet to be painted:\n");
    while (scanf("%d", &sq_feet) == 1)
    {
        cans = sq_feet / COVERAGE;
```

```

        cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
        printf("You need %d %s of paint.\n", cans, cans == 1 ? "can" :
"cans");
        printf("Enter next value (q to quit):\n");
    }
    return 0;
}

```

小结: 条件运算符

条件运算符: ?:

一般注解:

条件运算符需要 3 个运算对象，每个运算对象都是一个表达式。其通用形式如下：

`expression1 ? expression2 : expression3`

如果 `expression1` 为真，整个条件表达式的值是 `expression2` 的值；否则，是 `expression3` 的值。

示例:

`(5 > 3) ? 1 : 2` 值为 1

`(3 > 5) ? 1 : 2` 值为 2

`(a > b) ? a : b` 如果 `a >b`，则取较大的值

运算符优先级

类别	运算符	结合性
后缀	<code>() [] -> . ++ --</code>	从左到右
一元	<code>+ - ! ~ ++ -- (type)* & sizeof</code>	从右到左
乘除	<code>* / %</code>	从左到右
加减	<code>+ -</code>	从左到右
移位	<code><< >></code>	从左到右
关系	<code>< <= > >=</code>	从左到右
相等	<code>== !=</code>	从左到右
位与 AND	<code>&</code>	从左到右
位异或 XOR	<code>^</code>	从左到右
位或 OR	<code> </code>	从左到右
逻辑与 AND	<code>&&</code>	从左到右
逻辑或 OR	<code> </code>	从左到右
条件	<code>? :</code>	从右到左
赋值	<code>= += -= *= /= %= >>= <<= &= ^= =</code>	从右到左
逗号	<code>,</code>	从左到右

这里还有一个例子：

```
if (number != 0 && 12/number == 2)
    printf("The number is 5 or 6.\n");
```

如果 `number` 的值是 0，那么第 1 个子表达式为假，且不再对关系表达式求值。这样避免了把 0 作为除数。许多语言都没有这种特性，知道 `number` 为 0 后，仍继续检查后面的条件。

逗号运算符

整个逗号表达式的值是右侧项的值

逗号运算符：

逗号运算符把两个表达式连接成一个表达式，并保证最左边的表达式最先求值。逗号运算符通常在 `for` 循环头的表达式中用于包含更多的信息。整个逗号表达式的值是逗号右侧表达式的值。

示例：

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
    fargo += step;
```

```
// 一类邮资
#include <stdio.h>

int main(void)
{
    const int FIRST_OZ = 46;
    const int NEXT_OZ = 20;
    int ounces, cost;

    printf("ounces cost\n");
    for (ounces = 1, cost = FIRST_OZ; ounces <= 16; ounces++, cost +=
NEXT_OZ)
    {
        printf("%5d $%4.2f\n", ounces, cost / 100.0);
    }
    return 0;
}
```

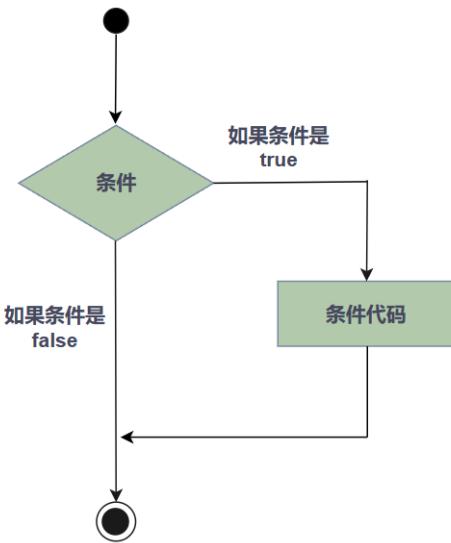
判断

if 语句

一个 if 语句 由一个布尔表达式后跟一个或多个语句组成。

```
if(boolean_expression)
{
    /* 如果布尔表达式为真将执行的语句 */
}
```

流程图



www.runoob.com

Demo

```

#include <stdio.h>

int main()
{
    /* 局部变量定义 */
    int a = 10;

    /* 使用 if 语句检查布尔条件 */
    if (a < 20)
    {
        /* 如果条件为真，则输出下面的语句 */
        printf("a 小于 20\n");
    }
    printf("a 的值是 %d\n", a);

    return 0;
}

```

if...else 语句

一个 **if** 语句 后可跟一个可选的 **else** 语句，**else** 语句在布尔表达式为 **false** 时执行。

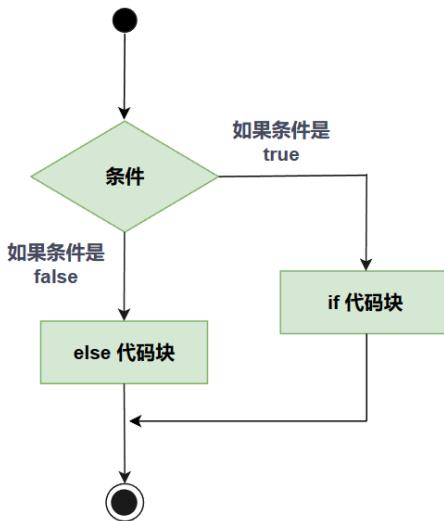
```

if(boolean_expression)
{
    /* 如果布尔表达式为真将执行的语句 */
}
else
{
    /* 如果布尔表达式为假将执行的语句 */
}

```

```
}
```

流程图



www.runoob.com

```
#include <stdio.h>

int main()
{
    /* 局部变量定义 */
    int a = 100;

    /* 检查布尔条件 */
    if (a < 20)
    {
        /* 如果条件为真，则输出下面的语句 */
        printf("a 小于 20\n");
    }
    else
    {
        /* 如果条件为假，则输出下面的语句 */
        printf("a 大于 20\n");
    }
    printf("a 的值是 %d\n", a);

    return 0;
}
```

与 `getchar()` 和 `putchar()` 的配合使用读取字符

```
// 更改输入，空格不变
#include <stdio.h>
#define SPACE ' '
int main(void)
```

```
{  
    char ch;  
  
    ch = getchar();  
    while (ch != '\n')  
    {  
        if (ch == SPACE)  
            putchar(ch);  
        else  
            putchar(ch + 1);  
        ch = getchar();  
    }  
    putchar(ch);  
  
    return 0;  
}
```

更高级的写法

```
// 替换输入的字母，非字母字符保持不变  
#include <stdio.h>  
#include <ctype.h>  
  
int main(void)  
{  
    char ch;  
  
    while ((ch = getchar()) != '\n')  
    {  
        if (isalpha(ch))  
            putchar(ch + 1);  
        else  
            putchar(ch);  
    }  
    putchar(ch); // 显示换行符，看循环条件（dog）  
    return 0;  
}
```

表 7.1 ctype.h 头文件中的字符测试函数

函数名	如果是下列参数时，返回值为真
isalnum()	字母数字（字母或数字）
isalpha()	字母
isblank()	标准的空白字符（空格、水平制表符或换行符）或任何其他本地化指定为白色的字符
iscntrl()	控制字符，如 Ctrl+B
isdigit()	数字
isgraph()	除空格之外的任意可打印字符
islower()	小写字母
isprint()	可打印字符
ispunct()	标点符号（除空格或字母数字字符以外的任何可打印字符）
isspace()	空白字符（空格、换行符、换页符、回车符、垂直制表符、水平制表符或其他本地化定义的字符）
isupper()	大写字母
isxdigit()	十六进制数字符

表 7.2 ctype.h 头文件中的字符映射函数

函数名	行为
tolower()	如果参数是大写字符，该函数返回小写字符；否则，返回原始参数
toupper()	如果参数是小写字符，该函数返回大写字符；否则，返回原始参数

if...else if...else 语句

一个 `if` 语句后可跟一个可选的 `else if...else` 语句，这可用于测试多种条件。

当使用 `if...else if...else` 语句时，以下几点需要注意：

- 一个 `if` 后可跟零个或一个 `else`，`else` 必须在所有 `else if` 之后。
- 一个 `if` 后可跟零个或多个 `else if`，`else if` 必须在 `else` 之前。
- 一旦某个 `else if` 匹配成功，其他的 `else if` 或 `else` 将不会被测试。

```
if(boolean_expression 1)
{
    /* 当布尔表达式 1 为真时执行 */
}
else if( boolean_expression 2)
{
    /* 当布尔表达式 2 为真时执行 */
}
else if( boolean_expression 3)
{
    /* 当布尔表达式 3 为真时执行 */
}
else
{
```

```
/* 当上面条件都不为真时执行 */  
}
```

实际上，`else if` 是已学过的 `if else` 语句的变式。例如，该程序的核心部分只不过是下面代码的另一种写法：

```
if (kwh <= BREAK1)  
    bill = RATE1 * kwh;  
else  
    if (kwh <= BREAK2)          // 360~468 kwh  
        bill = BASE1 + (RATE2 * (kwh - BREAK1));  
    else  
        if (kwh <= BREAK3)      // 468~720 kwh  
            bill = BASE2 + (RATE3 * (kwh - BREAK2));  
        else                    // 超过 720 kwh  
            bill = BASE3 + (RATE4 * (kwh - BREAK3));  
  
#include <stdio.h>  
  
int main ()  
{  
    int a = 100;  
  
    if( a == 10 )  
    {  
        printf("a 的值是 10\n" );  
    }  
    else if( a == 20 )  
    {  
        printf("a 的值是 20\n" );  
    }  
    else if( a == 30 )  
    {  
        printf("a 的值是 30\n" );  
    }  
    else  
    {  
        printf("没有匹配的值\n" );  
    }  
    printf("a 的准确值是 %d\n", a );  
  
    return 0;  
}
```

Tips:

- 如果没有花括号，`else` 与离他最近的 `if` 匹配

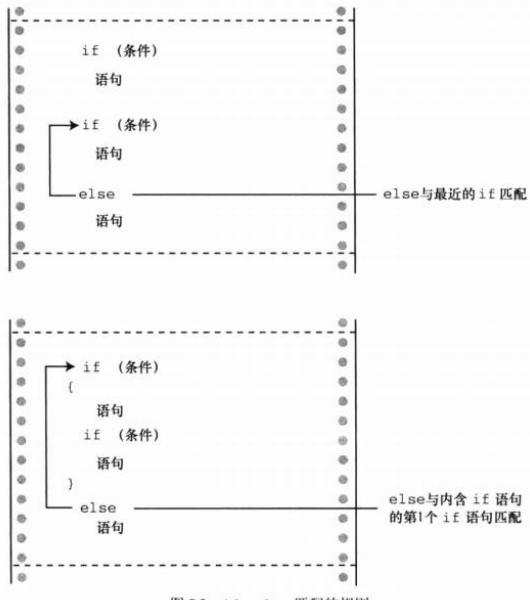


图 7.2 if else 匹配的规则

嵌套 if 语句

在 C 语言中，嵌套 if-else 语句是合法的，这意味着您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。

```
if( boolean_expression 1)
{
    /* 当布尔表达式 1 为真时执行 */
    if(boolean_expression 2)
    {
        /* 当布尔表达式 2 为真时执行 */
    }
}
```

```
#include <stdio.h>

int main ()
{
    int a = 100;
    int b = 200;

    if( a == 100 )
    {
        if( b == 200 )
        {
            printf("a 的值是 100, 且 b 的值是 200\n");
        }
    }
    printf("a 的准确值是 %d\n", a );
}
```

```
    printf("b 的准确值是 %d\n", b );  
  
    return 0;  
}
```

把嵌套 if 应用在下面的程序中：

给定一个整数，显示所有能整除它的约数。如果没有约数，则报告该数是一个素数。

在编写程序的代码之前要先规划好。首先，要总体设计一下程序。为方便起见，程序应该使用一个循环让用户能连续输入待测试的数。这样，测试一个新的数字时不必每次都要重新运行程序。下面是我们为这种循环开发的一个模型（伪代码）：

```
提示用户输入数字  
当 scanf() 返回值为 1  
    分析该数并报告结果  
    提示用户继续输入
```

使用 `scanf()`，把读取数字和判断测试条件确定是否结束循环合并在一起。

下一步，设计如何找出约数。也许最直接的方法是：

```
for (div = 2; div < num; div++)  
    if (num % div == 0)  
        printf("%d is divisible by %d\n", num, div);
```

优化代码：

该循环检查 $2 \sim num$ 之间的所有数字，测试它们是否能被 `num` 整除。但是，这个方法有点浪费时间。我们可以改进一下。例如，考虑如果 $144 \div 2$ 得 0，说明 2 是 144 的约数；如果 144 除以 2 得 72，那么 72 也是 144 的一个约数。所以，`num % div` 测试成功可以获得两个约数。为了弄清其中的原理，我们分析一下循环中得到的成对约数：2 和 72、2 和 48、4 和 36、6 和 24、8 和 18、9 和 16、12 和 12、16 和 9、18 和 8，等等。在得到 12 和 12 这对约数后，又开始得到已找到的相同约数（次序相反）。因此，不用循环到 143，在达到 12 以后就可以停止循环。这大大地节省了循环时间！

分析后发现，必须测试的数只要到 `num` 的平方根就可以了，不用到 `num`。对于 9 这样的数字，不会节约很多时间，但是对于 10000 这样的数，使用哪一种方法求约数差别很大。不过，我们不用在程序中计算平方根，可以这样编写测试条件：

```
for (div = 2; (div * div) <= num; div++)  
    if (num % div == 0)  
        printf("%d is divisible by %d and %d.\n", num, div, num / div);
```

如果 `num` 是 144，当 `div = 12` 时停止循环。如果 `num` 是 145，当 `div = 13` 时停止循环。

不使用平方根而用这样的测试条件，有两个原因。其一，整数乘法比求平方根快。其二，我们还没有正式介绍平方根函数。

还要解决两个问题才能准备编程。第 1 个问题，如果待测试的数是一个完全平方数怎么办？报告 144 可以被 12 和 12 整除显得有点傻。可以使用嵌套 if 语句测试 div 是否等于 num / div。如果是，程序只打印一个约数：

```
for (div = 2; (div * div) <= num; div++)
{
    if (num % div == 0)
    {
        if (div * div != num)
            printf("%d is divisible by %d and %d.\n", num, div, num / div);
        else
            printf("%d is divisible by %d.\n", num, div);
    }
}
```

注意

从技术角度看，if else 语句作为一条单独的语句，不必使用花括号。外层 if 也是一条单独的语句，也不必使用花括号。但是，当语句太长时，使用花括号能提高代码的可读性，而且还可防止今后在 if 循环中添加其他语句时忘记加花括号。

(你怎么现在才说，妈的)

第 2 个问题，如何知道一个数字是素数？如果 num 是素数，程序流不会进入 if 语句。要解决这个问题，可以在外层循环把一个变量设置为某个值（如，1），然后在 if 语句中把该变量重新设置为 0。循环完成后，检查该变量是否是 1，如果是，说明没有进入 if 语句，那么该数就是素数。这样的变量通常称为标记 (flag)。

一直以来，C 都习惯用 int 作为标记的类型，其实新增的 _Bool 类型更合适。另外，如果在程序中包含了 stdbool.h 头文件，便可用 bool 代替 _Bool 类型，用 true 和 false 分别代替 1 和 0。

下面程序体现了以上分析的思路。

为扩大该程序的应用范围，程序用 long 类型而不是 int 类型

```
// 使用嵌套 if 语句显示一个数的约数
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    unsigned long num;//待测试的数
    unsigned long div;//可能的约数
    bool isPrime;//素数标记

    printf("Please enter an integer for analysis: ");
    printf("Enter q to quit.\n");
    while (scanf("%lu", &num) == 1)
    {
        for (div = 2, isPrime = true; (div * div) <= num; div++)
        {
            if (num % div == 0)
            {
```

```
        if ((div * div) != num)
        {
            printf("%lu is divisible by %lu and %lu.\n", num,
div, num / div);
        }
        else
        {
            printf("%lu is divisible by %lu.\n",
num, div);
        }
        isPrime = false;
    }
}
if (isPrime)
{
    printf("%lu is prime.\n", num);
}
printf("Please enter another integer for analysis; Enter q to
quit.\n");
}
return 0;
}
```

注意，该程序在 `for` 循环的测试表达式中使用了逗号运算符，这样每次输入新值时都可以把 `isPrime` 设置为 `true`

小结：

小结：用 `if` 语句进行选择

关键字：`if`、`else`

一般注解：

下面各形式中，`statement` 可以是一条简单语句或复合语句。表达式为真说明其值是非零值。

形式 1:

```
if (expression)
    statement
```

如果 *expression* 为真，则执行 *statement* 部分。

形式 2:

```
if (expression)
    statement1
else
    statement2
```

如果 *expression* 为真，执行 *statement1* 部分；否则，执行 *statement2* 部分。

形式 3:

```
if (expression1)
    statement1
else if (expression2)
    statement2
else
    statement3
```

如果 *expression1* 为真，执行 *statement1* 部分；如果 *expression2* 为真，执行 *statement2* 部分；否则，执行 *statement3* 部分。

示例:

```
if (legs == 4)
    printf("It might be a horse.\n");
else if (legs > 4)
    printf("It is not a horse.\n");
else /* 如果 legs < 4 */
{
    legs++;
    printf("Now it has one more leg.\n");
}
```

一个统计单词的程序

编写一个统计单词数量的程序。该程序还可以计算字符数和行数。

涉及内容如下：

首先，该程序要逐个字符读取输入，知道何时停止读取。然后，该程序能识别并计算这些内容：字符、行数和单词。据此我们编写的伪代码如下：

读取一个字符

当有更多输入时

递增字符计数

如果读完一行，递增行数计数

如果读完一个单词，递增单词计数

读取下一个字符

前面有一个输入循环的模型：

```
while ((ch = getchar()) != STOP)
{
    ...
}
```

这里，STOP 表示能标识输入末尾的某个值。以前我们用过换行符和句点标记输入的末尾，但是对于一个通用的统计单词程序，它们都不合适。我们暂时选用一个文本中不常用的字符（如，|）作为输入的末尾标记。第 8 章中会介绍更好的方法，以便程序既能处理文本文件，又能处理键盘输入。

要查找一个单词里是否有某个字符，可以在程序读入单词的首字符时把一个标记（名为 `inword`）设置为 1。也可以在此时递增单词计数。然后，只要 `inword` 为 1（或 `true`），后续的非空白字符都不记为单词的开始。下一个空白字符，必须重置标记为 0（或 `false`），然后程序就准备好读取下一个单词。我们把以上分析写成伪代码：

这种方法在读到每个单词的开头时把 `inword` 设置为 1（真），在读到每个单词的末尾时把 `inword` 设置为 0（假）。只有在标记从 0 设置为 1 时，递增单词计数。如果能使用 `_Bool` 类型，可以在程序中包含 `stdbool.h` 头文件，把 `inword` 的类型设置为 `bool`，其值用 `true` 和 `false` 表示。如果编译器不支持这种用法，就把 `inword` 的类型设置为 `int`，其值用 1 和 0 表示。

如果使用布尔类型的变量，通常习惯把变量自身作为测试条件。如下所示：

```
用 if (inword) 替代 if (inword == true)  
用 if (!inword) 替代 if (inword == false)
```

可以这样做的原因是，如果 `inword` 为 `true`，则表达式 `inword == true` 为 `true`；如果 `inword` 为 `false`，则表达式 `inword == true` 为 `false`。所以，还不如直接用 `inword` 作为测试条件。类似地，`!inword` 的值与表达式 `inword == false` 的值相同（非真即 `false`，非假即 `true`）。

```
// 统计字符数、单词数、行数  
#include <stdio.h>  
#include <ctype.h>  
#include <stdbool.h>  
#define STOP '|'  
  
int main(void)  
{  
    char c;  
    char prev;  
    long n_chars = 0L;  
    int n_lines = 0;  
    int n_words = 0;  
    int p_lines = 0;  
    bool inword = false;  
  
    printf("Enter text to be analyzed (| to terminate):\n");  
    prev = '\n';  
    while ((c = getchar()) != STOP)  
    {  
        n_chars++;  
        if (c == '\n')  
            n_lines++;  
        if (!isspace(c) && !inword)
```

```

{
    inword = true;
    n_words++;
}
if (isspace(c) && inword)
    inword = false;
prev = c;
}
if (prev != '\n')
    p_lines = 1;
printf("character = %ld, words = %d, lines = %d, partial lines
= %d\n", n_chars, n_words, n_lines, p_lines);

return 0;
}

```

switch 语句

要对紧跟在关键字 switch 后圆括号中的表达式求值。在程序清单 7.11 中，该表达式是刚输入给 ch 的值。然后程序扫描标签（这里指，case 'a' :、case 'b' :等）列表，直到发现一个匹配的值为止。然后程序跳转至那一行。如果没有匹配的标签怎么办？如果有 default : 标签名，就跳转至该行；否则，程序继续执行在 switch 后面的语句。

break 语句在其中起什么作用？它让程序离开 switch 语句，跳至 switch 语句后面的下一条语句（见图 7.4）。如果没有 break 语句，就会从匹配标签开始执行到 switch 末尾。例如，如果删除该程序中的所有 break 语句，运行程序后输入 d，其交互的输出结果如下：

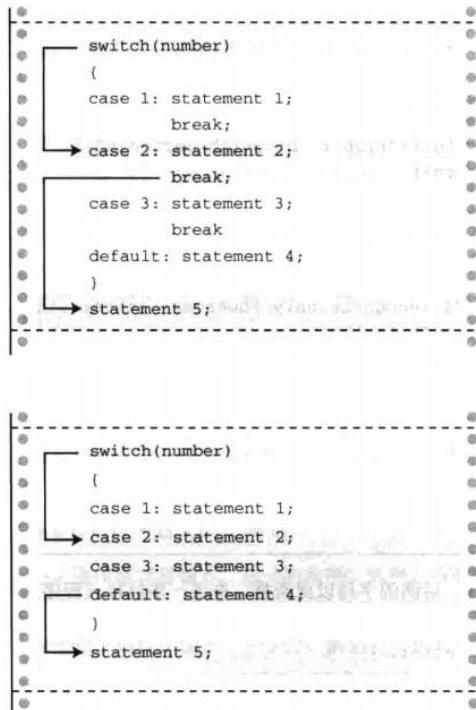


图 7.4 switch 中有 break 和没有 break 的程序流

```
Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
Please type in a letter; type # to end my act.
d [enter]
desman, aquatic, molelike critter
echidna, the spiny anteater
fisher, a brownish marten
That's a stumper!
Please type another letter or a #.
# [enter]
Bye!
```

如上所示，执行了从 `case 'd':` 到 `switch` 语句末尾的所有语句。

一个 `switch` 语句允许测试一个变量等于多个值时的情况。每个值称为一个 `case`，且被测试的变量会对每个 `switch case` 进行检查。

```
switch (expression)
{
    case constant - expression:
        statement(s);
        break; /* 可选的 */
    case constant - expression:
        statement(s);
        break; /* 可选的 */

    /* 您可以有任意数量的 case 语句 */
    default: /* 可选的 */
        statement(s);
}
```

`switch` 语句必须遵循下面的规则：

- `switch` 语句中的 `expression` 是一个常量表达式（不应该使用变量），必须是一个整型或枚举类型。
- 在一个 `switch` 中可以有任意数量的 `case` 语句。每个 `case` 后跟一个要比较的值和一个冒号。
- `case` 的 `constant-expression` 必须与 `switch` 中的变量具有相同的数据类型，且必须是一个常量或字面量。
- 当被测试的变量等于 `case` 中的常量时，`case` 后跟的语句将被执行，直到遇到 `break` 语句为止。
- 当遇到 `break` 语句时，`switch` 终止，控制流将跳转到 `switch` 语句后的下一行。
- 不是每一个 `case` 都需要包含 `break`。如果 `case` 语句不包含 `break`，控制

流将会继续后续的 case，直到遇到 break 为止。

- 一个 switch 语句可以有一个可选的 default case，出现在 switch 的结尾。default case 可用于在上面所有 case 都不为真时执行一个任务。default case 中的 break 语句不是必需的。

```
#include <stdio.h>

int main ()
{
    char grade = 'B';

    switch(grade)
    {
        case 'A' :
            printf("很棒！\n");
            break;
        case 'B' :
        case 'C' :
            printf("做得好\n");
            break;
        case 'D' :
            printf("您通过了\n");
            break;
        case 'F' :
            printf("最好再试一下\n");
            break;
        default :
            printf("无效的成绩\n");
    }
    printf("您的成绩是 %c\n", grade);

    return 0;
}
```

嵌套 switch 语句

您可以把一个 switch 作为一个外部 switch 的语句序列的一部分，即可以在一个 switch 语句内使用另一个 switch 语句。即使内部和外部 switch 的 case 常量包含共同的值，也没有矛盾。

```
switch (ch1)
{
case 'A':
```

```
printf("这个 A 是外部 switch 的一部分");
switch (ch2)
{
case 'A':
    printf("这个 A 是内部 switch 的一部分");
    break;
case 'B': /* 内部 B case 代码 */
}
break;
case 'B': /* 外部 B case 代码 */
}
```

```
#include <stdio.h>

int main()
{
    int a = 100;
    int b = 200;

    switch (a)
    {
    case 100:
        printf("这是外部 switch 的一部分\n");
        switch (b)
        {
        case 200:
            printf("这是内部 switch 的一部分\n");
        }
    }
    printf("a 的准确值是 %d\n", a);
    printf("b 的准确值是 %d\n", b);

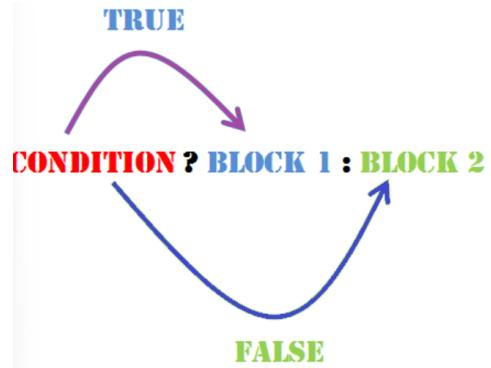
    return 0;
}
```

? : 运算符

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个表达式的值。



```

#include<stdio.h>

int main(void)
{
    int num;

    printf("输入一个数字 : ");
    scanf("%d",&num);

    (num%2==0)?printf("偶数"):printf("奇数");
    return 0;
}

```

循环

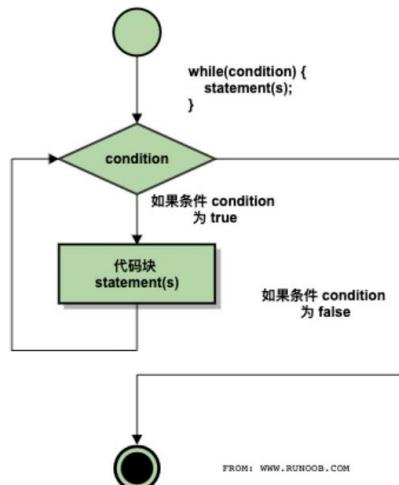
while 循环

```

while(condition)
{
    statement(s);
}

```

流程图



```
#include <stdio.h>

int main()
{
    int a = 10;

    while (a < 20)
    {
        printf("a 的值: %d\n", a);
        a++;
    }

    return 0;
}
```

Tips:

- 判断条件，非零就会执行（要多利用种种特性）
- 变量变化，当突破界限的时候（上述判断条件满足或不满足），结束循环
- 可能会存在初始化的量
- 死循环：嵌入式(如下!!!)

原来如此！对 C 而言，表达式为真的值是 1，表达式为假的值是 0。一些 C 程序使用下面的循环结构，由于 1 为真，所以循环会一直进行。

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

- 只有对测试条件求值时，才决定是终止还是继续循环

```
//根据用户输入的整数求和
#include <stdio.h>
```

```

int main(void)
{
    long num;
    long sum = 0L;
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num); //scanf()返回成功读取项的数量
    while (status == 1)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);
    return 0;
}

```

现在，我们来看看该程序的结构。总结如下：

把 sum 初始化为 0

提示用户输入数据

读取用户输入的数据

当输入的数据为整数时，

 输入添加给 sum，

 提示用户进行输入，

 然后读取下一个输入

输入完成后，打印 sum 的值

顺带一提，这叫作伪代码 (*pseudocode*)，是一种用简单的句子表示程序思路的方法，它与计算机语言的形式相对应。伪代码有助于设计程序的逻辑。确定程序的逻辑无误之后，再把伪代码翻译成实际的编程代码。使用伪代码的好处之一是，可以把注意力集中在程序的组织和逻辑上，不用在设计程序时还要分心如何用编程语言来表达自己的想法。例如，可以用缩进来代表一块代码，不用考虑 C 的语法要用花括号把这部分代码括起来。

总之，因为 while 循环是入口条件循环，程序在进入循环体之前必须获取输入的数据并检查 status 的值，所以在 while 前面要有一个 scanf()。要让循环继续执行，在循环内需要一个读取数据的语句，这样程序才能获取下一个 status 的值，所以在 while 循环末尾还要有一个 scanf()，它为下一次迭代做好了准备。可以把下面的伪代码作为 while 循环的标准格式：

获得第 1 个用于测试的值
当测试为真时
处理值
获取下一个值

(总结的真好!!!) (入口循环) (伪代码结构!!!)

C 风格读取循环（注意是读取，只是循环的一种情况）

```
status = scanf("%ld", &num);
while (status == 1)
{
    /* 循环行为 */
    status = scanf("%ld", &num);
}
```

可以用这些代码替换：

```
while (scanf("%ld", &num) == 1)
{
    /* 循环行为 */
}
```

第二种形式同时使用 `scanf()` 的两种不同的特性。首先，如果函数调用成功，`scanf()` 会把一个值存入 `num`。然后，利用 `scanf()` 的返回值（0 或 1，不是 `num` 的值）控制 `while` 循环。因为每次迭代都会判断循环的条件，所以每次迭代都要调用 `scanf()` 读取新的 `num` 值来做判断。换句话说，C 的语法规则让你可以用下面的精简版本替换标准版本：

当获取值和判断值都成功

处理该值

空语句；所有的任务都在测试条件下完成了

在该例中，测试条件后面的单独分号是空语句（*null statement*），它什么也不做。在 C 语言中，单独的分号表示空语句。有时，程序员会故意使用带空语句的 `while` 语句，因为所有的任务都在测试条件下完成了，不需要在循环体中做什么。例如，假设你想跳过输入到第 1 个非空白字符或数字，可以这样写：

```
while (scanf("%d", &num) == 1)
; /* 跳过整数输入 */
```

只要 `scanf()` 读取一个整数，就会返回 1，循环继续执行。注意，为了提高代码的可读性，应该让这个分号独占一行，不要直接把它放在测试表达式同行。这样做一方面让读者更容易看到空语句，一方面也提醒自己和读者空语句是有意而为之。处理这种情况更好的方法是使用下一章介绍的 `continue` 语句。

小结：while 语句

关键字：while

一般注解：

`while` 语句创建了一个循环，重复执行直到测试表达式为假或 0。`while` 语句是一种入口条件循环，也就是说，在执行多次循环之前已决定是否执行循环。因此，循环有可能不被执行。循环体可以是简单语句，也可以是复合语句。

形式：

```
while ( expression )
    statement
```

在 `expression` 部分为假或 0 之前，重复执行 `statement` 部分。

示例：

```
while (n++ < 100)
    printf(" %d %d\n", n, 2 * n + 1); // 简单语句
while (fargo < 1000)
{ // 复合语句
    fargo = fargo + step;
    step = 2 * step;
}
```

小结：关系运算符和表达式

关系运算符：

每个关系运算符都把它左侧的值和右侧的值进行比较。

<	小于
<=	小于或等于

==	等于
>=	大于或等于
>	大于
!=	不等于

关系表达式：

简单的关系表达式由关系运算符及其运算对象组成。如果关系为真，关系表达式的值为 1；如果关系为假，关系表达式的值为 0。

示例：

5 > 2 为真，关系表达式的值为 1
(2 + a) == a 为假，关系表达式的值为 0

do…while 循环

出口条件循环

```
do
{
    statement(s);

}while( condition );
```

伪代码：

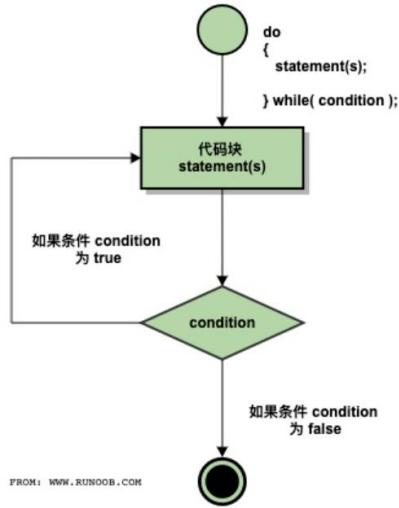
```
do
{
    提示用户输入密码
    读取用户输入的密码
} while ( 用户输入的密码不等于密码 );
```

避免使用这种形式的 do while 结构：

```
do
{
    询问用户是否继续
    其他行为
} while ( 回答是 yes );
```

这样的结构导致用户在回答“no”之后，仍然执行“其他行为”部分，因为测试条件执行晚了。

流程图



```

#include <stdio.h>

int main()
{
    int a = 10;

    /* do 循环执行，在条件被测试之前至少执行一次 */
    do
    {
        printf("a 的值: %d\n", a);
        a = a + 1;
    } while (a < 20);

    return 0;
}

```

Tips:

- 先干活，后判断，判断结果为 `true`，那么再继续执行 `do` 中的代码块
- 结尾有分号

小结: do while 语句

关键字: do while

一般注解:

do while 语句创建一个循环，在 expression 为假或 0 之前重复执行循环体中的内容。do while 语句是一种出口条件循环，即在执行完循环体后才根据测试条件决定是否再次执行循环。因此，该循环至少必须执行一次。statement 部分可是一条简单语句或复合语句。

形式:

```
do
    statement
while ( expression );
在 test 为假或 0 之前，重复执行 statement 部分。
```

示例:

```
do
    scanf("%d", &number);
while ( number != 20 );
```

for 循环

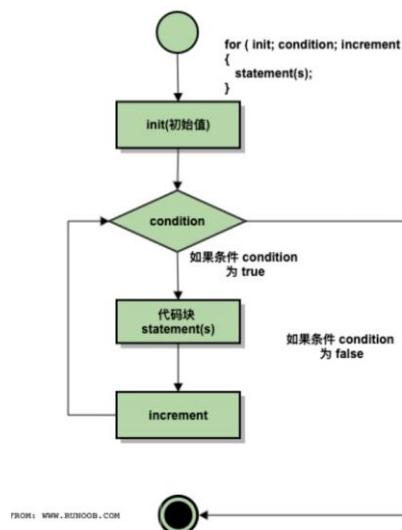
一个重复执行固定次数的循环中涉及了 3 个行为：

1. 必须初始化计数器；
2. 计数器与有限的值作比较；
3. 每次循环时递增计数器；

for 循环把上述 3 个行为（初始化、测试和更新）组合在一处。

```
for ( init; condition; increment )
{
    statement(s);
}
```

流程图



```
#include <stdio.h>

int main()
{
```

```

/* for 循环执行 */
for (int a = 10; a < 20; a = a + 1)
{
    printf("a 的值: %d\n", a);
}

return 0;
}

```

for 循环的其他 9 种用法（第一种递增计数器）：

- 可以使用递减运算符来递减计数器
- 可以让计数器递增 2、10 等
- 可以用字符代替数字计数
- 除了测试迭代次数外，还可以测试其他条件
- 可以让递增的量几何增长，而不是算术增长
- 第 3 个表达式可以使用任意合法的表达式
- 可以省略一个或多个表达式（但是不能省略分号）
- 第一个表达式不一定是给变量赋初值，也可以使用 printf() 只执行一次
- 循环体中的行为可以改变循环头中的表达式

Tips:

- 初始化的变量、
- 判断条件
- 自增变量
- 流程：1, 2, 3, 4, 2, 3, 4, 2, 3, 4……

小结：for 语句

关键字：for

一般注解：

for 语句使用 3 个表达式控制循环过程，分别用分号隔开。initialize 表达式在执行 for 语句之前只执行一次；然后对 test 表达式求值，如果表达式为真（或非零），执行循环一次；接着对 update 表达式求值，并再次检查 test 表达式。for 语句是一种入口条件循环，即在执行循环之前就决定了是否执行循环。因此，for 循环可能一次都不执行。statement 部分可以是一条简单语句或复合语句。

形式：

```
for ( initialize; test; update )
    statement
```

在 test 为假或 0 之前，重复执行 statement 部分。

示例：

```
for (n = 0; n < 10 ; n++)
    printf("%d %d\n", n, 2 * n + 1);
```

如何选择循环

首先，确定是需要入口条件循环还是出口条件循环，通常，入口条件循环用得比较多。

用得比较多，有几个原因。其一，一般原则是在执行循环之前测试条件比较好。其二，测试放在循环的开头，程序的可读性更高。另外，在许多应用中，要求在一开始不满足测试条件时就直接跳过整个循环。

那么，假设需要一个入口条件循环，用 `for` 循环还是 `while` 循环？这取决于个人喜好，因为二者皆可。要让 `for` 循环看起来像 `while` 循环，可以省略第 1 个和第 3 个表达式。例如：

```
for ( ; test ; )
```

与下面的 `while` 效果相同：

```
while ( test )
```

要让 `while` 循环看起来像 `for` 循环，可以在 `while` 循环的前面初始化变量，并在 `while` 循环体中包含更新语句。例如：

初始化；

```
while ( 测试 )
```

```
{
```

 其他语句

 更新语句

```
}
```

与下面的 `for` 循环效果相同：

```
for ( 初始化 ; 测试 ; 更新 )
```

 其他语句

一般而言，当循环涉及初始化和更新变量时，用 `for` 循环比较合适，而在其他情况下用 `while` 循环更好。对于下面这种条件，用 `while` 循环就很合适：

```
while ( scanf("%ld", &num) == 1 )
```

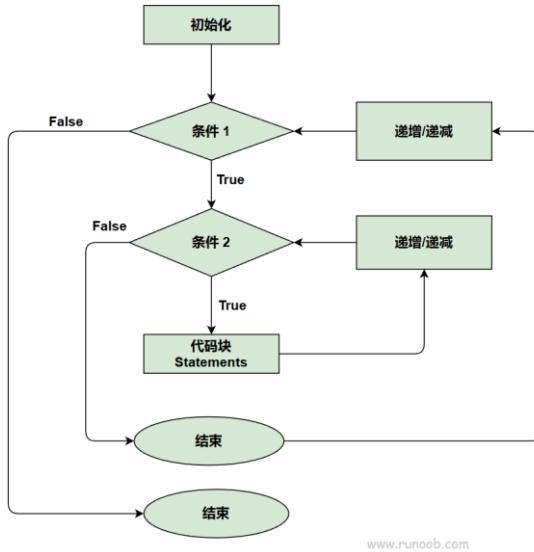
对于涉及索引计数的循环，用 `for` 循环更适合。例如：

```
for (count = 1; count <= 100; count++)
```

嵌套循环

```
for (initialization; condition; increment/decrement)
{
    statement(s);
    for (initialization; condition; increment/decrement)
    {
        statement(s);
        ...
    }
    ...
}
```

流程图



www.runoob.com

```
// for 嵌套
#include <stdio.h>

int main(void)
{
    int row, column;
    printf("九九乘法表\n");
    for (row = 1; row <= 9; row++)
    {
        for (column = 1; column <= 9; column++)
        {
            printf("%d * %d = %2d\t", row, column, row * column);
        }
        printf("\n");
    }
    return 0;
}
```

C:\Windows\system32\cmd.exe + -

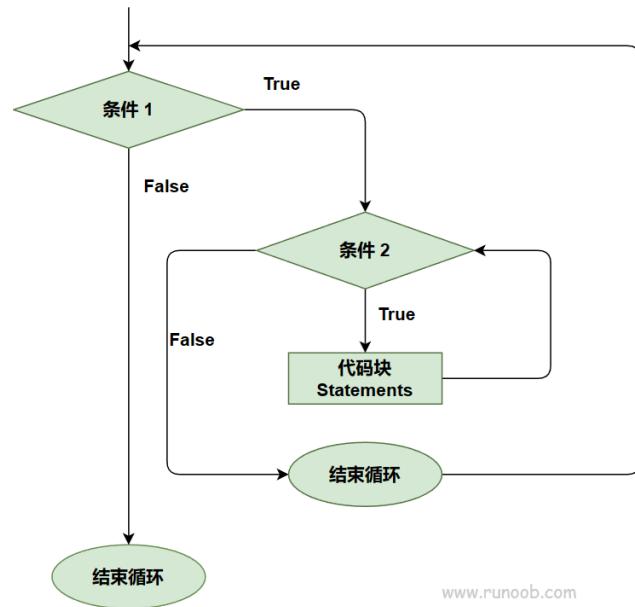
1 * 1 = 1	1 * 2 = 2	1 * 3 = 3	1 * 4 = 4	1 * 5 = 5	1 * 6 = 6	1 * 7 = 7	1 * 8 = 8	1 * 9 = 9
2 * 1 = 2	2 * 2 = 4	2 * 3 = 6	2 * 4 = 8	2 * 5 = 10	2 * 6 = 12	2 * 7 = 14	2 * 8 = 16	2 * 9 = 18
3 * 1 = 3	3 * 2 = 6	3 * 3 = 9	3 * 4 = 12	3 * 5 = 15	3 * 6 = 18	3 * 7 = 21	3 * 8 = 24	3 * 9 = 27
4 * 1 = 4	4 * 2 = 8	4 * 3 = 12	4 * 4 = 16	4 * 5 = 20	4 * 6 = 24	4 * 7 = 28	4 * 8 = 32	4 * 9 = 36
5 * 1 = 5	5 * 2 = 10	5 * 3 = 15	5 * 4 = 20	5 * 5 = 25	5 * 6 = 30	5 * 7 = 35	5 * 8 = 40	5 * 9 = 45
6 * 1 = 6	6 * 2 = 12	6 * 3 = 18	6 * 4 = 24	6 * 5 = 30	6 * 6 = 36	6 * 7 = 42	6 * 8 = 48	6 * 9 = 54
7 * 1 = 7	7 * 2 = 14	7 * 3 = 21	7 * 4 = 28	7 * 5 = 35	7 * 6 = 42	7 * 7 = 49	7 * 8 = 56	7 * 9 = 63
8 * 1 = 8	8 * 2 = 16	8 * 3 = 24	8 * 4 = 32	8 * 5 = 40	8 * 6 = 48	8 * 7 = 56	8 * 8 = 64	8 * 9 = 72
9 * 1 = 9	9 * 2 = 18	9 * 3 = 27	9 * 4 = 36	9 * 5 = 45	9 * 6 = 54	9 * 7 = 63	9 * 8 = 72	9 * 9 = 81

请按任意键继续... .

```
while (condition1)
{
    statement(s);
    while (condition2)
    {
        statement(s);
        ...
    }
}
```

```
    }  
    ... ... ...  
}
```

流程图



www.runoob.com

```
#include <stdio.h>  
int main()  
{  
    int i=1,j;  
    while (i <= 5)  
    {  
        j=1;  
        while (j <= i )  
        {  
            printf("%d ",j);  
            j++;  
        }  
        printf("\n");  
        i++;  
    }  
    return 0;  
}
```

A screenshot of a Windows command prompt window titled 'C:\Windows\system32\cmd.e'. The window displays the output of the provided C program, which prints the numbers 1 through 5 on separate lines, each preceded by its index. The terminal window includes standard window controls (minimize, maximize, close) and a status bar at the bottom.

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5  
请按任意键继续... . . .
```

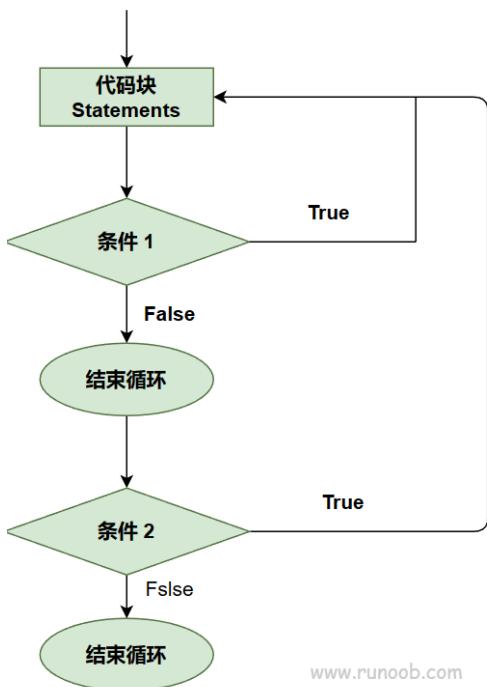
```
do
```

```

{
    statement(s);
    do
    {
        statement(s);
        ...
        ...
        }while (condition2);
        ...
        ...
    }while (condition1);

```

流程图



www.runoob.com

```

#include <stdio.h>
int main()
{
    int i = 1, j;
    do
    {
        j = 1;
        do
        {
            printf("*");
            j++;
        } while (j <= i);
        i++;
        printf("\n");
    } while (i <= 5);
    return 0;
}

```

```
{}

C:\Windows\system32\cmd.e + ~
*
**
***
****
*****
请按任意键继续. . .
```

6.13 关键概念

循环是一个强大的编程工具。在创建循环时，要特别注意以下 3 个方面：

- 注意循环的测试条件要能使循环结束；
- 确保循环测试中的值在首次使用之前已初始化；
- 确保循环在每次迭代都更新测试的值。

C 通过求值来处理测试条件，结果为 0 表示假，非 0 表示真。带关系运算符的表达式常用于循环测试，它们有些特殊。如果关系表达式为真，其值为 1；如果为假，其值为 0。这与新类型 _Bool 的值保持一致。

数组由相邻的内存位置组成，只储存相同类型的数据。记住，数组元素的编号从 0 开始，所有数组最后一个元素的下标一定比元素数目少 1。C 编译器不会检查数组下标值是否有效，自己要多留心。

使用函数涉及 3 个步骤：

- 通过函数原型声明函数；
- 在程序中通过函数调用使用函数；
- 定义函数。

函数原型是为了方便编译器查看程序中使用的函数是否正确，函数定义描述了函数如何工作。现代的编程习惯是把程序要素分为接口部分和实现部分，例如函数原型和函数定义。接口部分描述了如何使用一个特性，也就是函数原型所做的；实现部分描述了具体的行为，这正是函数定义所做的。

6.14 本章小结

本章的主题是程序控制。C 语言为实现结构化的程序提供了许多工具。`while` 语句和 `for` 语句提供了入口条件循环。`for` 语句特别适用于需要初始化和更新的循环。使用逗号运算符可以在 `for` 循环中初始化和更新多个变量。有些场合也需要使用出口条件循环，C 为此提供了 `do while` 语句。

典型的 `while` 循环设计的伪代码如下：

```
获得初值
while (值满足测试条件)
{
    处理该值
    获得下一个值
}
for 循环也可以完成相同的任务：
for (获得初值; 值满足测试条件; 获得下一个值)
    处理该值
```

这些循环都使用测试条件来判断是否继续执行下一次迭代。一般而言，如果对测试表达式求值为非 0，则继续执行循环；否则，结束循环。通常，测试条件都是关系表达式（由关系运算符和表达式构成）。表达式的关系为真，则表达式的值为 1；如果关系为假，则表达式的值为 0。C99 新增了 `_Bool` 类型，该类型的变量只能储存 1 或 0，分别表示真或假。

除了关系运算符，本章还介绍了其他的组合赋值运算符，如 `+=` 或 `*=`。这些运算符通过对其左侧运算对象执行算术运算来修改它的值。

接下来还简单地介绍了数组。声明数组时，方括号中的值指明了该数组的元素个数。数组的第一个元素编号为 0，第 2 个元素编号为 1，以此类推。例如，以下声明：

```
double hippos[20];
```

创建了一个有 20 个元素的数组 `hippos`，其元素从 `hippos[0]~hippos[19]`。利用循环可以很方便地操控数组的下标。

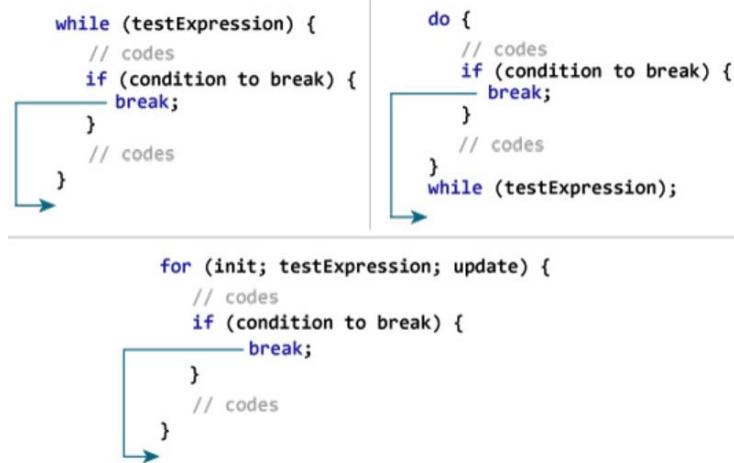
最后，本章演示了如何编写和使用带返回值的函数。

循环控制语句

break 语句

用法

- 当 `break` 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。（只能跳出一层循环）
- 它可用于终止 `switch` 语句中的一个 `case`。（没啥用）



```

#include <stdio.h>

int main()
{
    int a = 10;

    /* while 循环执行 */
    while (a < 20)
    {
        printf("a 的值: %d\n", a);
        a++;
        if (a > 15)
        {
            break;
        }
    }

    return 0;
}

```

正在执行任务: d:\VS_Code_Document\Bilibili_C_MicroFrank\circulate\bin\demo.exe

```

a 的值: 10
a 的值: 11
a 的值: 12
a 的值: 13
a 的值: 14
a 的值: 15
* 按任意键关闭终端。

```

continue 语句

3 种循环都可以使用 continue 语句。执行到该语句时，会跳过本次迭代的剩余部分，并开始下一轮迭代。如果 continue 语句在嵌套循环内，则只会影响包含该语句的内层循环。程序清单 7.9 中的简短程序演示了如何使用 continue。

如果用了 `continue` 没有简化代码反而让代码更复杂，就不要使用 `continue`。例如，考虑下面的程序段：

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

该循环跳过制表符，并在读到换行符时退出循环。以上代码这样表示更简洁：

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

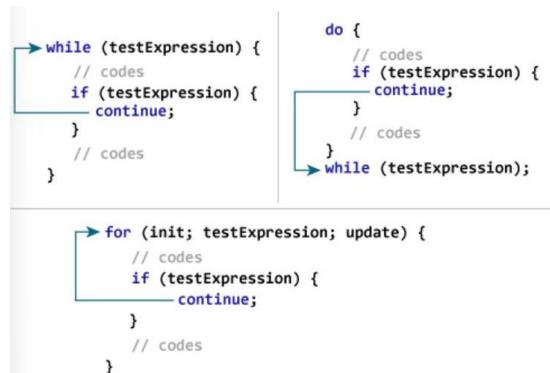
通常，在这种情况下，把 `if` 的测试条件的关系反过来便可避免使用 `continue`。

`continue` 会跳过当前循环中的代码，强迫开始下一次循环。

对于 `for` 循环，`continue` 语句执行后自增语句仍然会执行。

对于 `while` 和 `do...while` 循环，`continue` 语句重新执行条件判断语句。

(流程的不同!!!) C 语言——从上到下依次执行



```
1 #include <stdio.h>
2
3 int main ()
4 {
5     int a = 10;
6
7     do
8     {
9         if( a == 15)
10        {
11            /* 跳过迭代 */
12            a = a + 1; → 注意！！！
13            continue;
14        }
15        printf("a 的值: %d\n", a);
16        a++;
17
18    }while( a < 20 );
19
20    return 0;
21 }
```

```
* 正在执行任务: d:\VS_Code_Document\Bilibili_C_MicroFrank
\circulate\bin\demo.exe

a 的值: 10
a 的值: 11
a 的值: 12
a 的值: 13
a 的值: 14
a 的值: 16
a 的值: 17
a 的值: 18
a 的值: 19
* 按任意键关闭终端。
```

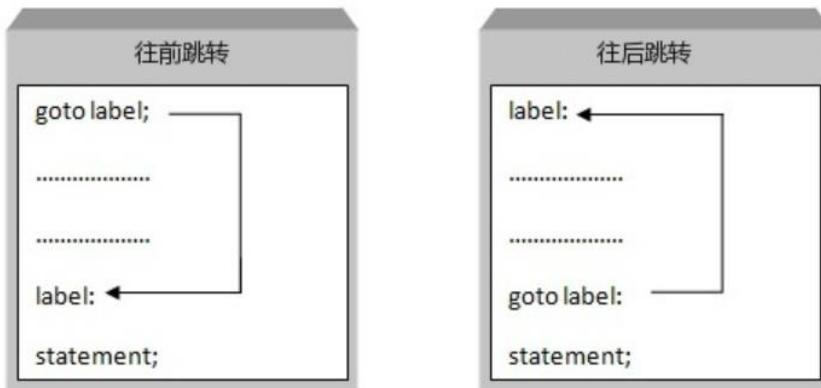
goto 语句

C 语言中的 `goto` 语句允许把控制无条件转移到同一函数内的被标记的语句。

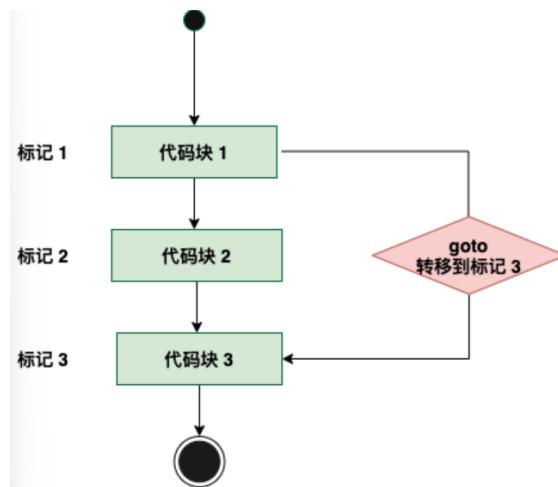
注意：在任何编程语言中，都不建议使用 `goto` 语句。因为它使得程序的控制流难以跟踪，使程序难以理解和难以修改。任何使用 `goto` 语句的程序可以改写成不需要使用 `goto` 语句的写法。

```
goto label;
...
.
label:
statement;
```

在这里，**label** 可以是任何除 C 关键字以外的纯文本，它可以设置在 C 程序中 **goto** 语句的前面或者后面。



流程图



```
#include <stdio.h>

int main()
{
    int a = 10;

LOOP:
    do
    {
        if (a == 15)
        {
            /* 跳过迭代 */
            a = a + 1;
            goto LOOP;
        }
        printf("a 的值: %d\n", a);
        a++;
    } while (a < 20);
```

```
    return 0;  
}
```

小结: 程序跳转

关键字: break、continue、goto

一般注解:

这 3 种语句都能使程序流从程序的一处跳转至另一处。

break 语句:

所有的循环和 switch 语句都可以使用 break 语句。它使程序控制跳出当前循环或 switch 语句的剩余部分，并继续执行跟在循环或 switch 后面的语句。

示例:

```
switch (number)  
{  
    case 4: printf("That's a good choice.\n");  
              break;  
    case 5: printf("That's a fair choice.\n");  
              break;  
    default: printf("That's a poor choice.\n");  
}
```

continue 语句:

所有的循环都可以使用 continue 语句，但是 switch 语句不行。continue 语句使程序控制跳出循环的剩余部分。对于 while 或 for 循环，程序执行到 continue 语句后会开始进入下一轮迭代。对于 do while 循环，对出口条件求值后，如有必要会进入下一轮迭代。

示例:

```
while ((ch = getchar()) != '\n')  
{  
    if (ch == ' ')  
        continue;  
    putchar(ch);  
    chcount++;  
}
```

以上程序段把用户输入的字符再次显示在屏幕上，并统计非空格字符。

goto 语句:

goto 语句使程序控制跳转至相应标签语句。冒号用于分隔标签和标签语句。标签名遵循变量命名规则。标签语句可以出现在 goto 的前面或后面。

形式:

```
goto label ;  
. . .  
label : statement
```

示例:

```
top : ch = getchar();  
. . .  
if (ch != 'y')  
    goto top;
```

7.9 关键概念

智能的一个方面是，根据情况做出相应的响应。所以，选择语句是开发具有智能行为程序的基础。C语言通过 if、if else 和 switch 语句，以及条件运算符 (?:) 可以实现智能选择。

if 和 if else 语句使用测试条件来判断执行哪些语句。所有非零值都被视为 true，零被视为 false。测试通常涉及关系表达式（比较两个值）、逻辑表达式（用逻辑运算符组合或更改其他表达式）。

要记住一个通用原则，如果要测试两个条件，应该使用逻辑运算符把两个完整的测试表达式组合起来。例如，下面这些是错误的：

```
if (a < x < z)           // 错误，没有使用逻辑运算符
...
if (ch != 'q' && != 'Q') // 错误，缺少完整的测试表达式
...

```

正确的方式是用逻辑运算符连接两个关系表达式：

```
if (a < x && x < z)           // 使用&&组合两个表达式
...
if (ch != 'q' && ch != 'Q') // 使用&&组合两个表达式
...

```

对比这两章和前几章的程序示例可以发现：使用第 6 章、第 7 章介绍的语句，可以写出功能更强大、更有趣的程序。

7.10 本章小结

本章介绍了很多内容，我们来总结一下。if 语句使用测试条件控制程序是否执行测试条件后面的一条简单语句或复合语句。如果测试表达式的值是非零值，则执行语句；如果测试表达式的值是零，则不执行语句。if else 语句可用于二选一的情况。如果测试条件是非零，则执行 else 前面的语句；如果测试表达式的值是零，则执行 else 后面的语句。在 else 后面使用另一个 if 语句形成 else if，可构造多选一的结构。

测试条件通常都是关系表达式，即用一个关系运算符（如，<或==）的表达式。使用 C 的逻辑运算符，可以把关系表达式组合成更复杂的测试条件。

在多数情况下，用条件运算符 (?:) 写成的表达式比 if else 语句更简洁。

ctype.h 系列的字符函数（如，issapce() 和 isalpha()）为创建以分类字符为基础的测试表达式提供了便捷的工具。

switch 语句可以在一系列以整数作为标签的语句中进行选择。如果紧跟在 switch 关键字后的测试条件的整数值与某标签匹配，程序就转至执行匹配的标签语句，然后在遇到 break 之前，继续执行标签语句后面的语句。

break、continue 和 goto 语句都是跳转语句，使程序流跳转至程序的另一处。break 语句使程序跳转至紧跟在包含 break 语句的循环或 switch 末尾的下一条语句。continue 语句使程序跳出当前循环的剩余部分，并开始下一轮迭代。

字符输入/输出和输入验证

函数

数组和指针

字符串和字符串函数