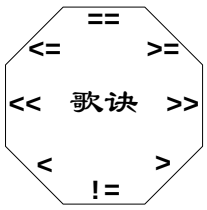
SECOND EDITION

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES



无极囫囵最为初，括号结构不离疏；

单符能有也能无，无极之后太极出；

两仪算子占首先，乘除余法源加减；

四相空间挪上下，化为左右合八卦；

八中余六中庸显，异同分明意境浅；

一零轮回二进制，位与异或分顺次；

二仪之末举逻辑，并且或者理明晰；

三目算符唯高妙，一事两面均虑到；

自作自受自繁难，舍己从人豁然宽；

从东往西逆流去，简符简判与简记；

有心至此余一符，诸君其中可找出？

此诀一出天下间，自始不见问谁先！

序

自 1978 年《C 程序设计语言》出版以来，计算机领域经历了一场革命。大型计算机已远大于从前，个人计算机的能力也足以媲美十年前的大型机。在此期间，C 语言也已悄然改变，并远远超越了其 UNIX 操作系统的编程语言这一初始角色。

C 语言的日益普及，加之这些年来的改变，以及一些非官方（组织开发的）编译器的出现，均需要一个比本书第一版更精确、更现代的 C 语言定义被提出。1983 年，美国国家标准协会（ANSI）成立了一个委员会，其目标是制定“一个无歧义的、与具体机器无关的 C 语言定义”，并保留 C 语言原有的精髓。这就是 C 语言 ANSI 标准的由来。

此标准规范了在本书第一版中提及但未进行描述的构成，特别是结构赋值和枚举。它提供了一种新的函数声明形式，允许对函数的定义与使用进行交叉检查。它详细规定了一个标准库，库中包含一个扩充函数集，完成输入输出、内存管理、字符串操作以及其他类似的任务。标准对原有定义中未清楚描述的特性的行为进行了精确定义，同时明确交代了语言在哪些方面仍旧依赖于具体机器。

《C 程序设计语言》的第二版描述了 ANSI 标准定义的 C 语言。尽管我们已摘录出语言的革新之处，但我们还是决定全部用新的形式进行撰写。对于大部分内容，这并未带来明显的差异；最显著的改变是新的函数声明及定义形式。现代编译器已经支持这一标准的大多数特性。

我们力求保持本书第一版的简洁性。C 不是一个大型语言，因而也不适合用一本大部头来介绍。我们改进了对关键特性的阐释，譬如 C 程序设计的核心——指针。我们还对原先的例子进行了提炼，并在一些章节中增加了新的例子。例如，复杂声明的处理部分增加了将声明与对应文字描述进行相互转换的函数。如前一版本，所有的例子其文本都为机器可读格式，已直接进行过测试。

附录 A 是参考手册而不是标准本身，但我们试图以较小篇幅呈现出标准的精华内容。它的目的是让编程者易于理解，而不是供编译器实现者参考的定义——那正是标准应当承担的角色。附录 B 是对标准库中功能函数的总结，其同样针对编程者，而非实现者。附录 C 是对之前版本所作变更的小结。

就如我们在第一版序中所说，C 语言“随着经验的积累其使用会愈加得心应手”。经过十几年的体验，我们仍然这么认为。希望本书能帮助你学好并用好 C 语言。

深深感谢那些帮助我们完成本书（第二版）的朋友们。Jon Bentley、Doug Gwyn、Doug McIlroy、Peter Nelson 和 Rob Pike 对本书手稿的几乎每页都给出了透彻的评注。我们非常感谢 Al Aho、Dennis Allison、Joe Campbell、G.R.Emlin、Karen Fortgang、Allen Holub、Andrew Hume、Dave Kristol、John Linderman、Dave Prosser、Gene Spafford 和 Chris Van Wyk，他们仔细地阅读了本书。我们也收到了来自 Bill Cheswick、Mark Kernighan、Andy Koenig、Robin Lake、Tom London、Jim Reeds、Clovis Tondo 和 Peter Weinberger 的有益建议。Dave Prosser 解答了许多关于 ANSI 标准的细节问题。我们大量地使用了 Bjarne Stroustrup 的 C++翻译程序来进行程序的局部测试。Dave Kristol 为我们提供了一个 ANSI C 编译器进行最终测试。Rich Drechsler 协助进行了大量的排版工作。

诚挚地感谢每个人。

Brian W. Kernighan
Dennis M. Ritchie

引言

C 语言是一种通用的程序设计语言。它同 UNIX 系统之间具有非常密切的联系——C 语言是在 UNIX 系统上开发的，并且，无论是 UNIX 系统本身还是其上运行的大部分程序，都是用 C 语言编写的。但是，C 语言并不受限于任何一种操作系统或机器。由于它很适合用来编写编译器和操作系统，因此被称为“系统编程语言”，但它同样适合于编写不同领域中的大多数程序。

C 语言的很多重要概念来源于由 Martin Richards 开发的 BCPL 语言。BCPL 对 C 语言的影响间接地来自于 B 语言，它是 Ken Thompson 为第一个 UNIX 系统而于 1970 年在 DECPDP-7 计算机上开发的。

BCPL 和 B 语言都是“无类型”的语言。与之相对，C 语言提供了很多数据类型。其基本类型包括字符、具有多种长度的整型和浮点数等。另外，还有通过指针、数组、结构和联合派生的各种数据类型。表达式由运算符和操作数组成。任何一个表达式，包括赋值表达式或函数调用表达式，都可以是一个语句。指针提供了与具体机器无关的地址算术运算。

C 语言为实现结构良好的程序提供了基本的控制流结构：语句组、条件判断(if—else)、多路选择(switch)、终止测试在顶部的循环(while、for)、终止测试在底部的循环(do)、提前跳出循环(break)等。

函数可以返回基本类型、结构、联合或指针类型的值。任何函数都可以递归调用。局部变量通常是“自动的”， 即在每次函数调用时重新创建。函数定义可以不列放在一起，而变量可以按块结构的形式进行声明。一个 C 语言程序的不同函数可以出现在多个单独编译的不同源文件中，变量可以仅在函数内可见，或可位于函数外且仅在单个源文件中可见，也可对于整个程序都可见。

编译的预处理阶段将对程序文本进行宏替换、包含其他源文件以及进行条件编译。

C 语言是一种相对“低级”的语言。这种说法并没有什么贬义，它仅仅意味着 C 语言可以处理大部分计算机能够处理的对象，比如字符、数字和地址。这些对象可以通过具体机器实现的算术运算符和逻辑运算符进行组合和移动。

C 语言没有提供直接处理诸如字符串、集合、列表或数组等复合对象的操作。尽管可以将结构作为整体单元进行拷贝，但语言中没有处理整个数组或字符串的操作。除了静态定义和运用于函数局部变量的栈规则之外，C 语言没有定义任何存储器分配工具，也不提供堆和对无用内存的回收。最后，C 语言本身没有提供输入 / 输出功能，没有 READ 或 WRITE 语句，也没有内置的文件访问方法。所有这些高层的机制必须由显式调用的函数提供。大多数 C 语言的具体实现中都已包括了一个适当的（这些函数的）标准集合。

类似地，C 语言只提供简单的单线程控制流，即测试、循环、分组和子程序，它不提供多道程序设计、并行操作、同步和协同例程。

尽管缺少其中的某些特性看起来像是严重的缺陷（“你的意思是必须通过调用函数来比较两个字符串？”），但是将语言保持在一个适度的规模有其实际的益处。 由于 C 语言相对较小，因此能用较小的篇幅来描述，也能很快学会。程序员有理由期望能了解、理解并真正日常使用整个语言。

多年以来，C 语言的定义就是《The C Programming Language》第 1 版中的参考手册。1983 年，美国国家标准协会（ANSI）成立了一个委员会以制定一个现代的、全面的 C 语言定义。最终的定义——ANSI 标准，即“ANSI C”，于 1988 年后期完成。该标准的大部分特性已被现代

编译器所支持。

此标准基于原有的参考手册制定。语言本身只做了相对较小的改动；此标准的目的之一就是确保现有的程序仍然有效，或者当无效时，编译器能对新的行为产生警告信息。

对大部分程序员来说，最重要的变化是新的函数声明和函数定义的语法。现在，函数声明中可以包含函数参数的描述信息；函数定义语法也有与之相匹配的改变。这些附加的信息使得编译器在检测因参数不匹配而导致的错误时容易了许多。根据我们的经验，这一扩充对于语言非常有用。

新标准对语言还做了一些小范围的改进。现在，已得到广泛支持的结构赋值和枚举成为了语言的正式部分；浮点运算可以以单精度进行计算；算术运算，特别是无符号类型的运算特性得到了明确定义；对预处理器的描述更加详尽。这些改进于对大部分程序员的影响大都比较小。

此标准的第二个重要贡献是为 C 语言定义了一个函数库。它描述了诸如访问操作系统（如读写文件）、格式化输入/输出、内存分配和字符串操作、以及其他类似的函数。此标准还定义了一系列标准头文件，它们为访问函数声明和数据类型声明提供了统一方法。程序使用这一函数库与宿主系统交互能够确保具有兼容的行为。此函数库的大部分内容高度效仿了 UNIX 系统的“标准 I/O 库”。此函数库已在本书的第 1 版中进行了描述，并业已在其他一些系统中广泛使用。同样，大多数程序员也不会感到这部分有太大变化。

由于大多数计算机本身直接支持 C 语言提供的数据类型和控制结构，因此只需要一个很小的运行时库就可以实现自包含程序。由于标准库中的函数只会被显式地调用，因此不需要的库函数即可被省略掉。大部分库函数能用 C 语言编写，而且除了其中隐藏的操作系统细节之外都可移植。

尽管 C 语言能够运行在许多计算机上，但它不依赖任何特定的硬件系统架构。只要稍加注意就可以编写出可移植的程序，即程序可以不加修改地运行在多种硬件上。ANSI 标准明确地提出了可移植性问题，并预设了一个常量的集合，用于描述运行程序的机器的特性。

C 语言不是一种强类型的语言，但随着它的发展，其类型检查机制已经得到了加强。尽管 C 语言的最初定义不赞成指针和整型变量相互转换，但并没有禁止；这种做法已经淘汰很长时间了。现在，ANSI 标准要求对变量进行恰当的声明和显式的类型转换，一些优秀的编译器已经强制要求那样做。新的函数声明方式则是朝此方向迈进的另一步。编译器将对大部分的数据类型错误发出警告，并且不自动执行不兼容数据类型之间的类型转换。不过，C 语言保持了其初始的设计思想，即程序员了解他们自己在做什么，它只是要求程序员将自己的意图明确地表达出来。

同任何其他语言一样，C 语言也有瑕疵。某些运算符的优先级不正确；语法的某些部分还能进一步优化。尽管如此，C 语言已被证实是一种可广泛用于多种程序设计应用的相当高效的、表达能力极强的语言。

本书按照以下结构编排：第 1 章将对 C 语言的核心部分进行简要介绍。其目的是让读者能尽快开始编写 C 语言程序，因为我们深信，实际编写程序才是学习一种新语言的好方法。这部分内容的介绍假定读者对程序设计的基本元素有一定的了解。我们在这部分内容中没有解释计算机、编译等概念，也没有解释诸如 n=n+1 这样的表达式。我们将尽量在合适的地方介绍一些实用的程序设计技术，但是，本书的中心目的并不是介绍数据结构和算法。在篇幅受限的情况下，我们将专注于讲解语言本身。

第 2 章到第 6 章将更详细地讨论 C 语言的各个方面，且比第 1 章要正式得多，但其依然强调完整的程序例子，而不是孤立的程序片段。第 2 章介绍基本的数据类型、运算符和表达式。第 3 章介绍控制流，如 if-else、switch、while 和 for 等。第 4 章介绍函数和程序结构——外部变

量、作用域规则和多源文件等等内容，同时还会涉及到一些预处理器的知识。第 5 章介绍指针和地址运算。第 6 章介绍结构和联合。

第 7 章介绍标准库。标准库提供了一个与操作系统交互的公共接口。这个函数库是由 ANSI 标准定义的，这就意味着所有支持 C 语言的机器都会支持它，因此，使用这个库执行输入、输出或其他访问操作系统的操作的程序可以不加修改地运行在不同机器上。

第 8 章介绍 C 语言程序和 UNIX 操作系统之间的接口，我们将把重点放在输入/输出、文件系统和存储分配上。尽管本章中的某些内容是针对 UNIX 系统所写的，但是使用其他系统的程序员仍然能从中获益，比如深入了解如何实现标准库以及有关可移植性方面的一些建议。

附录 A 是一个语言参考手册。虽然 C 语言的语法和语义的官方正式定义是 ANSI 标准本身，但是，ANSI 标准的文档首先是写给编译器的编写者看的，因此，对程序员来说不一定最合适。本书中的参考手册采用了一种不很严格的形式，更简洁地对 C 语言的定义进行了介绍。附录 B 是对标准库的一个总结，它同样是为程序员而非编译器实现者准备的。附录 C 是相对于 C 语言最初版本所做的变更的一个简要小结。但是，如果有不一致或疑问的地方，标准本身和各个特定的编译器则是解释语言的最终权威。本书的最后是索引部分。

第1章 入门介绍

本章首先对 C 语言做一个简要介绍。我们希望通过实际的程序来展示 C 语言的核心元素，而不在细节、规则以及例外情况上过多纠缠。出于这种考虑，本章并不力求内容的完整性甚至是精确性（但是所给的例子肯定是正确的）。我们希望使读者能尽快学会编写有用的程序。为此，本章着重介绍基础部分：变量与常量、算术运算、控制流、函数和基本输入/输出。我们有意省略了 C 语言中编写大型程序所需要的重要特性，包括指针、结构、C 语言丰富的运算符集中的大多数运算符、部分控制流语句以及标准库。

这一方法有其不足之处，最明显的是对任何具体语言特性的介绍都不完全，且可能会因为教程简略而对读者造成误导。所举例子也因没有充分发挥 C 语言的能力而不是足够的简洁和优美。我们已经尽力减小这些负面影响，但读者仍应注意。此方法的另一缺点是后续章节必须要重复本章的一些内容。期望这些重复带给读者的帮助会胜过烦扰。

无论如何，经验丰富的程序员应能从本章的材料中推知自己在程序设计中所需要的东西，初学者则应编写类似的小程序来增补本章所学。两种程度的读者都可以以本章为框架来学习后续章节的详细内容。

1.1 开始

学习一种新语言的唯一方法是用它编写程序。无论是哪种语言，第一个要学习编写的程序都一样：

```
打印文字
    hello, world
```

这是一个巨大的障碍，要逾越它你必须学会创建程序文本，成功编译程序，能够加载、运行程序，并能查看程序输出。在这些操作细节掌握之后，其余的事情就相对简单了。

C 语言中，打印“hello, world”的程序如下：

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

如何运行这个程序取决于所使用的系统。例如，在 UNIX 系统中，你需要创建一个名字以“.c”结尾的文件来存放这个程序，如 hello.c，然后使用下面的命令对其进行编译：

```
cc hello.c
```

此过程如果没有任何改动（比如遗漏了某个字符或是拼错了什么），那么编译将会安静地进行，最后生成一个名为 a.out 的可执行文件。如果输入指令

```
a.out
```

运行该文件，它将打印出

```
hello, world
```

在其他系统中，相应规则可能会有所差别，请向特定系统的专家请教。

下面就程序本身作一些解释。一个 C 语言程序无论其大小如何，均由**函数**和**变量**构成。函

数包含若干**语句**，语句指定了所要完成的计算操作；变量则用于存放计算过程中使用的值。C 语言的函数类似于 Fortran 语言的子程序和函数或者是 Pascal 语言的过程和函数。本例中，函数的名字为 main。通常函数名可由编程者任意指定，但是“main”这个名字比较特殊——每个程序都要从 main 函数的起始处开始执行。这意味着每个程序都必须在某处包含一个 main 函数。

第一个 C 程序

```
#include <stdio.h>                          包含标准库的相关信息

main()                                      定义名字为 main 的函数
                                            函数不接收任何参数
{                                           main 的所有语句被包含在一对花括号内
    printf("hello, world\n");               main 函数调用了库函数 printf
}                                           来打印这串字符
                                            \n 代表换行符
```

main 函数通常会调用其它函数来协助完成程序工作，这些函数中一些由编程者自行编写，而另一些则来自于函数库。程序的第一行

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

告诉编译器在本程序中包含标准输入输出库的相关信息；许多 C 程序源文件的起始处都包括这一行。标准库在第 7 章和附录 B 中进行介绍。

在函数之间交互数据的一种方法是由调用函数向被调函数提供一列数值，这些数值称为**参数**。参数列表位于函数名之后，并括上一对圆括号。在本例中，main 被定义成一个不需要参数的函数，这通过空参数表 () 来表示。

一个函数的全部语句括在一对花括号{}之内。本例中 main 函数只包含一条语句，

```
printf("hello, world\n");
```

函数的调用形式为函数名加上（由一对圆括号括上的）参数列表。上述语句即调用了函数 printf，且其参数为"hello, world\n"。printf 是一个库函数，用于打印输出，此处的输出内容即引号内的这串字符。

用双引号引上的字符序列，如"hello, world\n"，称为**字符串**或者是**字符串常量**。现阶段我们仅将字符串作为 printf 以及其它函数的参数使用。

字符串中的序列 \n 在 C 语言中表示**换行符**，在打印时，它表示从下一行的最左边继续打印。如果去掉它（值得一试☺），你会发现在输出打印完毕后并没有另起一行。要在 printf 的参数中包含换行符，必须用 \n；如果你试图用类似下面这样的代码来达到目的，

```
printf("hello,world
");
```

C 编译器将会产生一条错误信息。

printf 不会自动添加换行符，因而可以使用多次调用分几步来构成一行输出。我们的第一个程序也可以写成：

```
#include<stdio.h>

main()
{
    printf("hello, ");
    printf("world");
```

```
    printf("\n");
}
```

其产生的输出完全相同。

注意 \n 表示的仅为单个字符，它是一个**转义序列**。像 \n 之类的转义序列提供了一种通用的可扩展的机制，用于表示无法打印或是不可见的字符。C 语言提供的其他一些转义序列包括表示制表符的 \t 、表示回退符的 \b 、表示双引号的 \" 以及表示反斜杠自身的 \\ 等。2.3 节给出了转义序列的完整列表。

练习 1-1. 请读者在自己的系统上运行“hello, world”程序。尝试删除程序中的某一部分，看看会报告什么错误信息。

练习 1-2. 在 printf 函数的参数字符串中包含 *c* （*c* 为某个先前未曾列出的字符，可变），请试验并找出会产生的情况。

1.2 变量与算术表达式

接下来的这个程序使用公式 °C = (5/9)(°F-32)打印下面的华氏温度至摄氏温度的对应表：

```
1   -17
20  -6
40  4
60  15
80  26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
240 115
260 126
280 137
300 148
```

这个程序依然只包含一个名字为 main 的函数。它比之前那个打印“hello, world”的程序要长一些，但并不复杂。它引入了一些新的概念，包括注释、声明、变量、算术表达式、循环以及格式化输出。

```
#include <stdio.h>

/* 按 fahr = 0, 20, ..., 300
    打印华氏温度-摄氏温度对应表 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;        /* 温度表的下限 */
    upper = 300;      /* 温度表的上限 */
    step = 20;        /* 温度增长步幅 */
```

```
        fahr = lower;
        while (fahr <= upper) {
            celsius = 5 * (fahr-32) / 9;
            printf("%d\t%d\n", fahr, celsius);
            fahr = fahr + step;
        }
    }
```

其中的两行

```
    /* 按 fahr = 0, 20, ..., 300
       打印华氏温度-摄氏温度对应表 */
```

是程序**注释**，此例中的注释简要地说明了这个程序的功能。在 /* 和 */ 之间的任何字符串都会被编译器忽略；注释可以自由地运用在程序中，以使程序更加易于理解。在空格、制表符和换行符能够出现的地方都可以使用注释。

在 C 语言中，所有变量在使用之前都必须声明，变量声明一般放在函数的所有执行语句之前。**声明**用于说明变量的属性，它包括一个类型名和一列变量，如：

```
    int fahr, celsius;
    int lower, upper, step;
```

类型 int 表明其后所列的变量都是整数；相应地，类型 float 表明其后列出的变量都是浮点数(浮点数可能有小数部分)。int 和 float 类型的变量取值范围依赖于所使用的机器。一般而言，int 是 16 位的，取值范围在-32768 到+32767 之间，也有 32 位的 int。典型的 float 数是 32 位的，至少有 6 位有效数字，数量级一般在 10^{-38} 到 10^{38} 之间。

除 int 和 float 之外，C 还提供了其他一些基本数据类型，包括：

```
    char        字符——单个字节
    short       短整型
    long        长整型
    double      双精度浮点类型
```

这些数据类型对象的大小同样依赖于具体机器。此外，还存在由这些基本数据类型组成的**数组**、**结构**和**联合**，以及指向这些类型的**指针**和返回这些类型的**函数**，我们会在后续相应的章节中遇到它们。

该温度转换程序的计算部分的开头是**赋值语句**

```
    lower = 0;
    upper = 300;
    step  = 20;
    fahr  = lower;
```

它们为变量设置了初始值。每条赋值语句都以分号结尾。

由于对照表中每一行的计算方式都相同，我们用一个重复过程来产生每行的输出；这正是 while 循环的目的：

```
    while (fahr <= upper) {
        ...
    }
```

while 循环的执行顺序如下：首先，圆括号内的条件被检测。如果条件为真（fahr 小于或等于 upper），则执行循环体（花括号内的三条语句）。然后，条件被再次测试，如果仍然为真，则循环体被再次执行。直到当条件不成立时(fahr 超过 upper)，循环结束，继续执行紧接在循环语句之后的语句。在这个程序中，循环语句之后没有其它语句，于是程序终止执行。

while 的循环体可以是由花括号括起来的一条或多条语句（就像温度转换程序中的那样），

或者是一条单独的没有花括号的语句，例如：

```
while (i < j)
    i = 2 * i;
```

在这两种情况下，我们都会将由 while 循环控制的语句缩进一个制表位(在本书之中用 4 个空格表示)，这样读者一眼就能看出哪些语句包含在循环体之内。缩进增强了程序的逻辑结构。尽管 C 语言编译器并不关心程序看上去是什么样子，但是恰当的缩进和分隔对于提高程序的可读性至关重要。我们推荐每行只写一条语句，并且在运算符的两边都添加空格以使运算组合显得更加清楚。花括号的位置不那么重要，尽管人们都有各自喜欢的风格。我们从几种流行的风格中选取了一种，你可以选择一种适合自己的风格，并一直使用它。

在上面的程序中，循环体完成了绝大部分的工作。循环体内的语句

```
celsius = 5 * (fahr-32) / 9;
```

计算出对应的摄氏温度值并赋给变量 celsius。在该表达式中，之所以写成先乘以 5 然后再除以 9 而不是直接乘以 5/9，是因为 C 语言像许多其他的语言一样，其整数除法会进行**截取**处理，即结果中的小数部分会被丢弃掉。由于 5 和 9 是整数，5 除以 9 的结果会被截取为 0，于是所有求出的摄氏温度都会变成 0。

这个例子也进一步展示了 printf 函数的工作方式。printf 是一个通用的格式化输出函数，我们会在第 7 章对它进行详细介绍。它的第一个参数是要打印的字符串，其中每个百分号（%）指明了后续每个（第 2 个、第 3 个、…）参数替换到字符串中的位置，以及参数打印的格式。例如，%d 指定的是一个整型参数，于是下面的语句

```
printf("%d\t%d\n", fahr, celsius);
```

会将整数类型变量 fahr 和 celsius 的值打印出来，并且中间有一个制表符（\t）。

printf 函数的第 1 个参数中的各个“百分号结构”依次与第 2 个参数、第 3 个参数、... 相对应；它们的个数和类型必须匹配，否则将得到错误的输出结果。

顺便指出，printf 函数并不是 C 语言的一部分。C 语言本身并没有定义输入输出功能。printf 函数只是标准库中一个有用的函数，标准库中的函数通常都可以被 C 程序调用。而 printf 函数的行为已在 ANSI 标准中定义，因此，它的特性在所有遵循该标准的编译器和库中都应该一样。

为把着重点放在 C 语言本身，在第 7 章之前我们不会过多地讨论输入和输出，我们特别将格式化输入推后到第 7 章讨论。如果你需要输入数字，请参考 7.4 节中对 scanf 函数的讨论。scanf 函数很像 printf 函数，不过它的功能是读取输入而不是打印输出。

上面的温度转换程序还有一些问题。其中比较简单的一个是程序的输出不是太好看，因为输出的数字没有进行右对齐。这个问题好办，如果为 printf 语句中的每个%d 增加一个宽度值，那么打印出来的数字就会在这个宽度的区域内进行右对齐。比如，我们可能通过编写

```
printf("%3d %6d\n", fahr, celsius);
```

来打印每一行，其中第一个数占 3 个数字宽度，第二个数占 6 个数字宽度，其效果如下：

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
...
```

另一个严重一些的问题是，由于我们使用了整数算术运算，所得到的摄氏温度并不十分准

确。比如华氏 0° 对应的精确的摄氏温度是-17.8°，而不仅仅是-17°。为了得到更加精确的结果，我们应该采用浮点数运算来替代整数运算。这需要对程序做几处改动，下面是改动之后的版本：

```
#include <stdio.h>

/* 对 fahr = 0, 20, ..., 300; 打印华氏
   温度到摄氏温度的对应表。——浮点数版本 */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;        /* 温度表的下限 */
    upper = 300;      /* 温度表的上限 */
    step = 20;        /* 温度增长步幅 */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

这个版本和原先的版本大致相同，只是 fahr 和 celsius 被声明为浮点数类型，并且转换公式也写得更加自然。在之前的版本中我们不能使用 5/9，因为它会被整数除法截取为 0。这里的 5.0/9.0，由于常数中的小数点指明了其为浮点型的数，因此它表示两个浮点数的比值，因而不会被截取。

如果一个算术运算符的操作数都是整数，那么将会执行整数运算。但如果是一个浮点操作数和一个整数操作数，那么该整数首先会被转换成浮点数，然后再进行浮点运算。如果我们写成 fahr - 32，那么 32 会被自动转换为浮点类型。尽管如此，对于浮点型的常量，即便它的值是整数，显式地写成带小数点的形式可以明确它浮点类型的本质，以方便阅读者理解。

在第 2 章中，我们会详细地描述整数转化为浮点数的规则。这里只需要注意程序中的赋值语句

```
fahr = lower;
```

和测试语句

```
while (fahr <= upper)
```

也具有同样的工作原理，int 类型的数据会首先被转换成 float 类型的数据，然后再进行对应的操作。

printf 函数的转换格式说明%3.0f 表明该浮点数(即 fahr)在打印时至少要占 3 个字符的宽度，且不带小数点和小数部分。%6.1f 则表明另一个数(celsius) 在打印时至少要占 6 个字符的宽度，且在小数点后面要有 1 位小数。输出的结果类似：

```
  0  -17.8
 20   -6.7
 40    4.4
...
```

转换格式说明中的数据宽度或精度可以省略：%6f 表明这个数至少要占 6 个字符的宽度；%.2f 表明在小数点后面必须要有两个字符，但在宽度上没有限制；而%f 仅仅说明这个数要按照浮点类型来打印。

%d	按照十进制整数格式打印
%6d	按照十进制整数格式打印，至少占据 6 个字符宽度
%f	按照浮点数格式打印
%6f	按照浮点数类型打印，至少占据 6 个字符宽度
%.2f	按照浮点数类型打印，小数点后打印 2 位数字
%6.2f	按照浮点数类型打印，至少占据 6 个字符的宽度，小数点后打印 2 位数字

此外，printf 函数还支持下列格式说明：%o 表示八进制数，%x 表示十六进制数，%c 表示字符，%s 表示字符串，%%则表示百分号%本身。

练习 1-3．修改温度转换程序，在输出的转换表前加一个标题。

练习 1-4．编写一个程序，打印出摄氏温度到华氏温度的对照表。

1.3 for 语句

完成某一特定任务的程序会有很多种不同的实现方式。下面我们就将温度转换程序变化一下：

```
#include <stdio.h>

/* 打印华氏-摄氏温度对照表 */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

这个程序的执行结果与之前的程序相同，不过它看上去的确不一样。一个主要的改变是除 fahr 之外其它的变量都去掉了，fahr 也变成了 int 类型。温度的上限值、下限值和步长均以常量的形式出现在 for 语句（对读者而言这是一个新的语言成分）中，并且计算摄氏温度的表达式作为第 3 个参数出现在 printf 语句中，不再是一条独立的赋值语句。

最后这个改变体现了一个通用的规则：在任何可以使用某个类型的值的地方，都可以使用计算结果为该类型的更复杂的表达式。由于 printf 语句的第 3 个参数必须是与%6.1f 匹配的浮点数，因此任何浮点类型的表达式都可出现在这里。

for 语句是一个循环语句，是 while 语句的一般化形式。如果将它与之前的 while 相比较，你会觉得它的操作更加清楚。圆括号里包含以分号分隔开的三部分。第一部分，初始化语句

```
fahr = 0
```

会在正式进入循环之前执行一次。第 2 部分是用于控制循环的测试语句，或称条件语句

```
fahr <= 300
```

这个条件语句会被检测，如果成立，则循环体(这里就一条 printf 语句)被执行。然后，执行第三部分递增语句

```
fahr = fahr + 20
```

之后，条件语句被再次测试。直到条件语句不再成立，循环才会结束。与 while 语句一样，for

语句的循环体可以是一条单独的语句，也可以是由花括号括起来的一组语句。其初始化语句、条件语句和递增语句可以是任意的表达式。

在程序中选用 while 语句还是 for 语句是随便的，主要看使用哪一种更加清晰。for 语句通常适合于循环程序的初始化语句和递增语句都是单条语句且彼此在逻辑上有关联的情形，因为它把控制循环的语句都集中到了一起，比 while 语句更加紧凑。

练习 1-5. 修改温度转换程序，以相反的顺序打印温度对应表，即从 300°F 到 0°F。

1.4 符号常量

在正式结束对温度转换程序的讨论之前，我们再来看一下符号常量。把诸如 300、20 这样的“幻数”插进程序中可不是一种好的做法，对于今后需要阅读这段代码的人而言，它们几乎没有传达任何信息，并且难于进行系统化的修改。解决幻数问题的一种方法是给它们设定有意义的名字。#define 语句将一个**符号名字**或是**符号常量**定义为一串特定的字符：

```
#define  名字  替换文本
```

在定义之后，该名字出现的任何位置（只要不是在引号中或者是其它名字的一部分），都会被替换为对应的替换文本。符号名字的形式和变量名相同：都是以字母开头的字母和数字序列。而替换文本则可以是任意的字符序列，字符数量也没有限制。

```
#include <stdio.h>

#define LOWER  0     /* 温度表的下限 */
#define UPPER  300   /* 温度表的上限 */
#define STEP   20    /* 步长 */

/* 打印华氏-摄氏温度对照表 */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

程序中的 LOWER、UPPER 和 STEP 是符号常量，不是变量，因此它们不用进行声明。按惯例符号常量都用大写字母来书写，这样可以很容易地与小写字母书写的变量名相区别。需要注意在 #define 语句的结束位置没有分号。

1.5 字符输入和输出

接下来我们将要讨论一组彼此关联的字符数据处理程序。你会发现，许多程序不过是这里所讨论的原型程序的扩展而已。

标准库所支持的输入输出模型非常简单。无论输入自何处或者输出到哪里，输入或输出的文本都被视作字符流。**文本流**是一个被划分成多行的字符序列，每一行包含零到多个字符，行末带有一个换行符。标准库有责任让所有的输入流和输出流都符合这一模型。使用标准库的 C 程序员不用关心字符行在程序之外是如何表示的。

标准库提供了多个函数来每次读取或写出一个字符，其中最简单的函数莫过于 getchar 和 putchar。每次调用时，getchar 函数从文本流中读取下一个输入字符并返回它的值。这就是说，执行

```
c = getchar()
```

之后，变量 c 中就保存了下一个输入字符。这些字符通常是从键盘输入的；我们会在第 7 章中讨论从文件中输入字符的方法。

函数 putchar 在每次调用时打印一个字符：

```
putchar(c)
```

会将整数变量 c 中的值按字符类型打印出来，通常会显示在屏幕上。函数 putchar 和 printf 可以交替调用，输出次序和调用的次序一致。

1.5.1 文件拷贝

有了 getchar 和 putchar，在不了解更多关于输入输出知识的情况下，就能够编写出数量惊人的有用代码。最简单的程序例子是将输入字符一次一个地拷贝到它的输出：

```
读入一个字符
while (该字符不是文件结束符)
    输出刚读入的字符
    再读入一个字符
```

将上述文字转换成 C 程序就是：

```
#include <stdio.h>

/* 将输入拷贝到输出；版本 1 */
main()
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

其中，关系运算符 != 表示“不等于”。

无论字符在键盘和屏幕上是什么形态，它与其他任何东西一样在计算机内部都是以位模式存储的。数据类型 char 专门用于存放这些字符数据，不过任何整数类型同样可以存放它们。我们在这里使用 int 类型是由于一个微妙但却很重要的原因。

这个原因就是需要将表示输入结束的信息与正常的输入数据区分开来。解决办法是在没有新的输入之后，函数 getchar 返回一个与众不同的值，这个值不会和任何实际的字符相混淆。这个值称为 EOF，代表“end of file”。我们必须将 c 声明为足够大的数据类型，以便能够存放 getchar 返回的任何数据。c 不能使用 char 类型，因为除了存放任何可能的 char 型数据之外，它还必须能存放 EOF。因此我们使用了 int 类型。

EOF 是定义在 <stdio.h> 中的一个整数，它的值具体是多少并不重要，只要不同于任何字符的值即可。通过使用符号常量，我们可以确保程序中任何部分都不依赖于具体的数值。

有经验的 C 程序员可以把这个拷贝程序写得更精练一些。C 语言中任何赋值语句，比如

```
c = getchar();
```

都是一个表达式，并且有自身的值。它的值就是赋值之后等号左边变量的值。这就意味着，赋值语句可以作为更大的表达式的一部分出现。如果将这个 c 的赋值语句放到 while 循环的测试部分中去，那么程序就可以这样写：

```
#include <stdio.h>

/* 将输入拷贝到输出；版本 2 */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

while 语句取得一个字符，将它赋予变量 c，然后检测这个字符是不是结束符 EOF。如果不是，则执行循环体打印这个字符，然后继续循环。当达到输入末尾时，while 循环终止，main 也随之终止。

这个版本将输入操作集中起来（只引用了一次 getchar 函数），缩短了整个程序。这使得程序更加紧凑，并且读者一旦掌握了这种惯用法之后它会更容易阅读。你会经常见到这种风格。（但是，过分使用这种风格可能会使代码变得难于理解，因此本书会对此加以控制）。

在条件语句中，赋值操作两边的圆括号必不可少。由于 != 的**优先级**高于 = ，这意味着如果没有圆括号，比较测试 != 会在赋值操作 = 之前完成。因此语句

```
c = getchar() != EOF
```

等价于

```
c = (getchar() != EOF)
```

其结果就是根据 getchar 的调用是否遇到文件结束而将变量 c 设置为 0 或 1，这并不是我们希望达到的效果。（关于运算优先级的更多内容在第 2 章中介绍）

练习 1-6. 验证表达式 getchar() != EOF 的值是 0 还是 1.

练习 1-7. 写一个程序，将 EOF 的值打印出来。

1.5.2 字符计数

接下来的这个程序用于统计字符的个数，它与前面的拷贝程序类似。

```
#include <stdio.h>

/* 统计输入中字符的个数；版本 1 */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

其中，语句

```
++nc;
```

展示了一个新的运算符 ++，它表示将其作用的变量增 1。你可以用 nc = nc+1 替代它，但 ++nc 更加简洁，效率常常也更高。同样，还存在一个与之对应的运算符 --，它表示减 1。运算符 ++ 和 -- 既可以是前缀运算符（++nc），也可以是后缀运算符（nc++）；这两种形式在表达式中有着不同的取值，但它们都将 nc 增加了 1，这些特性将会在第 2 章中介绍。现在，我们只采用其中的前缀形式。

此字符统计程序使用了一个 long 型变量保存计数值，而没有用 int 型。long 型（长整型）数至少有 32 比特位。尽管在某些机器中，int 和 long 具有相同的大小，但在其他一些机器中，int 只有 16 比特位，最大值为 32767，因此相对少的输入量就会造成 int 型的计数变量溢出。转换格式说明%ld 告诉 printf 对应的参数是一个 long 型整数。

如果需要处理比 long 还要大的数，则可使用 double（双精度浮点）类型。我们还将用 for 语句替代 while 语句，以说明循环的另一写法：

```
#include <stdio.h>

/* 统计输入字符的个数; 版本 2 */
main()
{
    double nc;

    for (nc = 0; gechar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

printf 使用%f 来代表 float 和 double 类型。格式%.0f 强制不打印小数点和小数部分（nc 的小数部分是 0）。

程序中 for 的循环体为空，这是因为所有的工作都在测试和递增部分做了。但是 C 语言的语法要求 for 语句必须有一个循环体。那个单独的分号就是为了满足这条规则，其称为**空语句**，将它单独放在一行是为了更加醒目。

在结束对字符统计程序的讨论之前，请注意如果输入不包含任何字符，那么 while 或 for 的测试在第一次调用 getchar 后就会失败，程序的执行结果将为 0——是正确的结果。这一点很重要。while 及 for 语句有一个很棒的特点就是它们在循环顶部进行测试——在执行循环体之前。如果没有什么需要做（条件不成立），那就什么也不做，甚至这意味着执行可能从未经过循环体。程序应该在出现 0 长度输入时灵活地进行处理。while 和 for 语句有助于确保程序在边界情况时进行合理的操作。

1.5.3 行计数

接下来的这个程序用于统计输入文本的行数。正如我们之前提到的，标准库保证了输入文本流以一连串文本行的形式出现，每行以一个换行符结束。因此，统计行数其实就是统计换行符的个数。

```
#include <stdio.h>

/*  统计输入的行数  */
main()
```

```
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

现在 while 的循环体是一条 if 语句，if 语句又控制着递增语句++nl。if 语句先测试圆括号中的条件，如果条件为真，则执行紧接其后的语句（或花括号中的一组语句）。我们仍使用缩进格式来表明语句之间的控制关系。

双等号 == 是 C 语言中表示“等于”的记号（就像 Pascal 中的 = 或者 Fortran 中的.EQ.）。由于 C 语言用单等号 = 表示赋值操作，因此使用与之区别的 == 表示相等测试。特别提请注意：初学者有时会在打算使用 == 时却写成 = ，在第 2 章将会看到，这样的误用一般会得到一个合法的表达式，所以系统不会给出任何警告。

用单引号引上的单个字符代表一个整数值，这个值等于具体机器字符集中该字符对应的数值。它被称作**字符常量**，尽管这只不过是一个较小整数的另一种写法而已。例如，'A'就是一个字符常量，在 ASCII 字符集中它的值是 65——即字符 A 在机器中的内部表示。不过显然'A'比 65 更为可取：它的意义很明显，并且独立于特定的字符集。

字符串常量中的转义序列用作字符常量也是合法的，因此'\n'代表换行符对应的值，在 ASCII 字符集中其值为 10。读者应该特别注意，'\n'是单个字符，在表达式中它只是一个整数；而"\n"却是一个恰巧只包含单个字符的字符串常量。有关字符串与字符的对比关系，我们将在第 2 章中进一步讨论。

练习 1-8. 写一个程序，统计输入中的空格、制表符和换行符的个数。

练习 1-9. 写一个程序，将输入拷贝到输出，并且将输入中连续的多个空格替换为单个空格。

练习 1-10. 写一个程序，将输入拷贝到输出，将每个制表符替换为\t，将每个空格替换为\b，将每个反斜杠替换为\\。这使得制表符和空格能够被明确地分辨出来。

1.5.4 单词计数

这个系列的第 4 个实用程序用于统计行、单词及字符的数目。这里对单词的定义比较宽松：它是任何不包含空格、制表符及换行符的字符序列。本程序是 UNIX 中 wc 程序的一个最简化的“光杆”版本。

```
#include <stdio.h>

#define IN   1    /* 在单词之内 */
#define OUT  0    /* 在单词之外 */

/* 统计输入中的行、单词和字符的个数 */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
```

```
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

程序每当碰到一个单词的首字符时就会将单词数加 1。变量 state 记录了程序当前的处理是否正处于某个单词之内；最开始程序处于“没有在处理单词”的状态，于是 state 被赋值为 OUT。就符号常量 IN 和 OUT 及其具体值 1 和 0 而言，我们更倾向于使用前者，因为它们会让程序的可读性更好。这对于像这么丁点大的程序不会有什么差别，然而对于更大型的程序，其带来的清晰性会非常值得从一开始就增加这一小点额外的努力。读者还会发现，当幻数只以符号常量的形式出现时，对程序的大范围修改会更加容易。

语句

```
nl = nw = nc = 0;
```

将这三个变量的值都设为 0。这不是一种特殊用法，而是由于每个赋值语句事实上是一个带值的表达式以及赋值操作从右至左的结合方式共同作用的结果。这个语句与如下写法

```
nl = (nw = (nc = 0));
```

是一样的。

运算符 || 的意思是 OR（逻辑或），所以

```
if (c == ' ' || c == '\n' || c == '\t')
```

的意义是“如果 c 是空格，或者 c 是换行符，或者 c 是制表符”。（之前提到过转义序列 \t 是制表符的可见表示形式。）还有一个与之对应的运算符 && 用于表示 AND（逻辑与），它比运算符 || 高一个优先级。由 || 或者 && 连接起来的表达式会从左向右进行求值，其运算机制保证整个逻辑表达式的真假一旦判定，求值就会马上停止。也就是说，如果 c 是一个空格，那么就没必要再测试它是否是一个换行符或制表符了，所以测试就不再进行。这一特性在此处并不是特别重要，但在更复杂的情况下其重要性将突显，我们很快就会看到这一点。

这个例子还给出了 else 部分，它指定了若 if 语句的测试条件为假(false)时所要另行完成的操作。其一般形式为：

```
if (表达式)
    语句1
else
    语句2
```

与 if-else 关联的这两条语句之中，有且只有一条会被执行。如果*表达式*为真，那么*语句*$_1$被执行；否则，*语句*$_2$被执行。*语句*$_1$和*语句*$_2$既可以是一条单独的语句，也可以是放在花括号中的多条语句。上述单词计数程序中，在 else 之后的那条 if 语句控制的就是由一对花括号括上的两条语句。

练习 1-11. 如果由你来测试这个单词计数程序，你打算怎么做？如果程序存在错误，哪些输入

最可能测出它们呢？

练习 1-12. 写一个程序把输入的单词打印出来，每行一个。

1.6 数组

现在我们编写一个程序来统计输入中各个数字、空白符（空格、制表符和换行符）以及所有其他字符出现的次数。这样虽然有些做作，但却可以在一个程序中展示 C 语言几个方面的内容。

输入的字符共有 12 种类别，因此使用一个数组来保存每个数字出现的次数比使用 10 个单独的变量更为方便。下面是该程序的一个版本：

```
#include <stdio.h>

/*  统计数字、空白符和其它字符的个数  */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
}
```

这个程序如果把自身作为输入，那么将会输出

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

程序中的声明语句

```
int ndigit[10];
```

将 ndigit 声明为一个由 10 个整数构成的数组。在 C 语言中，数组的下标总是从 0 开始，因此该数组的元素分别是 ndigit[0], ndigit[1], ..., ndigit[9]，这从初始化和打印该数组的两个 for 语句中可以看出来。

数组的下标可以是任何整数表达式，其中也包括像 i 这样的整数变量以及整数常量。

这个特定的程序依赖于数字的字符表示特性。例如，测试

```
if (c >= '0' && c <= '9') ...
```

用于确定 c 中的字符是否为数字。如果是数字，那么它的值等于

```
c - '0'
```

这只有在'0', '1', ..., '9' 具有连续递增的值时才会成立。幸运的是，在所有的字符集中这都是正确的。

根据定义，字符不过是小的整数，所以在算术表达式中 char 型的变量和常量与 int 的作用相同。这很自然也很方便。例如，c-'0'是一个整数表达式，如果 c 存储的是'0'到'9'中的一个字符，那么它的值就在 0 到 9 之间，这也正是数组 ndigit 的一个有效下标。

至于一个字符是数字、空白符还是其它字符，程序通过下面的序列完成判断：

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

模式

```
if (条件1)
    语句1
else if (条件2)
    语句2
...
    ...
else
    语句n
```

作为表述多路判定的一种方式在程序中经常出现。程序会对该模式中的条件从上到下依次测试，直到某个条件成立为止，然后执行该条件对应的语句，此后整个判定结构结束。（同样，每个条件对应的语句也可以是包含在花括号中的多条语句。）如果所有条件都不满足，而且存在最后一个 else 项，则该部分的语句将被执行；如果最后的 else 及语句被省略掉（就如之前的单词统计程序中的情况），则什么动作都不会发生。在最先的 if 和最后的 else 之间，可以存在任意数量的

```
else if(条件)
    语句
```

就编程风格而言，我们建议这种判定结构采用本节所给的格式。如果每个 if 语句都比前面的 else 缩进一级，那么一个较长的判定序列就可能会超出页面的右边界。

在第 3 章我们将讨论 switch 语句，它提供了另一种处理多路分支的方法，其尤其适合于判定某个整数表达式或字符表达式的值是否与一个常数集合中的某个值相匹配的情况。作为对比，在 3.4 节中我们将给出这个程序的 switch 版本。

练习 1-13. 写一个程序，打印输入中单词长度的直方图。水平方向的直方图比较容易；垂直方向的直方图更有难度。

练习 1-14. 写一个程序，打印输入中不同字符出现频度的直方图。

1.7 函数

C 语言的函数类似于 Fortran 语言的子程序或函数，或 Pascal 语言的过程或函数。函数为计算的封装提供了一种便捷的方法，在使用它时不必关心它的具体实现。有了恰当设计的函

数，你就可以不用了解一个工作是如何完成的，只需要知道完成了什么就足够了。C 语言中使用函数的方法简单、方便而且高效。你会经常看到一些定义后仅调用了一次的短函数，其目的只是为了使某些代码段更加清晰。

到目前为止，我们只使用过类似 printf、getchar 和 putchar 等提供给我们的函数；现在是时候写一些自己的函数了。C 语言没有提供与 Fortran 语言的 ** 类似的幂运算符，我们将编写一个幂函数 power(m,n) 并以此说明函数定义的机制。函数 power(m,n) 计算整数 m 的 n 次幂（n 为正整数）。也就是说，power(2,5) 的值为 32。这个函数并不是一个实用的幂运算程序，因为它只能处理较小整数的正整数次幂，不过用于说明它完全够用了。（标准库中提供了一个 pow(x,y) 函数用于计算 x^y。）

下面是函数 power 以及用于执行它的程序 main，因此整个程序结构读者立刻就能够看见。

```
#include <stdio.h>

int power(int m, int n);

/* 测试 power 函数 */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power: 计算 base 的 n 次幂; n >= 0 */
int power(int base, int n)
{
    int i,  p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

函数定义的形式如下：

```
返回值类型 函数名称(可能存在的形参声明)
{
    声明部分
    语句序列
}
```

不同函数的定义可以以任意次序出现在一个或多个文件中，但是单个函数定义不能分割放到多个文件里。如果源程序分散在多个文件中，那么可能必须用更多的指令来对它进行编译和加载，但那是操作系统的问题，而不是语言的属性。我们暂且假定这两个函数放在同一个文件中，这样之前所学的有关运行 C 程序的知识仍然有效。

函数 power 被 main 调用了两次，都在代码行

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

之中。每次调用时程序向 power 传递两个实参，power 则返回一个将被格式化并打印的整数。在表达式中，power(2,i) 就像 2 和 i 那样是一个整数。（并非所有的函数都生成一个整数值，在第 4 章中我们将进行讨论。）

整个 power 函数的第一行

```
int power(int base, int n)
```

声明了形参的类型和名称，以及函数返回值的类型。power 所使用的参数名只在 power 内部起作用，对其他函数不可见，也就是说其他函数可以使用相同的名字而不会发生冲突。变量 i 和 p 同样如此：power 函数中的 i 与 main 函数中的 i 互不相关。

我们通常使用**形参**（parameter）指代函数定义内括号中列出的变量名，而用**实参**（argument）指代函数调用时使用的常量或者变量。有时也使用术语**形式参数**（formal argument）和**实在参数**（actual argument）进行同样的区分。

power 计算得到的值通过 return 语句返回给 main 函数。return 之后可以跟任意的表达式：

```
return 表达式;
```

函数不一定要返回一个值；一个不带表达式的 return 语句将导致控制权交还给调用者，但不返回有用的值，就像碰到函数终止处的右花括号而从函数“尾部脱离”一样。调用函数也可忽略被调函数返回的值。

可能你已经注意到了，main 的末尾处有一个 return 语句。由于 main 本身也是函数，它也可以返回一个值给它的调用者——实际就是该程序的执行环境。典型地，返回值为 0 代表程序正常结束，非 0 值则表示异常或错误中止的情况。为简单起见，在此之前所有的 main 函数都省略了 return 语句，不过此后我们在其中都将加入 return 语句，以提醒读者一个程序应当向其运行环境返回它的状态。

在 main 函数之前的声明语句

```
int power(int base, int n);
```

表明 power 是一个函数，需要两个 int 型的实参并返回一个 int 型的值。这个声明被称为**函数原型**，它必须与 power 函数的定义及其使用相一致。如果某个函数的定义或使用与它的原型不一致，则是错误的。

形参的名字不需要一致。实际上，形参名在函数原型中是可选的，因此 power 的原型也可以写成：

```
int power(int, int);
```

然而，精心选择的名字有很好的说明作用，所以我们经常会加上它们。

历史注记：ANSI C 和早期的 C 版本之间最大的变化是函数声明和定义的方式。按照 C 语言的最初定义，power 函数会写成如下这样：

```
/* power:  计算 base 的 n 阶幂; n >= 0 */
/*         (老式版本) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
       p = p * base;
    return p;
}
```

形参的名字在圆括号中指定，而它们的类型则在函数开始的左花括号之前声明；未声明类型的形参会被当作 int 类型。（函数体的形式不变。）

程序开始处的 power 的声明看上去会像这样：

```
int power();
```

其不允许包含形参列表，因此编译器不能轻易地检验 power 是否被正确地调用。实际上，由于缺省情况下 power 被假定返回 int 值，这整个声明或许都会被省略掉。

新的函数原型语法使得编译器对实参数目及类型的错误检测要容易得多。老式的声明和定义在 ANSI C 中仍然有效，至少是在一个转换时期内有效，但是我们强烈推荐读者在编译器支持的情况下使用新的形式。

练习 1-15. 改写 1.2 节中的温度转换程序，用一个函数完成温度转换。

1.8 实参——按值调用

C 函数有一个方面可能对于习惯使用其它语言（尤其是 Fortran）的程序员不太熟悉。在 C 语言中，所有函数实参都是“按值”传递的。也就是说，被调函数所得到的是具有实参值的临时变量，而不是原有的实参本身。这导致 C 语言的实参传递特性与 Fortran 等“按引用调用”的语言以及 Pascal 中带有 var 选项的形参的相应特性有所差异，后两者的被调例程可以访问原有的实参，而不仅仅是例程内部的副本。

其主要区别在于，C 语言中被调函数不能直接修改调用函数中的变量，它只能修改其私有的临时副本。

然而，按值调用是一个优点而非缺点。由于在被调例程中参数可以方便地当作已初始化的局部变量，因此一般会减少额外变量的使用，使程序更加紧凑。例如下面的利用了这一性质的 power 函数版本：

```
/* power:  计算 base 的 n 次幂; n >= 0; 版本 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

参数 n 用作一个临时变量，通过一个递减的 for 循环向下计数直至为 0；这样变量 i 就不再需要了。函数 power 内部对 n 所做的任何操作都不会影响到调用 power 时所给的实参本身。

必要时也可以从被调函数内部来修改调用函数中变量的值。这种情况调用者必须提供这个变量的**地址**（技术化的说法是“指向该变量的**指针**”），被调函数必须将对应参数声明为一个指针，并通过它间接访问这个变量。我们会在第 5 章讨论指针的相关内容。

对于数组而言情况就不同了。当数组名作为参数时，传递给函数的值是这个数组的起始位置（或称起始地址），数组的元素并不进行拷贝。通过下标的方式，函数可以访问并修改数组内的任意元素。这是我们下一节要讨论的话题。

1.9 字符数组

字符数组是 C 语言中最常见的数组类型。为了展示字符数组的使用方法以及函数如何操作

它们，让我们来写一个程序，读入一组文本行，并打印出其中最长的一行。程序的基本框架足够简单：

```
while ( 还有未处理的行 )
    if ( 它比之前最长的行更长 )
        保存它
        保存它的长度
打印最长的行
```

上述框架很清晰——程序很自然地分为多个部分：一部分负责读入新的一行，一部分负责进行测试，一部分负责保存，余下的部分负责控制整个过程。

由于逻辑功能划分很明确，程序按照这种划分来编写也较为理想。首先，我们编写一个独立的函数 getline 用来获取输入的下一行。我们会尽量让这个函数在其它环境下也能使用。getline 至少要能返回一个表示已到文件末尾的信号。一种更有用的设计是函数返回文本行的长度，如果是文件末尾则返回 0。0 是一个可接受的文件末尾标记，因为它不会是有效的文本行长度。每个文本行至少包含一个字符，只包含一个换行符的文本行长度为 1。

当找到一个行比之前读入的最长文本行更长时，必须把该行保存起来。这意味则需要另一个函数 copy 将这个新行拷贝到一个安全的位置。

最后，我们需要一个 main 函数来控制 getline 和 copy 函数。得到的程序如下：

```
#include <stdio.h>
#define MAXLINE 1000   /* 最大输入行长度 */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* 打印最长的输入行 */
main()
{
    int len;               /* 当前行的长度 */
    int max;               /* 已读入的最长行的长度 */
    char line[MAXLINE];     /* 当前输入行 */
    char longest[MAXLINE];  /* 当前最长的文本行 */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0)  /* 如果存在一行 */
        printf("%s", longest);
    return 0;
}

/* getline:  将一行文本读取到 s 中，并返回其长度  */
int getline(char s[],int lim)
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
```

```
    }
    s[i] = '\0';
    return i;
}

/* copy:  将 'from' 拷贝到 'to'，假定 to 足够大 */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

函数 getline 和 copy 在程序的开始处进行了声明，我们假定程序放在单个文件中。

函数 main 和 getline 通过一组参数和一个返回值进行交互。在函数 getline 中，参数由代码行

```
int getline(char s[], int lim)
```

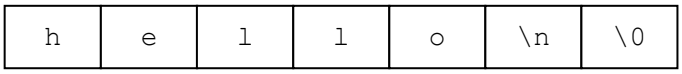
声明，第一个参数 s 是一个字符型数组，第二个参数 lim 是一个整数。在数组声明中提供长度的目的是预留存储空间。在 getline 中，数组 s 的长度不是必须的，因为 main 中已经设置了它的长度。与 power 函数一样，getline 通过 return 语句将一个值返回给调用者。该代码行还声明了 getline 的返回值为 int 类型；由于 int 为默认的返回类型，因此也可以省略掉。

一些函数会返回一个有用的值，而另一些函数（如 copy）只是用于完成一些操作，不需要有返回值。函数 copy 的返回类型为 void，显式地说明了函数不返回任何值。

函数 getline 会在其创建的字符数组尾部放置字符'\0'来标记该字符串的结束。（'\0'称为**空字符**，其值为 0。）这一约定也被用在 C 语言中：当 C 程序中出现如

```
"hello\n"
```

这样的字符串常量时，它会以一个字符数组的形式存储。这个数组包含该字符串的全部字符，并在尾部有一个'\0'标记字符串结束。



函数 printf 中格式说明%s 对应的参数就是以这种形式表示的字符串。函数 copy 同样需要其输入参数以'\0'结尾，并会将该字符拷贝至输出参数。（上述这些处理的一个隐含条件是'\0'不是普通文本的一部分。）

顺带值得一提的是，即使是这么小的程序也会遇到一些麻烦的设计问题。例如，如果输入字符串的长度超过了限制，main 函数应该如何处理？这种情况 getline 会可靠地工作，因为即使没有接收到换行符，当数组被填满时它也会停止接收数据。通过检查返回的字符串的长度和最后一个字符，main 函数可判断出该行是否过长，然后按其所需进行处理。为简洁起见，我们忽略了这个问题。

getline 函数的使用者无法预知输入行究竟会有多长，因此 getline 会对溢出情况进行检查。另一方面，copy 函数的使用者已经知道(或是能够查出)输入字符串的长度，因此我们没有在其中添加错误检测。

练习 1-16. 对 longest-line 程序的 main 函数进行修改，使之能够处理任意长度的输入行，以及尽可能多的输入文本。

练习 1-17. 编写一个程序，打印出所有长度超过 80 个字符的输入行。

练习 1-18. 编写一个程序，它将每个输入行末尾的所有空格和制表符去掉，并将全空白的行删除掉。

练习 1-19. 编写一个函数 reverse(s)，将字符串 s 翻转过来。使用这个函数写一个程序，把所有输入行进行翻转，每次翻转一行。

1.10 外部变量与作用域

函数 main 中的变量，如 line、longest 等，对于 main 来说是私有或局部的。因为它们在 main 函数内部声明，其他任何函数都不能直接访问它们。对于其他函数中的变量而言同样如此。例如，函数 getline 中的变量 i 与函数 copy 中的 i 毫不相关。每个函数中的局部变量只有在函数被调用时才存在，并在函数退出时消失。这就是此类变量通常被称为**自动**变量的原因（命名方式仿效了其它的语言）。以后我们将使用自动变量来指代局部变量。（在第 4 章将会讨论静态存储类型，这类局部变量在函数的不同调用之间能够保持其值。）

由于自动变量会随着函数的调用而产生和消亡，因此在不同的调用之间它们的值不会保留。每次函数调入时必须显式地设置自动变量的值，否则它们包含的将是无用数据。

除了自动变量，还可以使用另一种变量，此变量在所有函数的**外部**定义，任何函数都可通过变量的名字来访问它。（这一机制与 Fortran 语言中的 COMMON，以及 Pascal 语言中最外层区块中声明的变量极其类似）。由于外部变量可全局访问，它们可以代替参数列表用于函数之间的数据交互。并且，由于外部变量一直存在，不会随着函数的调用和退出而出现和消失，因此它们的值即使在设置该值的函数返回之后也仍然能够保持。

外部变量必须在所有函数之外**定义**，且只能定义一次；该定义为变量预留出存储空间。每个希望访问该变量的函数中还必须**声明**它；该声明说明了变量的类型。这个声明可能是显式的 extern 语句，也可能通过上下文隐式地说明。为了让讨论更加具体，我们将 line、longest 和 max 作为外部变量重新编写 longest-line 程序。全部三个函数的调用、声明以及函数体都需要改变。

```
#include <stdio.h>

#define MAXLINE 1000    /* 允许的输入行最大长度 */

int max;                /* 当前最长输入行长度 */
char line[MAXLINE];     /* 当前输入行 */
char longest[MAXLINE]; /* 当前最长输入行 */

int getline(void);
void copy(void);

/* 打印当前最长输入行，特殊版本 */
main()
{
    int len;
    extern int max;
    extern char longest[];
```

```
    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0)  /* there was a line */
        printf("%s", longest);
    return 0;
}

/* getline: 特殊版本 */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE - 1
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: 特殊版本 */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

main、getline 和 copy 中的外部变量定义在上面例子的开头几行。定义说明了它们的类型并为之分配存储空间。在语法上，外部变量的定义就像是局部变量的定义，但由于定义位于所有函数之外，因而成为外部变量。函数在使用外部变量之前必须知道该变量的名字，做到这点的方法之一就是在函数中给出一个外部声明；这种声明与之前的声明一样，只是加上了关键字 extern。

某些情况下 extern 声明能够被省略。如果一个源文件中外部变量的定义出现在某个使用该变量的具体函数之前，那么在该函数中就无需该变量的 extern 声明。因此 main、getline 和 copy 中的 extern 声明都是多余的。事实上，普遍的做法是将所有外部变量的定义放在源文件的起始位置，然后省略掉所有的 extern 声明。

如果程序分为多个源文件，而且变量在文件 1 中定义，并在文件 2 和文件 3 中使用，那么在文件 2 和文件 3 中就需要用 extern 声明来关联出现的变量。惯常的做法是将变量和函数的 extern 声明放在一个单独的文件中(其历史称谓为**头文件**)。每个源文件通过开头的#include 语句来包含头文件。按惯例头文件的名字后缀为 .h 。例如，标准库的函数就在类似 <stdio.h> 这样的头文件中声明。这个主题将在第 4 章中详细讨论，标准库本身则将在第 7

章和附录 B 中讨论。

由于特殊版本的 getline 和 copy 没有参数，从逻辑上说它们在源文件起始处的函数原型应该是 getline() 和 copy()。但是为了兼容旧版本的 C 程序，ANSI 标准将空参数表当作旧式声明，并关闭所有的参数表检查；而明确为空的参数表必须用关键字 void 来表示。相关内容将在第 4 章中进一步讨论。

读者应当注意到在本节当我们提及外部变量时很谨慎地使用*定义*和*声明*这两个词。变量被创建并分配存储的地方使用“定义”；给出变量特性但不分配存储的地方使用“声明”。

顺带提及有这样一种倾向存在：将所有用到的东西都设为外部变量，因为这看上去能简化程序的交互——参数列表很短，而且这些变量在要用时随时可用。但是外部变量即便在你不需要时也总是存在。过多地依赖于外部变量会使程序遍布危险，因为这将会导致程序的数据关联不明显——变量可由一些出其不意甚至是疏忽的方式改变，而且程序也会难于修改。程序 logest-line 的第二个版本不如第一个版本，部分是由于这些原因，部分则是由于其将需要操作的变量名字写到了函数 getline 和 copy 中，而使得这两个有用的函数失去了通用性。

至此，我们已经涵盖了那些也许会被称作 C 语言传统核心的内容。基于这些少量的语言“构件”，读者可能编写出规模可观的有用程序。或许停下来用足够长的时间去练习编写程序会是一个好主意。下面的程序练习比起本章先前的练习要复杂一些。

练习 1-20. 编写一个程序 detab，它将输入中的制表符替换为恰当数目的空格，使间隔达到下一制表符停止位。假定一组固定的制表符停止位，比如每 n 列一个停止位。n 应当是一个变量还是一个符号参数？

练习 1-21. 编写一个程序 entab，它将一连串空格替换为相同间隔的最小数目的制表符和空格。使用与 detab 相同的制表符停止位。当单个制表符或者单个空格都能达到制表符停止位时，选用哪一种更好？

练习 1-22. 编写一个程序，其将一个长输入行“折”为较短的几行，折行的位置为输入的第 n 列之前的最后一个非空白符之后。确保你的程序能够智能地处理很长的输入行，以及在指定列之前没有空格或制表符的情况。

练习 1-23. 编写一个程序，用于去掉 C 程序的所有注释。注意正确地处理带引号的字符串和字符常量。C 语言的注释不允许嵌套。

练习 1-24. 编写一个程序，其用于检查 C 程序的基本语法错误，例如不配对的圆括号、方括号和花括号。别忘了对单引号、双引号、转义序列以及注释的处理。（如果读者想把它写成完全通用的程序，难度会比较大。）

第 2 章 类型、运算符以及表达式

变量和常量是程序处理的基本数据对象。声明列出了所要使用的变量，说明这些变量的类型并可能给出它们的初始值。运算符指定了这些变量所要进行的运算。表达式将一些变量和常量结合起来生成新的数值。对象的类型决定了该对象所能被赋予的值以及能够进行的运算。以上这些语言元素将是本章的主题。

ANSI 标准对基本类型以及表达式做了许多小的增改。现在所有整数类型都具有 signed（带符号）和 unsigned（无符号）两种形式，无符号常量及十六进制字符常量也有了特定的表示法。浮点运算可以按单精度进行；同时也有用于扩展精度的 long double 类型。多个字符串常量可以在编译时进行拼接。枚举类型，这种已长期存在的特性正式变成了语言的一部分。对象可以被声明为 const 类型以避免其值被改变。在算术类型之间的强制类型转换规则已被扩展用于处理更多的类型。

2.1 变量名

尽管我们在第 1 章中没有说明，但是对变量和符号常量的命名是有一些规则要求的。名字由字母以及数字组成；首字符必须是一个字母。下划线“_”被当作一个字母；有时候它对于提高长变量名的可读性比较有用。但是不要让变量名以下划线开头，因为库例程常常使用以下划线开头的名字。大写字母和小写字母会被区分，因此 x 和 X 是不同的两个名字。传统上 C 的做法是变量名使用小写字母，而符号常量则使用全大写的字母。

内部名字（internal name）的至少前 31 个字符是有效的。对于函数名和外部变量来说，名字的字符数量可能会少于 31 个，因为外部名字（external name）可能会被语言无法控制的汇编器和加载器所使用。对于外部名字，ANSI 标准只保证前六个字符的唯一性且不区分大小写。像 if、else、int、float 等关键字都是保留的：你不能够将它们用作变量名。关键字使用的必须是小写字母。

选择与变量作用有关系的变量名是明智的做法，这样也不容易造成拼写上的混淆。我们倾向于对本地变量使用短名字，特别是循环控制变量，而对外部变量使用较长的名字。

2.2 数据类型及大小

C 语言中只有几种基本数据类型：

```
char        单个字节，用于存储本地字符集中的一个字符。
int         一个整数，通常为具体机器上整数的自然大小。
float       单精度浮点数。
double      双精度浮点数。
```

此外，还有一些能够用于这些基本类型的限定词。short 和 long 可用于整型数：

```
short int sh;
long int counter;
```

这种声明中的 int 可以省去（这也是典型的做法）。

引入 short 与 long 的目的是为了提供满足实际需求的不同长度的整数；int 通常是特定

机器的字长。short 常常为 16 比特，long 为 32 比特，而 int 则为 16 或者 32 比特。每个编译器可以根据其硬件自由选择适当的大小，唯一的限制是 short 和 int 至少有 16 比特，long 至少有 32 比特，并且 short 不能比 int 长，int 不能比 long 长。

限定符 signed 与 unsigned 可用于 char 型或者任何整数类型。被 unsigned 限定的数一定是正数或 0，并遵循模 2^n 原则，其中 n 为该数据类型的比特位数。例如，如果 char 为 8 比特，unsigned char 型变量的值域为 0~255，而 signed char 型变量的值域为-128~127（在采用二进制补码的机器上）。未限定的 char 型变量是有符号（signed）还是无符号（unsigned）取决于具体机器，但是可打印字符一定是正值。

long double 类型用于指定扩展精度浮点数。与整数类似，浮点对象的大小也由具体实现定义，float、double 和 long double 类型的对象可以表示相同的大小，也可以表示两种或者三种不同的大小。

标准头文件 <limits.h> 和 <float.h> 包含了与具体机器和编译器相关的所有这些大小以及其他特性的符号常量。这些内容将在附录 B 中进行讨论。

练习 2-1. 编写一个程序来确定 char、short、int 和 long 型变量的取值范围，包括 signed 及 unsigned 类型。通过打印标准头文件中的相应值以及直接计算两种方法来完成。如果通过计算方式，确定各种浮点类型的取值范围会更困难。

2.3 常量

如 1234 这样的整数常量是 int 型常量。long 型常量以字母 l 或 L 结尾，如 123456789L；超过 int 型存放范围的整数常量也会被当作 long 型常量处理。无符号常量以字符 u 或 U 结尾，后缀 ul 或 UL 则用于表示 unsigned long 型常量。

浮点常量包含一个小数点（如 123.4）或指数（如 1e-2）或两者皆有。未带后缀的浮点常量的类型为 double；后缀 f 或 F 表示 float 常量；而后缀 l 或 L 表示 long double 常量。

除十进制外，整数值还可以表示为八进制或十六进制。一个整数常量如果以数字 0 开头则为八进制；以 0x 或 0X 开头则为十六进制。例如，十进制数 31 可以写成八进制数 037 或者十六进制数 0x1f 或 0X1F。八进制和十六进制常量也可以带上表示 long 型的后缀 L 以及表示 unsigned 型的后缀 U：例如，0XFUL 即为一个值为 15（十进制）的 unsigned long 型常量。

字符常量是一个整数，其形式为一个括上单引号的字符，如'x'。字符常量的值是具体机器的字符集中该字符对应的数值。例如，在 ASCII 字符集中，字符常量'0'的值为 48——与数值 0 毫无关系。如果代码使用字符'0'而不是使用 48 这样依赖于字符集的数值，那么程序将会独立于特定的值并且更易阅读。虽然字符常量主要用来与另外的字符进行比较，但也可以像其他任何整数一样参与数值运算。

在字符常量和字符串常量中，有些字符可能通过转义序列来表示，如 \n（换行符）；这些序列看上去像两个字符，但其仅表示单个字符。此外，字节大小的任意位模式都能用以下形式指定：

'*ooo*'

其中 *ooo* 代表一至三个八进制数字（0…7）或者

'\x*hh*'

其中*hh*是一至多个[①]十六进制数字(0…9，a…f，A…F)。所以我们编写的代码可能会是：

```
#define  VTAB  '\013'   /* ASCII 纵向制表符 */
#define  BELL  '\007'   /* ASCII 响铃符 */
```

或按十六进制写为

```
#define  VTAB  '\xb'    /* ASCII 纵向制表符 */
#define  BELL  '\x7'    /* ASCII 响铃符 */
```

下面是所有的转义序列：

\a	响铃符	\\	反斜杠
\b	回退符	\?	问号
\f	换页符	\'	单引号
\n	换行符	\"	双引号
\r	回车符	*ooo*	八进制数
\t	横向制表符	\x*hh*	十六进制数
\v	纵向制表符		

字符常量'\0'表示一个值为 0 的字符，即空字符。我们常使用'\0'来替代 0 以强调某些表达式的字符性质，但其数值就是 0。

常量表达式是其中只包含常量的表达式。这种表达式的求值可以在编译时完成，而不必等到运行时才进行，因而它可用于常量能够出现的任何位置。例如：

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

或

```
#define LEAP 1   /* 闰年 */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

字符串常量也叫**字符串字面值**，是用双引号括上的由零个到多个字符组成的序列。例如：

```
"I am a string"
```

或：

```
""  /* 空字符串 */
```

双引号不是字符串的一部分，它仅用于限定字符串。在字符常量中使用的转义序列同样用在字符串中； \" 表示双引号字符。首尾相邻的字符串常量可以在编译时被拼接起来：

```
"hello," " world"
```

等价于

```
"hello, world"
```

此特性在将长字符串拆分为若干源代码行时较为有用。

从技术角度看，字符串常量是一个字符数组。字符串的内部表示在结尾处有一个空字符'\0'，因此存放字符串所需的物理存储比双引号内的字符数多 1。这种表示方法意味着对字符串的长度没有限制，但程序必须扫描整个字符串以确定其长度。标准库函数 strlen(s)用于返回其字符串参数 s 的长度，不包括终止位置的'\0'。下面是我们给出的版本：

```
/* strlen: 返回 s 的长度*/
int strlen(char s[])
{
    int i;
```

[①]对于 *h* 的数目没有限制，但如果构成的字符值超过了最大的字符，则行为是未定义的。——译注

```
    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

strlen 和其他字符串函数在标准头文件 <string.h> 中声明。

请注意区分字符常量和仅包含单个字符的字符串：'x'与"x"并不相同。前者是一个整数，用于产生字母 x 在机器字符集中对应的数值；后者是包含一个字符（即字母 x）及一个'\0'的字符数组。

C 语言中还有一种常量，称为**枚举常量**。枚举是一个由常量整数值组成的列表，例如：

```
enum boolean { NO, YES };
```

在枚举常量中，默认第一个名字的值是 0，下一个是 1，依次类推，除非名字给定了明确的值。如果不是所有的值都被给定，未给定的值会依着最后一个给定值向后递增，如下面例子中的第二例：

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB  = '\v', RETURN = '\r' };

enum moths { JAN = 1, FEB , MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC };
                /* FEB 为 2，MAR 为 3，依次类推 */
```

不同枚举常量中的名字必须各不相同，同一枚举常量中各个名字的值无须不同。

枚举提供了一种将常量值和名字关联起来的便利方法。作为#define 语句的另一选择，其优点在于常量的值可以自动生成。虽然可以声明 enum 类型的变量，但编译程序不需要检查在这个变量中存储的值是否是该枚举的有效值；不过枚举变量提供了进行这种检查的机会，因而常常比使用#define 更好。此外，调试程序还能以符号形式打印出枚举变量的值。

2.4 声明

所有变量在使用之前都必须声明，尽管其中一些声明可以通过上下文隐式地给出。一个声明指定了一个类型，并包含了由一个或多个此类型的变量组成的列表。例如：

```
int  lower, upper, step;
char c, line[1000];
```

变量可按任何样式分布于声明之中；上述列表可等价地写成

```
int  lower;
int  upper;
int  step;
char c;
char line[1000];
```

后一种形式所占空间更多，但对每一声明添加注释或者以后修改都较为方便。

变量还可以在其声明中进行初始化。如果变量名之后带有一个等号和一个表达式，那么该表达式就起到初始化单元的作用。例如：

```
char esc = '\\';
int  i = 0;
int  limit = MAXLINE+1;
float eps = 1.0e-5;
```

如果变量不是自动变量，那么初始化仅进行一次，概念上在程序开始执行之前完成，并且初始化单元必须是一个常量表达式。显式初始化的自动变量当其所在函数或程序块每次进入时都要进行初始化；其初始化单元可以是任何表达式。外部变量和静态变量默认被初始化为零。未显式初始化的自动变量的值则是未定义的（如无用数据）。

限定符 const 可被应用于任何变量的声明中，指明该变量的值不会被改变。对于数组而言，const 限定符则说明数组的元素不会被改动。

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

const 声明也能用在数组参数上，表明此函数不会修改这个数组：

```
int strlen(const char[]);
```

如果试图修改一个 const 变量，其结果由具体实现定义。

2.5 算术运算符

二元算术运算符包括 +、 - 、*、 / 以及模运算符 % 。整数除法会截掉任何未被整除的部分。表达式

```
x % y
```

的结果为 x 除以 y 所得的余数，当 y 整除 x 时其值为零。例如，某年份为闰年的条件是它能被 4 整除但不能被 100 整除，或者能被 400 整除。因此

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
   printf( "%d 是闰年\n" , year);
else
   printf( "%d 不是闰年\n" , year);
```

运算符 % 不能用于 float 型和 double 型。对于负操作数， / 的截取方向和 % 结果的正负依赖于具体机器，如同上溢或下溢时的处理方式一样。

二元运算符 + 和 - 具有相同的优先级，它们的优先级比 *、/、% 低，而 *、/、% 的优先级又比一元运算符 + 和 - 低。算术运算符是从左至右进行结合的。

位于本章最后的表 2-1 总结了全部运算符的优先级与结合律。

2.6 关系运算符和逻辑运算符

关系运算符

```
>   >=  <   <=
```

具有相同的优先级。优先级仅比它们低一级的是相等运算符：

```
==  !=
```

关系运算符的优先级低于算术运算符，因此 i < lim-1 这样的表达式会像预料中那样按 i < (lim-1) 处理。

更加有趣的是逻辑运算符 && 和 || 。以 && 或 || 连接的表达式按从左至右的次序求值，一旦获知结果的真假，求值过程就会立即终止。许多C程序依赖于这样的特性。例如，下面

的循环来自于我们在第 1 章中编写的输入函数getline[1]：

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

在读入一个新字符之前，必须检查数组 s 中是否有存放它的空间，因此测试 i < lim-1 必须首先进行。并且如果这一测试失败，我们就不必继续读入另一字符了。

类似地，假如 c 不等于 EOF 的测试在 getchar 调用之前进行，那么程序就会错误地执行；因此该调用及赋值必须发生在 c 中的字符被测试之前。

&& 的优先级比 || 高，但它们的优先级比关系运算符和相等运算符都要低。因此类似

```
i<lim-1 && (c = getchar()) != '\n' && c != EOF
```

这样的表达式不需要额外的括号。但由于 != 的优先级比赋值运算符高，因而

```
(c = getchar()) != '\n'
```

需要括号来达成期望的结果：首先对 c 赋值然后将 c 与'\n'进行比较。

根据定义，当关系或逻辑为真时，关系表达式或逻辑表达式的数值为 1；为假时表达式的数值为 0。

一元取反运算符 ! 将一个非零操作数转换为 0，而将为零的操作数转换为 1。! 的一个普遍用法是在类似下面的代码构成中用

```
if (!valid)
```

来替代

```
if (valid == 0)
```

两种形式哪种更好很难一概而论。类似 !valid 这样的代码构成读上去很顺(“if not valid”即“如果不是有效的”)，但更复杂的构成理解起来则可能会很困难。

练习 2-2. 编写一个与上面的 for 循环等价但没有采用 && 或 || 的循环。

2.7 类型转换

当一个运算符拥有不同类型的操作数时，这些操作数会依据一些规则转换为同一种类型。一般而言，自动转换只是哪些将“较窄的”操作数转换为“较宽的”操作数而不丢失信息的转换，例如表达式 f + i 中整数向浮点数的转换。无意义的表达式（如使用浮点数作为下标等）是不允许的。可能丢失信息的表达式（如将一个长整型数赋给一个短整型数，或将一个浮点型数赋给一个整数）会引发一个警告，但它们并不是非法的。

char 就是一个较小的整数，因此 char 可以自由地用于算术表达式中。这为某几类字符转换带来了相当的灵活性。下面这个初步实现的函数 atoi 就是一例，其将一个表示数字的字符串转换为对应的数字。

```
/* atoi: 将 s 转换为对应的整数 */
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
```

[1] 此循环与 1.9 节的循环在形式上不是完全相同，但在功能上是等价的。

```
    return n;
}
```

正如我们第 1 章中讨论的那样，表达式

```
s[i] - '0'
```

给出了 s[i] 中存放字符所对应的数，这是因为 '0' 、'1' … 等字符的值是一个连续递增的序列。

char 转换为 int 的另一个例子是函数 lower，它将单个字符映射为 ASCII 字符集中的小写形式。如果该字符不是一个大写字母，那么 lower 将其原样返回。

```
/* lower: 将 c 转换为小写形式; 仅适用于 ASCII 码 */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

代码对于 ASCII 码是有效的，因为 ASCII 码中大写字母和对应小写字母的距离是固定的数值，并且每个字母表都是连续的——在 A 和 Z 之间全是字母。后者对 EBCDIC 字符集并不适用，因此在 EBCDIC 中上述代码不会只对字母的进行转换。

在附录 B 中描述的标准头文件 <ctype.h> 定义了一组独立于字符集的用于字符测试和转换的函数。例如，函数 tolower(c) 当 c 为大写时返回 c 的小写值，因此 tolower 是上面给出的 lower 函数的一个可移植的替代者。类似地，测试

```
c >= '0' && c <= '9'
```

可以被替代为

```
isdigit(c)
```

从现在起，我们将使用 <ctype.h> 给出的函数。

关于字符到整数的转换有一个微妙之处：C 语言没有规定 char 型变量是有符号还是无符号的量；当 char 转换为 int 时，究竟能否生成一个负的整数？答案随具体机器不同而变化，反映出机器架构的差异。在一些机器上，最左端比特位为 1 的 char 型量会被转换为负整数（“符号扩展”）；而在其他机器上，会通过将 char 型量的左端添加 0 来扩展为整数，因而其值一定为正。

C 语言的定义保证了机器的标准可打印字符集中的任何字符都不为负，因此这些字符在表达式中总是正值。但是任意给定比特模式的字符变量可能在某些机器上为负，而在其他机器上为正。为了程序的可移植性，如果打算将非字符数据存放在 char 型变量中，请指定 signed 或 unsigned 类型。

根据定义，如 i > j 等关系表达式以及由 && 和 || 连接的逻辑表达式如果为真，则表达式的值为 1，如果为假则值为 0。因此下面的赋值

```
d = c >= '0' && c <= '9'
```

当 c 是数字时将 d 置为 1，否则置为 0。然而，isdigit 等函数却会返回任意的非零值表示真。在 if、while、for 等语句的测试部分，“真”仅意味着“非零”，因此两者不会造成差别。

隐式的算术转换多数会像预料的那样运作。一般而言，如果类似 + 或 * 等需要两个操作数的运算符（即二元运算符）其操作数的类型不同，那么“较低”类型会被**提升**为“较高”类型，然后才进行运算；运算结果是较高的类型。附录 A 的第 6 节精确地说明了相关转换规则。

但如果没有 unsigned 型的操作数，下面这组非正式的规则就足够了：

若任一操作数是 long double 型，将另一操作数转换为 long double 型；
否则，若任一操作数是 double 型，将另一操作数转换为 double 型；
否则，若任一操作数是 float 型，将另一操作数转换为 float 型；
否则，将 char 和 short 型转换为 int 型；
然后，如果任一操作数是 long 型，将另一操作数转换为 long 型。

请注意在表达式中 float 型不会自动地转换为 double 型；这是对 C 语言最初定义的一个改变。通常，像那些在 <math.h> 中的数学函数会使用双精度（即 double）。使用 float 的主要原因是为了节省大型数组的存储空间，较少些的原因是为了节省某些双精度运算代价特高的机器的运行时间。

当表达式中包含 unsigned 类型的操作数时，转换规则要复杂一些。具体原因是有符号值与无符号值的比较与机器相关，这取决于各个整数类型的大小。例如，假定 int 为 16 位，long 为 32 位，那么-1L < 1U，这是因为 unsigned int 型的 1U 被提升为 signed long 类型；但是 -1L > 1UL，这是因为-1L 被提升为 unsigned long 类型，这样它就变成了一个大的正数。

赋值时也会进行类型转换；赋值运算右边的值会被转换为左边变量的类型，这也是结果的类型。

字符转换为整数，无论通过符号扩展与否，其做法如前所述。

较长的整数转换成较短的整数或字符时，要把超出的高位部分去掉。于是当程序

```
int i;
char c;

i = c;
c = i;
```

执行后，c 的值将会保持不变，无论是否进行符号扩展都是这样。然而，如果把两个赋值语句的次序颠倒一下，则可能会丢失信息。

如果 x 是 float 型且 i 是 int 型，那么 x = i 以及 i = x 都会导致类型转换；将 float 型转换成 int 型时会截掉任何小数部分。而当 double 型转换成 float 型时，是舍入还是截取取决于具体实现。

由于函数调用的参数是一个表达式，所以类型转换也会发生在将参数传递给函数的过程中。在没有函数原型的情况下，char 和 short 类型将会转换成 int，float 类型将会转换成 double。这就是为什么我们之前将函数参数声明成 int 或者 double，甚至当函数被调用时传递的参数是 char 或者 float 时也这样做。

最后，在任何表达式中都可以进行显式的类型转换（即所谓的“强制转换”），这时要使用一个叫做**强制转换**（cast）的一元运算符。在如下结构中：

```
(类型名) 表达式
```

表达式按上述转换规则被转换成由强制类型转换运算符所指定的类型。强制类型转换的精确含义就像是表达式首先被赋给一个所指定类型的变量，然后再将其替换到整个结构所在的位置。例如，库函数 sqrt 需要一个 double 类型的参数，但如果无意中给成了其他类型，那么就会产生无意义的结果。（sqrt 在 <math.h> 中声明。）因而，如果 n 是一个整数，那么可以用

```
sqrt((double )n)
```

将 n 的值转换成 double 型之后再传递给 sqrt。注意，强制类型转换只是按恰当类型产生 n 的值，n 本身并未被修改。如本章末尾的表中总结的那样，强制类型转换运算符的优先级与其

他一元运算符相同。

如果参数通过函数原型进行了声明（通常会是这样），该声明会使得在函数调用时自动进行参数的强制类型转换。这样，给出 sqrt 的一个函数原型：

```
double sqrt(double);
```

调用

```
root2 = sqrt(2);
```

不需要强制转换运算符就能自动将整数 2 强制转换成 double 型的值 2.0。

标准库包含有一个可移植的伪随机数生成器以及一个用于初始化其种子的函数；前者给出了一个强制转换的示例：

```
unsigned long int next = 1;

/* rand: 返回一个伪随机整数，其值域为 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: 为 rand()设置种子 */
void srand(unsigned int seed)
{
    next = seed;
}
```

练习 2-3. 编写函数 htoi(s)，其将一个表示 16 进制数的字符串（包括可选的 0x 或 0X）转换为对应的整数值。允许出现的数字包括 0~9、a~f 以及 A~F。

2.8 递增与递减运算符

C 语言提供了两个特殊的运算符，分别用于变量的递增和递减。递增运算符 ++ 将操作数增 1，而递减运算符 -- 将操作数减 1。我们已频繁地使用 ++ 来递增变量，譬如

```
if (c == '\n')
    ++nl;
```

特殊的方面是 ++ 和 -- 既可以用作前缀运算符（在变量之前，如++n），又可用作后缀运算符（在变量之后，如 n++）。两种情况的运算效果都是将 n 递增。但是表达式 ++n 在 n 的值被使用之前将 n 递增，而 n++ 则在 n 的值使用之后再将其递增。这意味着在需要使用表达式值的上下文中，除对 n 的作用之外，++n 与 n++ 是不同的。如果 n 为 5，那么

```
x = n++;
```

将 x 置为 5，而

```
x = ++n;
```

将 x 置为 6。两种情况下，n 都变为 6。递增和递减运算符只能作用于变量；像 (i+j)++ 这样的表达式是不合法的。

在只需要递增效果，而不需要表达式值的上下文中，如

```
if (c == '\n')
```

```
        nl++;
```

前缀和后缀是相同的。但是某些情况下会特别运用其中的某一种。例如，考虑函数 squeeze(s,c)，其将字符串 s 中出现的字符 c 全部去掉。

```
/* squeeze: 删除 s 中的所有 c */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

每当出现一个不是 c 的字符，该字符即被复制到当前位置 j，此时 j 才被递增以备存放下一个字符。该 if 语句完全等同于

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

结构与之类似的另一例子来自第 1 章中我们编写的 getline 函数，我们可将

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

替换为更加紧凑的

```
if (c == '\n')
    s[i++] = c;
```

作为第三个例子，考虑标准库函数 strcat(s,t)，其将字符串 t 拼接到字符串 s 的末尾。strcat 假定 s 中有足够的空间容纳组合后的字符串。我们编写的 strcat 没有返回值；标准库的版本会返回一个指向该组合字符串的指针。

```
/* strcat: 将 t 拼接到 s 的末尾; s 必须够大 */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0')  /* 找到 s 的末尾 */
        i++;
    while ((s[i++] = t[j++]) != '\0')  /* 复制 t */
        ;
}
```

随着每一字符从 t 复制到 s，后缀 ++ 分别作用于 i 和 j，以保证它们处于适当位置以备下一轮循环处理。

练习 2-4. 编写 squeeze 函数的另一版本 squeeze(s1,s2)，其从字符串 s1 中删除掉那些但凡出现在字符串 s2 中的字符。

练习 2-5. 编写函数 any(s1,s2)，其返回 s1 中首次出现字符串 s2 中任何字符的位置；如果 s1 中不包含 s2 中的任何字符，则返回 -1 。（标准库函数 strpbrk 完成同样的工作，但

它返回一个指向该位置的指针。）

2.9 按位运算符

C 语言提供了六种用于位操作的运算符；它们只能用于整型操作数，即有符号或无符号的 char、short、int 以及 long 型整数。

```
&       按位与
|       按位或
^       按位异或
<<      左移
>>      右移
~       按位取反（一元运算符）
```

按位与运算符 & 常用于屏蔽一些位；例如，

```
n = n & 0177;
```

将 n 除了低 7 位之外的比特位设为 0。

按位或运算符 | 则常用于打开某些位：

```
x = x | SET_ON;
```

按照 SET_ON 中为 1 的位，将 x 中对应的比特位设为 1。

按位异或运算符 ^ 当两个操作数对应比特位（的值）不同时，将该位设为 1；相同时将该位设为 0。

读者必须将按位运算符 & 及 | 与逻辑运算符 && 及 || 区分开，后者意味着从左至右进行逻辑真假的求值。例如，如果 x 为 1、y 为 2，那么 x & y 为 0 而 x && y 为 1。

移位运算符 << 和 >> 按其右操作数指定的位数（必须是非负值）将其左操作数进行左移或者右移。因此 x << 2 将 x 的值左移两位，并将空出的位用 0 填充；这相当于 x 乘以了 4。右移一个无符号数时空出的位总是用 0 填充。右移一个有符号数时，一些机器上会填充符号位（“算术右移”），而另一些机器上则会填充 0（“逻辑右移”）。

一元运算符 ~ 将一个整数按位取反，即将每个为 1 的比特位换为 0，将为 0 的位换为 1。例如，

```
x = x & ~077
```

将 x 的最后 6 位设置为 0。注意，x & ~077 与字长无关，因而比类似 x & 0177700 等写法更好，后者假定 x 是一个 16 比特的量。这种可移植的形式不需要额外的开销，因为 ~077 是一个常量表达式，可以在编译时求值。

作为一些位运算符的示例，考虑函数 getbits(x,p,n)，其返回 x 中从位置 p 起始的 n 个比特位（已右对齐）。我们假定位置 0 为最右端的比特位，n 和 p 都是合理的正值。例如，getbit(x,4,3)返回位置为 4、3、2 的（并经过了右对齐的）比特位。

```
/* getbits: 获取从位置 p 起始的 n 个比特位 */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

表达式 x >> (p+1-n)将所需位段移动到字的最右端。~0 是比特值全为 1 的数；~0 << n 将它左移 n 位，将最右边的 n 个比特位设置为 0；最后用 ~ 取反得到最右 n 个比特位为 1 的掩码。

练习 2-6. 编写一个函数 setbits(x,p,n,y)，其将 x 从位置 p 起始的 n 个比特位设置为 y 的最右 n 个比特位，x 的其余位不变；返回 x。

练习 2-7. 编写一个函数 invert(x,p,n)，其将 x 从位置 p 起始的 n 个比特位取反（即 1 变为 0，0 变为 1），x 的其余位不变；返回 x。

练习 2-8. 编写一个函数 right(x,n)，其返回整数 x 向右循环移动 n 位所得到的值。

2.10 赋值运算符与赋值表达式

诸如

```
i = i + 2
```

等左边变量随即出现在右边的表达式可被写为如下压缩形式：

```
i += 2
```

运算符 += 被称为**赋值运算符**。

大多数二元运算符（与 + 类似的具有左操作数和右操作数的运算符）都有对应的赋值运算符 *op*= ，*op* 为如下运算符之一：

```
+    -    *    /    %    <<   >>   &    ^    |
```

如果 $expr_1$ 和 $expr_2$ 是表达式，那么

$expr_1$ *op*= $expr_2$

相当于

$expr_1$ = ($expr_1$) *op* ($expr_2$)

只是前者的 $expr_1$ 仅会被计算 1 次。留意 $expr_2$ 周围的括号：

```
x *= y + 1
```

表示

```
x = x * (y + 1)
```

而不是

```
x = x * y + 1
```

一个具体的例子是函数 bitcount，它统计其整数参数中为 1 的比特位的个数。

```
/* bitcount: 统计 x 中为 1 的比特位 */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

将参数 x 声明为 unsigned 类型保证了 x 向右位移时空出的比特位会被填充为 0——无论程序运行在何种机器上。

撇开简洁性不谈，赋值运算符的优点是与人们的思维方式匹配得更好。我们会说"将 2 加到 i 中"或者"将 i 增加 2"，而不是"取 i，加上 2，然后将结果放回 i 中"。因此表达式 i += 2

比 i = i + 2 要好。此外，对于一个复杂的表达式，譬如

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

赋值运算符使得代码更易理解，读者不必费力地检查两个长表达式是否真正一致，或是奇怪为什么它们不一样。赋值运算符甚至还能帮助编译器产生高效的代码。

我们已经见到过：赋值语句有其自己的值，并且能够出现在表达式中。最常见的例子是

```
while ((c = getchar()) != EOF)
    …
```

其他赋值运算符（+=、-= 等等）也能出现在表达式中，不过要少见一些。

在所有这类表达式中，赋值表达式的类型是它左操作数的类型，它的值则等于赋值完成后的值。

练习 2-9. 在采用二进制补码的系统中，x &= (x-1) 将 x 中最右边为 1 的比特位去掉（即置为 0）。请解释原因。利用这一经验编写一个速度更快的 bitcount 版本。

2.11 条件表达式

语句

```
if (a > b)
    z = a;
else
    z = b;
```

计算 a 与 b 的最大值并放到 z 中。使用三元运算符"?:"编写的**条件表达式**提供了另一种编写上述语句或者类似结构的方法。在表达式

$expr_1$? $expr_2$: $expr_3$

中，首先表达式 $expr_1$ 被求值，如果为非零值（真），则表达式 $expr_2$ 被求值，并作为这整个条件表达式的值；否则 $expr_3$ 被求值并作为整个表达式的值。$expr_2$ 和 $expr_3$ 中仅有一个会被求值。因此将 z 设置为 a 与 b 中最大值的代码为：

```
z = (a > b) ? a : b;    /* z = max(a, b) */
```

应当注意条件表达式是一个真正的表达式，它可以用在任何其他表达式能用的位置。如果 $expr_2$ 和 $expr_3$ 是不同的类型，结果的类型依照本章先前讨论的转换规则确定。例如，如果 f 是 float 型而 n 是 int 型，那么表达式

```
(n > 0) ? f : n
```

无论 n 是否为正，其类型都为 float。

上述条件表达式中第一个表达式的括号是不必要的，因为 ?: 的优先级非常低，它只比赋值运算符高一级。但是我们仍建议加上括号，因为它们让表达式的条件部分显得更加清晰。

用条件表达式常常能编写出简洁的代码。例如下面这个循环，它打印数组中的 n 个元素，每 10 个元素一行，各列用一个空格分开，并且每行（包括最后一行）以换行符结束。

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

在每 10 个元素或第 n 个元素之后会打印换行符，所有其他元素之后都带有一个空格。这可能看起来比较复杂，但它比与之等价的 if-else 要更加紧凑。另一个较好的例子是

```
printf("You have %d item%s.\n", n, n==1 ? "" : "s");
```

练习 2-10. 重写 lower 函数，它将大写字母转换为小写字母，使用一个条件表达式来替代 if-else。

2.12 算符优先级与求值次序

表 2-1 总结了所有运算符的优先级和结合率，包括那些我们还未讨论过的运算符。同一行中的运算符拥有相同的优先级；各行按优先级递减的次序排列。例如，*、/、% 具有相同的优先级，它们的优先级比二元运算符 + 和 - 要高。"运算符" () 代表函数调用。运算符 -> 和 . 用于访问结构的成员；它们以及运算符 sizeof（给出对象大小）将在第 6 章中介绍。第 5 章讨论 *（通过指针间接访问）和 &（取对象的地址）。第 3 章讨论逗号运算符。

表 2-1 运算符优先级和结合率

运算符	结合率
() [] -> .	从左至右
! ~ ++ -- + - * & (类型) sizeof	从右至左
* / %	从左至右
+ -	从左至右
<< >>	从左至右
< <= > >=	从左至右
== !=	从左至右
&	从左至右
^	从左至右
|	从左至右
&&	从左至右
||	从左至右
?:	从右至左
= += -= *= /= %= &= ^= |= <<= >>=	从右至左
,	从左至右

一元运算符 +、-、* 和 & 的优先级高于相应的二元运算符

注意按位运算符 &、^ 和 | 的优先级低于 == 和 != 。这意味着像下面这样的位测试表达式

```
if ((x & MASK) == 0) …
```

必须括上括号才能得到正确的结果。

与大多数语言类似，C 语言并未指定运算符的多个操作数的求值次序。（&&、||、?: 以及 , 是例外。）例如，类似

```
x = f() + g();
```

这样的语句中，f 的求值可能在 g 之前，也可能在 g 之后；因此如果 f 或者 g 将改变另一方所依赖的变量，那么 x 的值就会取决于它们求值的次序。可以将中间结果存放在临时变量中以确保特定的求值顺序。

类似地，函数的参数之间的求值次序也是不确定的，因此语句

```
printf("%d %d\n", ++n, power(2, n));     /* 错了！ */
```

用不同的编译器能产生不同的结果，这取决于 n 的递增在 power 调用之前还是之后完成。显然，其解决办法是写成：

```
++n;
printf("%d %d\n", n, power(2, n));
```

函数调用、嵌套的赋值语句以及递增和递减运算符会引起"副作用"——对表达式的求值会附带造成某些变量的改变。在任何带有副作用的表达式中，对组成表达式的变量的求值次序存在着微妙的依赖关系。下面这个语句是一种不爽情形的典型代表：

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

其问题在于数组下标究竟是 i 的旧值还是新值。编译程序对此可以有不同的解释，并根据不同解释产生不同的结果。C 语言标准有意留下了诸多此类问题未作具体规定。表达式中的副作用（即对变量赋值）何时发生留待编译器自己考虑，因为最好的求值次序取决于具体机器的架构。（标准明确规定了函数参数的所有副作用都必须在该函数被调用之前生效，但是这对于上面 printf 函数调用里的情况没有帮助。）

就风格而言，编写依赖于求值次序的代码对于任何语言都不是一种好的编程习惯。很自然，知道哪些事情不能做是必要的，但如果不知道某些事情在各种机器上会如何完成，同样也不要试图去利用某种（可能依赖于具体机器的）特定实现来做它们。

第 3 章 控制流

编程语言中的控制流语句用于规定计算指令的执行顺序。在前面的例子中我们已经遇到过最常见的一些控制流结构；本章会对所有控制流结构进行讨论，并对之前已讨论的部分给出更精确的描述。

3.1 语句和程序块

一个表达式（譬如 x = 0 或 i++ 或 printf(…)）跟上一个分号之后就变成了一个语句，如

```
x = 0;
i++;
printf(…);
```

在 C 语言中，分号是一个语句结束符，而不像在 Pascal 等语言中那样是一个语句分隔符。

花括号 { 和 } 将多个声明和语句合到一起，构成一个复合语句（或称程序块），这样多个声明和语句在句法上等同于单个语句。函数里包含所有语句的花括号是一个明显的例子；跟在 if、else、while 以及 for 之后包含多个语句的花括号则是另一个例子。（在任意程序块中都能声明变量，这一点会在第 4 章中探讨。）表示一个程序块结束的右花括号之后不带分号。

3.2 If-Else

if-else 语句用于表述条件判定，其正式语法为

```
if (表达式)
    语句 1
else
    语句 2
```

其中 else 部分是可选的。表达式会被求值，如果它为真（即如果表达式的值不为零），执行语句 1；若为假（表达式值为零）且有 else 部分，则执行语句 2。

由于 if 只是简单地测试表达式的数值，因而可以采用某些代码简写形式。最明显的是以

```
if (表达式)
```

来替代

```
if (表达式 != 0)
```

有时候这样简写清楚自然；而有时候又会让人难于理解。

由于 if-else 中的 else 部分是可选的，因此当嵌套的 if 序列中省略某个 else 时会引起歧义。歧义的消除通过将 else 与之前离它最近且无 else 配对的 if 相匹配来完成。例如在下面的代码中，

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

如缩进结构所示，else 语句与内部的 if 匹配。如果那不是你想要的，那么必须使用花括号来强制得到期望的匹配：

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

类似下面的情况中这种歧义性尤为有害：

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("…");
            return i;
        }
else        /* 错了! */
    printf("错误 -- n是一个负数\n");
```

其缩进结构明确地表达了编写者希望的做法，但编译器无法得到这一信息，而将程序中的 else 与内部的 if 相匹配。这种错误会很难发现；因此当存在嵌套的 if 时使用花括号是一个好主意。

顺便提醒读者注意，代码

```
if (a > b)
    z = a;
else
    z = b;
```

中在 z = a 之后有一个分号，这是因为语法上 if 之后带有一个语句，而像"z = a;"之类的表达式语句总是以分号来结尾的。

3.3 Else-If

程序结构

```
if (表达式)
    语句
else if (表达式)
    语句
else if (表达式)
    语句
else if (表达式)
    语句
else
    语句
```

相当常见，因而值得单独拿出来简要讨论一下。这种 if 语句序列是编写多路判定的最普通的方式。其中的*表达式*被依次求值；如果某一*表达式*为真，那么与之相对应的*语句*就被执行，并且整个判定序列终止。就像一直以来的那样，每个*语句*的代码要么是单个语句，要么是用花括号括上的一组语句。

最后的 else 部分用于处理"以上均不是"即所有其他条件都不满足时的缺省情况。有时无需显式的缺省处理，那样序列尾部的

```
else
    语句
```

可被省略掉，或者可以用作错误检测来捕获"不可能的"条件。

这里以一个二分查找函数为例来说明三路判定的用法，该函数判定在已排好序的数组 v 中是否存在一个特定的值 x。这里 v 的元素必须以递增次序排列。如果 v 中存在 x，则函数返回 x 在 v 中的位置（介于 0 与 n-1 之间的一个数），否则返回 -1。

二分查找法首先将输入值 x 与数组 v 的中间元素进行比较。如果 x 小于中间值，那么集中查找数组的前半部分，否则查找数组的后半部分。无论哪种情况，下一步都是将 x 与所选那一半的中间元素进行比较。这种将查找范围一分为二的过程会一直持续到找到输入值或者查找范围变空为止。

```
/* binsearch: 在 v[0] <= v[1] <= … <= v[n-1] 中查找 x */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else   /* 找到匹配 */
            return mid;
    }
    return -1;    /* 无匹配 */
}
```

此程序的主要判定操作是每一步中 x 是否小于、大于还是等于中间元素 v[mid]；这正是 else-if 所擅长的工作。

练习 3-1. 我们的二分查找程序在循环内部进行了两次测试，其实一次测试就足够了（代价是在循环之外要有更多的测试）。编写在循环内只有一次测试的版本，并比较两者运行时间的差别。

3.4 Switch

switch 语句是一种多路判定，它测试一个表达式是否与一组整型常量中的某个值相匹配，并根据匹配情况执行程序分支。

```
switch (表达式) {
    case 常量表达式:  语句序列
    case 常量表达式:  语句序列
    default:  语句序列
}
```

每个 case 带有一个或多个具有整型值的常量或者常量表达式的标签。如果某个 case 与待判定表达式的值相匹配，则程序从该 case 开始执行。所有 case 的表达式（的值）必须互不相同。在所有其他 case 都不满足匹配时，标记为 default 的 case 被执行。default 是可选的；如果它不存在，并且没有匹配的 case，则任何操作都不会发生。所有的 case 以及 default 项可以按任意的次序排列。

在第 1 章中我们编写了一个程序，它使用一连串 if … else if … else 来统计每个数

字、空白符、以及所有其他字符出现的次数。这里的程序使用一个 switch 语句完成相同的功能：

```
#include <stdio.h>

main()    /* 统计数字、空白符、及其他字符数 */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
    return 0;
}
```

break 语句使得程序的执行立即从 switch 中退出。因为 case 只起到标签的作用，因此在某 case 的对应代码执行完毕后，如果编程者没有采取显式的退出操作，执行将**穿透**到接下来的代码。break 和 return 是退出 switch 最常用的手段。break 语句还可用于强制从 while、for 和 do 循环中立即退出，这将在本章后面的部分讨论。

穿透特性是一把双刃剑。好的方面是它允许多个 case 关联到单个操作，就如这个例子中对数字的处理。但它同时隐含的是普通情况下每个 case 都必须以 break 结束，以免穿透到下一个 case。从一个 case 穿透到另一个 case 的特性不够健壮，程序在修改时其完整性容易被破坏。除了多个标签对应单一计算处理的情况之外，穿透应当少用，在使用时也应加上注释。

就编程形式而言，在最后的 case（在这里是 default）后面放置一个 break 语句是好的做法，尽管这在逻辑上是没有必要的。某一天当在后面增加其他 case 时，这一丁点保护性的编码可能会帮上大忙。

练习 3-2. 编写一个函数 escape(s,t)，其将字符串 t 复制到字符串 s 中，并在复制过程中将换行符和制表符之类的字符转换为可见的转义字符序列，如 \n 与 \t。请使用一个 switch 语句。再编写一个反向转换函数，其将转义字符序列转换为实际的字符。

3.5 While 循环和 For 循环

while 循环和 for 循环我们已经见到过。while 循环

```
while (表达式)
    语句
```

对其中的表达式求值，如果值不为零，则执行语句并再次对表达式求值。重复这一过程直到表达式变为零为止，然后程序从语句之后继续执行。

除 continue 的行为之外，for 循环语句

```
for (表达式1; 表达式2; 表达式3)
    语句
```

等价于

```
表达式1;
while (表达式2) {
    语句
    表达式3;
}
```

continue 语句在 3.7 节说明。

语法上 for 循环的三个组成都是表达式，最常见的情况是：表达式 1 和表达式 3 是赋值语句或函数调用，而表达式 2 是一个关系表达式。三者中的任何一个都可被省略掉，但分号必须保留。如果表达式 1 或表达式 3 被省略，执行会简单地穿越它们；测试部分（即表达式 2）如果未给出，则被视为永真，因此

```
for (;;) {
    …
}
```

是一个"无限"循环。它大概会以譬如 break 或者 return 等其他的方式终止。

用 while 还是用 for 在很大程度上是一个个人喜好问题。例如，代码

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;   /* 略过空白字符 */
```

中没有初始化或重新初始化操作，因此使用 while 最自然。

当存在简单的初始化和递增操作时更倾向于使用 for，因为它让循环控制语句集在一起且在循环顶部就可被看见。这一点在代码

```
for (i = 0; i < n; i++)
    …
```

中最为明显。这是 C 语言处理某个数组前 n 个元素的惯用方法，相当于 Fortran 语言的 DO 循环或者 Pascal 语言的 for 循环。但该类比并不完全对应，因为 C 的 for 循环可以在其内部改变循环的索引和边界，且当循环由于任何原因而终止时，索引变量 i 的值都可保留下来。由于 for 的组成部分是任意的表达式，因此 for 循环并非仅限于使用算术级数表达式。尽管如此，将不相干的计算指令强加到 for 的初始化和递增部分是一种坏的编程风格，将这些部分留给循环控制操作会更好。

更大些的例子是另一版本的 atoi 函数（其将字符串转换为其表示的数字）。这个版本比第 2 章中的版本稍微通用一些；它能处理可能存在的打头的空白符和一个+号或者-号。（第 4 章给出的 atof 函数对浮点数进行相同的转换处理。）

程序的结构反映了输入形式：

```
若有空白符，略过它们
若有符号，获取符号
得到整数部分并进行转换
```

每个步骤处理数字的一部分，并将状态干净的内容留给下一步处理。当遇到第一个不可能是数字部分的字符时，整个处理过程结束。

```
#include <ctype.h>

/* atoi: 将 s 转换为整数; 版本 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)   /* 跳过空白字符 */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')   /* 跳过符号 */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

标准库提供了一个更精巧的函数 strtol，其用于将字符串转换为长整数；参见附录 B 的第 5 节。

当存在多层嵌套循环时，让循环控制集中起来的优点愈加显著。下面的希尔排序函数用于整数数组的排序。该排序算法由 D.L.Shell 于 1959 年发明，其基本思想是：在初始阶段对相隔较远的元素进行比较，而不是像简单交换排序那样比较相邻的元素。这样有助于快速消除大量乱序情况，从而减少后续阶段的工作量。比较元素的间隔会逐渐递减为 1，此刻的排序实质上已变成一种相邻交换的方法。

```
/* shellsort: 将 v[0]…v[n-1]按递增次序排序 */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

这里有三个嵌套的循环，最外层循环控制比较元素的间隔，间隔开始为 n/2，每轮除以 2 直至变成零为止。中间的循环对元素依次处理。最内层循环以 gap 为间隔比较每对元素，并将逆序元素颠倒过来。由于 gap 最后递减为 1，因此所有元素最终会被正确地排序。请注意 for 的通用性是如何使得外层循环具有与其他循环相同的形式的，虽然其控制变量并不是一个等差级数。

最后的一个运算符是逗号运算符“,”，其最常见于 for 循环语句中。逗号运算符分开的一对表达式被从左至右求值，运算结果的类型和值就是右操作数的类型和值。这样就可能在一个 for 语句中的各部分设置多个表达式，例如对两个索引进行并行处理。下面以函数 reverse(s) 为例来说明这一点，它将字符串 s 本身颠倒过来。

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

```
/* reverse: 将字符串 s 本身颠倒过来 */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

用于将函数参数以及声明中的变量等分隔开的逗号不是逗号运算符，也不能保证按从左至右的顺序求值。

逗号运算符应该少用。其最适合用于相互关系较紧密的代码结构，就如 reverse 函数中的 for 循环，以及需要多个计算步骤的单个表达式宏。逗号表达式也可能适用于 reverse 函数中的元素互换，这里互换可被当成是单个操作：

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

练习 3-3. 编写一个 expand(s1, s2)，它将字符串 s1 中类似 a-z 的速记符在 s2 中扩展为等价的完全列表 abc…xyz。函数支持大小写字符和数字，并可处理 a-b-c、a-z0-9 以及-a-z 之类的用例。打头和结尾的 - 按普通字符处理。

3.6 Do-while 循环

像我们在第 1 章中讨论的那样，while 循环和 for 循环在其顶部测试终止条件。与之相反，C 语言中的第 3 种循环——do-while——每次执行完循环体之后在底部进行测试；它的循环体至少会执行一次。

do 循环的语法为

```
do
    语句
while (表达式);
```

其中语句执行之后表达式才被求值。如果表达式为真，则语句被再次执行，重复这一过程；当表达式变为假时，循环终止。除了测试的含义不同之外，do-while 等价于 Pascal 的 repeat-until 语句。

经验表明 do-while 的使用比 while 和 for 要少得多，但是不时仍会用到它，就像下面的 itoa 函数那样。该函数将一个数字转换为对应的字符串（与 atoi 函数相反）。此任务可能会比一开始想象的要稍微复杂一些，因为较容易的生成数字的方法是以相反的次序生成它们的。我们选用的做法是先生成反向的字符串，然后再将它颠倒过来。

```
/* itoa: 将 n 转换为字符形式存放到 s 中 */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0) /* 记录符号 */
        n = -n;         /* 使 n 变为正数 */
```

```
    i = 0;
    do {       /* 以反向次序生成数字 */
        s[i++] = n % 10 + '0';  /* 得到下一个数 */
    } while ((n /= 10) > 0);       /* 删除该数 */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

这里的 do-while 是必要的，或者至少是方便的，因为即使是 n 为零的情况，仍至少有一个字符必须放到数组 s 中。我们还用花括号括上了构成 do-while 循环体的单个语句，虽然这不是必要的，但这样做不会让某些粗心的阅读者将 while 部分误认为是一个 while 循环的开始。

练习 3-4．按照二进制的补码表示法，我们的 itoa 版本不能处理最大的负数，即 n 的值等于 $-(2^{字长-1})$。请解释为什么不能。请修改函数，使它无论运行在何种机器上在都能正确地打印该值。

练习 3-5．编写函数 itob(n, s, b)，它将整数 n 转换为以 b 为底的数所对应的字符表示并放到字符串 s 中。特别地，itob(n, s, 16)将 n 按十六进制数的格式转换到字符串 s 中。

练习 3-6．编写函数 itoa 的另一版本，它接受的参数是三个而不是两个。第三个参数是最小字段宽度；如果必要，转换后的数的左边必须被填充空格以达到足够的宽度。

3.7 Break 和 Continue

如果能通过顶部或底部测试以外的方式退出循环，有时会比较方便。break 语句提供了从 for、while 和 do 中提前退出的功能，这与它在 switch 中的作用一样。break 会使得最内层的闭合循环或 switch 立即退出。

下面的函数 trim 用于把一个字符串末尾的空格、制表符和换行符去掉；当找到最右端的一个不是空格、制表符及换行符的字符时，它使用一个 break 语句从循环中退出。

```
/* trim: 删除字符串末尾的空格、制表符和换行符 */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        for (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

函数 strlen 返回待处理字符串的长度。程序中的 for 循环从该字符串的末尾向前扫描，查找第一个不是空格、制表符或换行符的字符。当找到一个字符，或者当 n 变为负值时（即当整个字符串都已扫描完毕时），循环退出。读者应能验证，即使是在字符串为空或只包含空白符的情况下，这样的处理仍然是正确的。

continue 语句与 break 语句有关联，但它没有 break 语句常用。continue 语句使得 for、while 或者 do 循环开始执行下一轮循环处理。在 while 和 do 中，这意味着立即执行循环的测试部分；在 for 中，则意味着循环控制转移到递增步骤。continue 语句只能用于循

环，而不能用于 switch。在某个循环内的 switch 中的 continue 语句会导致循环进入下一轮处理。

举个例子，下面的程序片段中只对数组 a 中的非负元素进行处理；负值则被跳过。

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)    /* 跳过负元素 */
        continue;
    …   /* 处理非负元素 */
}
```

continue语句常用于循环中其后的部分较为复杂的情况，如果不用continue语句，则需要反向测试，并缩进一级处理，这可能会导致程序嵌套得过深。[①]

3.8 Goto 与标签

C 语言提供了极易被滥用的 goto 语句，及其跳转所用的标签。正规说来，goto 语句没有必要存在。在实践中，绝大多数情况下编写不含 goto 语句的代码都比较容易。我们在本书中没有使用过 goto 语句。

然而，有几种情况 goto 语句可以适用。最常见的是需要取消位于某些较深的嵌套结构中的处理，例如从两层或者更深的循环中一次性退出来。这种情况不能直接使用 break 语句，因为它只能退出最内层的循环。这样：

```
        for ( … )
            for ( … ) {
                …
                if (disaster)
                    goto error;
            }
        …

    error:
        错误处理部分
```

如果相应的错误处理代码不是很少，而且错误会在好几个地方发生，那么这种组织方式会比较便捷。

标签具有与变量名一样的形式，但其后带有一个冒号。它可指向对应 goto 语句所在函数的任一语句。标签的作用域就是它所在的函数。

作为另一个例子，考虑判定两个数组 a 与 b 是否存在相同元素的问题。一种可能的做法是：

```
        for (i = 0; i < n; i++)
            for (j = 0; j < m; j++)
                if (a[i] == b[j])
                    goto found;
        /* 没有找到相同的元素 */
        …
    found:
        /* 找到一个: a[i] == b[j] */
        …
```

包含 goto 语句的代码总能编写成完全不用 goto 语句的代码，尽管那样可能会增加一些重复测试或一个额外变量作为代价。例如，上述数组查找代码可改写为：

[①]这句话的理解，考虑按 a[i] >= 0 {怎样怎样} 编写时的情况。译注

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* 找到一个: a[i-1] == b[j-1] */
    …
else
    /* 没有找到相同的元素 */
    …
```

除了与这里所举例子类似的少数情况之外，使用 goto 语句的代码通常比没有 goto 语句的代码更加难于理解和维护。尽管我们对待这个问题的态度并不武断，但是我们确实认为应该尽可能少地使用 goto 语句。

第 4 章 函数与程序结构

函数将大的计算任务分解成若干小任务，并使得人们可在已有程序的基础上构造程序，而无须从头做起。一个设计得当的函数可以将其余程序不需知道的操作细节隐藏起来，从而使得整个程序更加清晰，程序改动的难度也得到降低。

C 语言的设计中考虑了函数的高效性和易用性。C 程序通常由许多小函数（而不是少量大函数）组成。一个程序可以放在一个或多个源文件中，各个源文件可以分别编译，并与库中事先编译好的函数一并进行加载。在这里我们不打算深入到这一过程，因为不同系统的编译加载细节互有差异。

ANSI 标准对 C 语言所做的最明显的修改是函数声明与函数定义两方面。正如第 1 章中所见，C 语言现在允许在声明函数时声明参数的类型。函数定义的语法也有相应改变，以使函数声明和定义彼此匹配。这使得编译器可以检测出比原先更多的错误。此外，这还使得参数在说明得当的情况下能够自动进行适当的强制类型转换。

ANSI 标准进一步明确了名字作用域规则，特别是要求每个外部变量只能有一个定义。初始化的范围变得更广：现在自动数组变量和自动结构变量都可以进行初始化。

C 语言预处理器的功能也得到了增强。新的预处理器工具中包含了更完整的条件编译指令集、一种通过宏参数创建带引号字符串的方法、以及对宏扩展过程的更好的控制。

4.1 函数基础知识

首先我们来设计并编写一个程序，它将输入中包含特定“模式”或字符串的各行打印出来（这是 UNIX 程序 grep 的一个特例）。例如，在下面一组文本行中查找包含字母字符串“ould”的行：

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

将产生如下输出：

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

此任务可以简洁地分为三部分：

```
while  ( 还有未处理的行 )
    if  ( 该行包含指定的模式 )
        打印该行
```

尽管将这些代码全都放到主程序 main 中肯定是可以的，但更好的方式是利用上述结构将每一部分设计成一个独立的函数。处理三个较小的部分要比处理一个大的整体更加容易，因为这样能够将不相关的细节隐藏在函数中，从而使不必要的相互影响的机会降到最低。并且，这些函数还可能用于其他的程序。

我们用函数 getline 来实现“还有未处理的行”，这个函数已在第 1 章介绍过；用 printf 函数来实现“打印该行”，这个函数别人已经提供了；这意味着我们只需要编写一个判定“该行

包含指定的模式”的函数。

我们编写函数 strindex(s, t) 来解决问题。该函数返回字符串 t 出现在字符串 s 中的起始位置或下标。当 s 中不包含 t 时，则返回-1。由于 C 语言数组的下标从 0 开始，下标的值为 0 或正数，因此适于用 -1 之类的负数来表示失败的情况。如果以后需要更复杂的模式匹配，只需替换 strindex 函数即可，程序的其余部分可以保持不变。（标准库中提供的库函数 strstr 的功能类似于函数 strindex，只是该库函数返回的是指针而不是下标。）

有了这些设计，编写程序的细节内容就很直接了。下面是完整的程序，读者可以看到这几部分是怎样结合在一起的。目前查找的模式是字符串文本，这不是一种最通用的机制。关于字符数组的初始化方式很快会回过来讨论。第 5 章会介绍如何使模式成为程序运行时可设定的参数。getline 函数的版本也稍有不同，读者可将它与第 1 章中的版本进行比较，或许能从中得到一些启发。

```
#include <stdio.h>
#define MAXLINE 1000    /* 最大输入行长度 */

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";   /* 待查找的模式 */

/* 找出所有与模式相匹配的行 */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: 取一行放到 s 中，并返回该行的长度 */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: 返回 t 在 s 中的位置，若未找到则返回-1 */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
```

```
            for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
                ;
            if (k > 0 && t[k] == '\0')
                return i;
        }
        return -1;
    }
```

每个函数定义均为如下形式：

```
返回类型 函数名(参数声明)
{
    声明和语句
}
```

其中多个部分可以省略，最简单的函数是：

```
dummy( ) { }
```

这个函数什么也不做，也不返回任何值。像这样的"无为"函数有时在程序开发期间被用来预留位置。如果函数定义中省略了返回类型，则默认为 int 类型。

程序可以看成变量定义和函数定义的集合。函数之间通过参数、函数返回值以及外部变量进行交互。函数可按任意次序出现在源文件中，源程序可分成多个文件，只要不将一个函数分在几个文件中就行。

return 语句用于从被调函数向调用者返回值，return 之后可以跟任何表达式：

```
return 表达式;
```

必要时表达式会被转换为函数的返回类型。表达式常常被括上一对圆括号，但这不是必需的。

调用函数可以自己决定是否忽略掉返回值。而且，return 也不必一定带有表达式，在不带表达式的情况下，没有值返回给调用者。而当被调用函数因执行到最后的右花括号而从"尾部脱离"时，控制权也会不带值地返回给调用者。如果函数在一处调用时有返回值而在另一处调用时没有返回值，这并不非法，但这可能是函数存在问题的征兆。在任何情况下，如果一个函数没有成功地返回一个值，那么它的"值"肯定是无用的。

上面的模式查找程序从 main 中返回一个状态，即所匹配的数目。这个值可以供调用该程序的环境使用。

不同系统对存放于多个源文件的 C 程序的编译与加载机制互有差别。例如，UNIX 系统上使用第 1 章中提到的 cc 命令来完成这项工作。假定这三个函数分别存放在名为 main.c、getline.c 与 strindex.c 的三个文件中，那么命令

```
cc main.c getline.c strindex.c
```

编译这三个文件，并将目标代码存放在文件 mail.o、getline.o 与 strindex.o 中，然后再将这三个文件一起加载为可执行文件 a.out。如果有一个错误，比如说出现在文件 main.c 中，那么可以用命令

```
cc main.c getline.o strindex.o
```

对 main.c 文件重新编译，并将编译结果与之前的目标文件 getline.o 和 strindex.o 一起加载。cc 命令使用 .c 与 .o 这两个命名约定来区分源文件与目标文件。

练习 4-1. 编写函数 strrindex(s, t)，其返回字符串 t 在 s 中最右边出现的位置，如果 s 中不包含 t，则返回-1。

4.2 返回非整型值的函数

到目前为止，我们的例子函数都是没有返回值（void）或者返回 int 类型值的函数。假如某个函数必须返回其他类型的值，那该怎么做呢？许多数学函数如 sqrt、sin 与 cos 等返回的是 double 类型的值；另一些专用函数则返回其他类型的值。为了说明让函数返回非整数值的方法，我们编写并使用函数 atof (s)，该函数将字符串 s 转换成相应的双精度浮点数。atof 函数是 atoi 函数的扩充，第 2 章与第 3 章已讨论过 atoi 函数的几个版本。atof 函数要处理可选的符号与小数点，并要考虑可能缺少整数部分或小数部分的情况。我们这个版本并不是一个高质量的输入转换函数，它所占用的空间应能更节省一些。标准库中包含了一个 atof 函数，它在头文件<stdlib.h>中声明。

首先，由于 atof 函数的返回类型不是 int，因此必须声明其返回值的类型。该类型名称位于函数名称之前：

```
#include <ctype.h>

/* atof: 把字符串 s 转换成相应的双精度浮点数 */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)   /* 略过空白符 */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

第二，也同样重要的是，调用程序必须知道 atof 函数返回的是非整数值。确保这一点的一种方法是在调用程序中显式声明 atof 函数。下面的原始计算器程序（其仅适用于支票簿核算）中给出了这个声明。程序按行每次读入一个数（每行一个数，数的前面可能带有正负号），并将这些数加在一起，并在每次输入之后将当前的总和打印出来：

```
#include <stdio.h>

#define MAXLINE 100

/* 原始计算器 */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);
```

```
    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

其中，声明

```
double sum, atof (char []);
```

表明 sum 是一个 double 类型的变量，atof 是一个函数，该函数带有一个 char[] 类型的参数并返回一个 double 类型的值。

函数 atof 的声明与定义必须一致。如果在同一源文件中，atof 函数与 main 中对它的调用具有不一致的类型，那么编译器将会检测出这个错误。但是，假如 atof 函数是独立编译的（这种可能性更大），那么这种不匹配的错误就不会被检测出来，atof 函数会返回 double 类型的值，而 main 函数则会将其按 int 类型处理，最终得到的结果就会毫无意义。

按照之前我们关于声明是如何必须与定义相匹配的讨论，这似乎很令人吃惊。不匹配现象会发生的缘由是，如果没有函数原型，则该函数会在表达式中首次出现时被隐式声明，譬如：

```
sum += atof(line)
```

如果某个之前没有声明过的名字出现在某个表达式中，并且紧跟着一个左圆括号，那么根据上下文它会被认为是一个函数名字的声明，该函数的返回值被假定为 int 类型，而对其参数则不做任何假定。而且，如果一个函数声明不包含参数，就像

```
double atof();
```

那么也同样意味着对于 atof 函数的参数不做任何假定，所有的参数检查都被关闭。空参数表的这种特殊意义是为了使较老的 C 程序可以被新的编译器编译。但是，在新程序中也这样做是不明智的。如果函数有参数，请声明它们；如果没有参数，请使用 void。

有了正确声明的 atof 函数，我们可以基于它来编写 atoi 函数（将字符串转换为 int）：

```
/* atoi: 利用 atof 函数把字符串 s 转换为整数 */
int atoi(char s[])
{
    double atof(char s[]);

    return (int) atof(s);
}
```

请注意其中的声明和 return 语句的结构。在语句

```
return 表达式;
```

中*表达式*的值在返回之前会被转换为所在函数的类型。因此，当函数 atof 出现在上述程序的 return 语句中时，其返回的 double 类型的值将被自动转换为 int 类型。但是，这一操作有丢失信息的可能，因此一些编译器会对此给出警告。而此函数中的强制转换显式地表明了所要进行的操作，屏蔽掉了相关的警告信息。

练习 4-2. 对 atof 函数进行扩充，使之可以处理形如

```
123.45e-6
```

的科学计数法，即一个浮点数，其后可能紧跟 e 或 E 及一个（可能带正负号的）指数。

4.3 外部变量

C 程序由一组外部（external）对象构成，或者是变量或者是函数。形容词"external"与"internal"相对，internal 用于描述在函数内定义的参数和变量。外部变量在函数之外定义，因而有可能被多个函数使用。函数自身一定是外部的，这是因为 C 语言不允许在函数中定义其他函数。默认情况下，外部变量与函数具有如下性质：通过相同名字引用的外部对象，所引用的是同一对象实体，即便引用它们的是独立编译的函数也是如此。（标准将这种性质称为**外部链接**）。在这个意义上，外部变量类似于 Fortran 语言的 COMMON 块或 Pascal 语言中在最外层程序块中定义的变量。我们将在后面见到如何定义只在单个源文件内可见的外部变量和函数。

由于外部变量是全局可访问的，这就为函数之间的数据交换提供了一种可以替代函数参数及返回值的方法。任何函数都可以通过外部变量的名字来访问该变量，只要这个名字已通过某种方式进行了声明。

如果必须在函数之间共享大量的变量，使用外部变量要比使用长参数表更加方便和高效。然而，正如第 1 章中所指出的，这样做必须保持谨慎，因为这可能会对程序结构造成不利的影响，并可能导致程序中的函数出现过多的数据关联。

外部变量的用途还体现在它们更大的作用域和更长的生存期。自动变量只在函数内部起作用；变量从其所在函数被调用时开始存在，到函数退出时消失。而外部变量是永久存在的，它们的值在不同的函数调用之间可以保持住。因此，如果两个函数必须共享某些数据而互不调用对方，那么将这些共享数据存放在外部变量中（而不是通过参数传入传出）常常相当方便。

我们通过一个更大的例子来验证这一点。该例子是编写一个提供加（+）、减（-）、乘（*）、除（/）四则运算符的计算器程序。为了更易于实现，在计算器中使用逆波兰表示法来替代普通的中辍表示法（逆波兰表示法被用于某些袖珍计算器中，以及如 Forth 和 Postscript 等一些语言上）。

在逆波兰表示法中，每个运算符都紧跟在它的操作数之后。如中辍表达式

```
(1 - 2) * (4 + 5)
```

用逆波兰表示法表示为：

```
1 2 - 4 5 + *
```

表达式不再需要圆括号；只要知道每个运算符需要的操作数的个数，逆波兰表示法就是明确的。

程序的实现很简单。每个到来的操作数都被压入栈中；当一个运算符到来时，从栈中弹出相应数目的操作数（对于二目运算符是两个），按该运算符进行运算，并将运算结果压回栈中。例如，在上述用例中，首先 1 和 2 被压栈，然后它们被替换为两者之差-1；接着 4 和 5 被压栈，然后被替换为两者之和 9；最后，栈中的-1 与 9 被替换为它们的积-9。当到达输入行的末尾时，弹出栈顶的值并将其打印出来。

程序的结构是一个循环，对每次循环出现的运算符或操作数执行相应的操作：

```
while (下一个操作符或操作数不是文件结束标识符)
    if (数)
        将它压栈
    else if (运算符)
        弹出操作数
        进行运算
        将结果压栈
    else if (换行符)
```

```
        弹出并打印栈顶的值
    else
        错误
```

栈的压入与弹出操作的代码很少，但加上错误检测与恢复操作之后，程序就会变得比较长，因此最好将它们设计为独立的函数，以免整个程序通篇包含重复的代码。另外还应该有一个单独的函数用来获取下一个输入的运算符或操作数。

到目前还没有讨论的一个主要设计决策是，栈放在哪里？即哪些函数可以直接访问它？一种可能是把它放在主函数 main 中，并将栈及其当前位置传递给执行压入或弹出操作的函数。但是，main 不需要知道控制栈的变量，它只进行压入和弹出操作。因此，我们决定将栈及其相关信息放在外部变量中供 push 与 pop 函数访问，而不供 main 来访问。

将上述概要方案译成代码很容易。设想这些程序全都放在一个源文件中，那么程序看上去会像这样：

```
一些#include
一些#define

用于 main 的函数声明

main() {…}

用于函数 push 和 pop 的外部变量

void push(double f) {…}
double pop(void) {…}

int getop(char s[]) {…}

供函数 getop 调用的函数
```

后面我们将会讨论如何将这些代码分为两个或多个源文件的情况。

主函数 main 是一个循环，该循环中包含了一个大的 switch 语句，处理不同种类的运算符与操作数；这里对 switch 的使用要比 3.4 节所示的例子更为典型。

```
#include <stdio.h>
#include <stdlib.h> /* 为了使用 atof() 函数 */

#define MAXOP    100    /* 操作数或运算符的最大长度 */
#define NUMBER '0'      /* 标示找到一个数 */

int getop(char []);
void push(double);
double pop(void);

/* 逆波兰计算器 */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
```

```
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if (op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: 除零溢出\n");
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: 未知命令 %s\n", s);
                break;
            }
        }
        return 0;
    }
```

由于 + 与 * 运算符满足交换律，因此弹出的操作数的运算次序无关紧要，但对于 - 与 / 则必须区分其左右操作数。下列语句

```
    push(pop() - pop());    /* 错了！ */
```

中对两次 pop 函数调用（的返回值）的运算次序没有区分清楚。为了确保正确的次序，有必要像 main 中所做的那样将第一个值弹出到某个临时变量中。

```
    #define MAXVAL 100  /* 栈 val 的最大深度 */

    int sp = 0;         /* 栈的下一个空闲位置*/
    double val[MAXVAL]; /* 存放值的栈 */

    /* push: 把 f 压入栈中 */
    void push(double f)
    {
        if (sp < MAXVAL)
            val[sp++] = f;
        else
            printf ("error: 栈满，不能将%g 压栈\n", f);
    }

    /* pop: 弹出并返回栈顶的值 */
    void pop(void)
    {
        if (sp > 0)
            return val[--sp];
        else {
            printf ("error: 栈空\n");
            return 0.0;
        }
```

```
}
```

一个变量如果在所有函数之外定义，那么它就是外部变量。因此我们把必须被函数 push 和 pop 共享的栈和栈顶索引定义在这些函数的外部。但 main 本身并没有引用该栈或栈顶位置，因此可对 main 隐藏掉这些内容。

现在我们来看函数 getop 的实现。该函数用于获取下一个运算符或操作数。这个任务比较容易：跳过空格与制表符；如果下一个字符不是数字或小数点，则返回；否则，将这些数字字符串收集起来（其中可能包含小数点），并返回 NUMBER——标示已经收集好了一个数。

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: 取下一个运算符或数值操作数 */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* 不是数字 */
    i = 0;
    if (isdigit(c))  /* 收集整数部分 */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')    /* 收集小数部分 */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

这里函数 getch 与 ungetch 的用途是什么？常有这样的情况，一个程序在读入过多输入之前无法判定是否已读了足够的内容。一个例子是在收集用于组成数的字符时，直到看见第一个非数字字符后，才能确定数被完整地读入了。但这时程序已经向前多读了一个字符，而该字符不是它所要的内容。

假如可能将这个不需要的字符"反读"回去，那么这个问题就能得到解决。那样，每次程序多读了一个字符时，就能将它压回到输入中，对其余代码而言就像这个字符并没有被读过一样。幸运的是，模拟反取一个字符比较容易，可以编写一对相互配合的函数来完成：函数 getch 根据情况传递下一个输入字符；而函数 ungetch 记下那些要放回输入中的字符，这样在对 getch 的后续调用中它会首先返回那些记下的字符，然后再读入新的输入。

两者的协作机制也很简单。函数 ungetch 将那些要压回的字符放入一个共享缓冲区（一个字符数组），函数 getch 在该缓冲区不为空时就从中读取字符，而在缓冲区为空时调用函数 getchar 从输入中读字符。同时这里还必须有一个下标变量来记录缓冲区中当前字符的位置。

由于缓冲区与下标被函数 getch 与 ungetch 所共享，并且其值必须在这些函数调用之间保持住，因此它们必然是这两个函数的外部变量。这样，函数 getch 与 ungetch 及其共享变量可以编写为：

```
#define BUFSIZE 100

char buf[BUFSIZE];  /* 供unget函数使用的缓冲区 */
int bufp = 0;           /* buf中的下一个空闲位置 */

int getch(void)  /* 取一个字符（可能是压回的字符）*/
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* 将字符压回到输入中 */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: 压回字符过多\n");
    else
        buf[bufp++] = c;
}
```

标准库中包含了函数 ungetc，其提供压回单个字符的功能，我们在第 7 章中将会讨论它。这里我们用了一个数组而没有用单个字符，目的是为了说明一种更为通用的方法。

练习 4-3. 有了基本框架后，对计算器程序的扩展就简单了。在该程序中加入取模（%）运算符并增加对负数的支持。

练习 4-4. 增加下列栈操作命令：在不弹出的情况下打印栈顶元素、复制栈顶元素、交换栈顶的两个元素。增加一个用于清空栈的命令。

练习 4-5. 增加对 sin、exp 与 pow 这类库函数的访问。参见附录 B 第 4 节中的<math.h>。

练习 4-6. 增加处理变量的命令（提供 26 个具有单个字母名称的变量很容易）。增加一个变量存放最近一次打印的值。

练习 4-7. 编写一个函数 ungets(s)，其用于将整个字符串压回到输入中。ungets 函数需要知道 buf 与 bufp 吗？它能否只使用 ungetch 函数？

练习 4-8. 假定最多只需要压回一个字符。请相应地修改 getch 与 ungetch 函数。

练习 4-9. 上面所介绍的 getch 与 ungetch 函数不能正确地处理压回的 EOF。请你决定在压回 EOF 时这两个函数应具有什么性质，然后实现你的设计。

练习 4-10. 另一种程序构建方法是使用 getline 函数读入整个输入行；这样就没必要使用 getch 与 ungets 函数了。请运用这一方法更新计算器程序。

4.4 作用域规则

构成 C 程序的函数和外部变量不需要全部一起编译；程序的源代码可以按多个文件存放（和编译），还可以从库中加载事先编译好的例程。这里边所要关心的问题是：

- 声明如何编写才能使变量在编译过程中被恰当地声明？
- 声明如何安排才能让程序的各个部分能够在程序加载时正确地关联起来？
- 声明如何组织才能只存在一份副本？
- 外部变量如何进行初始化？

我们通过把计算器程序重新组织成多个文件来讨论上述话题。从实际角度看，该计算器程序太小了，不值得对其进行划分，但是它能很好地说明在较大程序中出现的对应问题。

一个名字的作用域（scope）是指程序中可以使用该名字的部分。对于在函数开头声明的自动变量，其作用域是声明该变量名字的函数。在不同函数中声明的同一名字的局部变量彼此互不相关。对于函数的参数同样如此，参数在效果上可看作是局部变量。

外部变量或函数的作用域是从它们在文件中被声明的位置开始到该被编译文件的末尾为止。举例来说，如果 main、sp、val、push 以及 pop 定义在一个文件中，并按照前述程序中所示的顺序放置，即

```
main() { … }

int sp = 0;
double val[MAXVAL];

void push(double f) { … }

double pop(void) { … }
```

那么，对于变量 sp 和 val，在 push 和 pop 中不需要对它们进行声明就可以简单地通过名字来使用它们。但这两个名字在 main 中不可见，push 和 pop 本身在 main 中也不可见。

另一方面，如果一个外部变量在定义之前就需要被使用，或者该外部变量定义在与使用它的源文件不同的源文件中，那么相应的声明要强制性地使用关键词 extern [那么需要强制性地使用“extern”声明]。

弄清楚外部变量的声明和定义二者间的区别很重要。声明只是标明了变量的属性（主要是它的类型）；而定义还会为其留出存储空间。如果代码行

```
int sp;
double val[MAXVAL];
```

出现在所有函数之外，那么它们定义了外部变量 sp 和 val 并为其留出存储空间，同时还作为该源文件余下部分的变量声明。另一方面，代码行

```
extern int sp;
extern double val[];
```

为该源文件余下部分声明了 sp 是一个 int 型的数，val 是一个 double 型的数组（其大小在其他地方决定）。但它们没有创建这些变量或者为变量保留存储空间。

在源程序的所有组成文件之中，一个外部变量只能有一个定义；其他文件可以通过包含该变量的 extern 声明来访问它。（包含变量定义的文件中仍可以包含该变量的 extern 声明。）数组的大小必须在其定义中给出，但在 extern 声明中是可选的。

外部变量的初始化只会在变量定义时进行。

假定函数 push 和 pop 在某一文件中定义，而变量 val 和 sp 在另一文件中定义和初始化（尽管对于此程序这样的组织形式不太可能），那么为了将它们联系在一起，如下的定义和声明是有必要的：

文件 1 中：

```
extern int sp;
extern double val[];

void push(double f) { … }

double pop(void) { … }
```

文件 2 中：

```
int sp = 0;
```

```
double val[MAXVAL];
```

由于文件 1 中的 entern 声明位于函数定义之外且早于它们，因此可被用于所有这些函数；对于文件 1，这样一组声明就足够了。如果在一个文件中 sp 和 val 的定义位于其使用之后，那么该文件也需要以同样的方式进行组织。

4.5 头文件

现在考虑将计算器程序分为若干个源文件，就比如每一组成部分都比现在大得多时可能会划分成的那样。函数 main 会划分为一个文件，我们将其称为 main.c；push、pop 以及它们使用的变量划分为第二个文件——stack.c；getop 划分为第三个——getop.c。最后，getch 与 ungetch 划分为第四个文件，getch.c；之所以将后两个函数与其他程序分开，原因是在实际程序中它们可能来自于一个独立编译的库。

还有一件事情需要考虑——在这些文件间共享的定义和声明。我们希望尽可能将它们集中起来，这样在程序演变时只需要维护一个副本。相应地，我们将这些公共内容放到一个**头文件** calc.h 中，在必要时可以包含它。（#include 语句在 4.11 节中描述。）照此划分后的程序看起来像如下这样：

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    …
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    …
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    …
}
void ungetch(int) {
    …
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    …
}
double pop(void) {
    …
}
```

在每个文件只可访问它所需要的信息这一期望与维护更多头文件的难度更大这一现实之间

需要折衷考虑。对于那些中等规模的程序，用一个头文件包含所有那些需要被程序的不同部分共享的内容可能是最好的办法；那正是我们在这里所作的决策。而对于那些大得多的程序，则可能会需要更多的组织方式和更多的头文件。

4.6 静态变量

stack.c 中的变量 sp 和 val 以及 getch.c 中的变量 buf 和 bufp 仅供它们各自所在的源文件中的函数使用，而不希望被任何其他的程序访问。static 声明若应用于外部变量或者函数，则会将所声明对象的作用域限制在该被编译源文件余下的部分之内。这样外部 static 就提供了一种隐藏名字的方法，例如 getch-ungetch 组合中的 buf 和 bufp，为了能被共享，它们必须是外部的，但 getch 与 ungetch 的使用者则不应该见到它们。

静态存储通过在普通声明之前加上关键字 static 来进行说明。如果上述函数和变量在一个文件中编译，即如：

```
static char buf[BUFSIZE];  /* 供 ungetch 使用的缓冲区 */
static int bufp = 0;           /* buf 的下一个空闲位置 */

int getch(void) { … }

void ungetch(int c) { … }
```

那么其他例程都将无法访问 buf 和 bufp，并且这两个名字不会与同一程序的其他文件中的相同名字发生冲突。同样地，通过静态声明的方式，可以将 push 和 pop 用于栈操作的变量 sp 和 val 隐藏起来。

外部静态声明多用于变量，但也可用于函数。通常情况下，函数名是全局的，对于整个程序的任何部分都可见。但是如果一个函数被声明为静态的，那么它在包含该声明的文件之外是不可见的。

静态声明还可用于内部变量。内部静态变量就像自动变量一样，是某个特定函数的局部变量，但与自动变量不同，它们是一直存在着的，而不是随着每次函数调用而产生和消亡。这意味着内部静态变量提供了仅在单个函数内使用的、持续的存储空间。

练习 4-11. 修改 getop 函数，使其不再需要使用 ungetch 函数。提示：使用一个内部静态变量。

4.7 寄存器变量

register 声明用于提示编译器其声明的变量将被极频繁地使用。其思想是应将 register 变量放到机器的寄存器中，这样可能得到更小、执行速度更快的程序。但是编译器可以无条件地忽略掉这一提示。

register 声明的形式如下：

```
register int x;
register char c;
```

register 声明只能应用在自动变量和函数的形式参数上。后者的形式如下：

```
f(register unsigned m, register long n)
```

```
{
    register int i;
    …
}
```

受底层硬件实际情况影响，寄存器变量在实际应用中存在限制。每个函数只有少许变量可以被存放在寄存器中，且只允许一些特定类型的变量。然而，过多的寄存器声明并没有害处，这是由于关键字 register 在声明过多或者是声明不允许类型的情况下会被忽略掉。另外，无论一个寄存器变量是否真正存放在寄存器中，取这个变量的地址都是不可能的（取址是第 5 章所涵盖的话题）。对寄存器变量的数目和类型的限制因具体机器而不同。

4.8 程序块结构

C 语言不是 Pascal 等语言意义上的块结构语言，它不允许在函数中定义函数，但是，其允许在函数中按块结构的形式定义变量。变量的声明（包括初始化）不是只能跟在函数开头的左花括号之后，而是可以跟在任意引入复合语句的左花括号的后面。以这种方式声明的变量可以屏蔽掉位于该程序块之外的同名变量，且在与左花括号匹配的右花括号出现之前一直存在。例如，在程序

```
if (n > 0) {
    int i;     /* 声明一个新的变量 i */

    for (i = 0; i < n; i++)
        …
}
```

之中，变量 i 的作用域是 if 语句“为真”的分支；这个 i 与该程序块之外的任何 i 都不相关。在程序块中声明与初始化的自动变量在每次进入该程序块时都会进行初始化。静态变量只在首次进入程序块时进行初始化。

自动变量（包括形式参数）也会屏蔽掉与之同名的外部变量和函数。对于如下声明：

```
int x;
int y;

f(double x)
{
    double y;
    …
}
```

在函数 f 之内出现的 x 引用的是函数参数，是一个 double 型变量；而在函数 f 之外，x 引用的是 int 型的外部变量。变量 y 的情况同样如此。

就编程风格而言，最好避免使用那些会屏蔽外层作用域中名字的变量名；否则极易引起混淆和错误。

4.9 初始化

初始化的概念之前已多次提及，但一直是其他话题里的附带内容。本节在已讨论过各种存储类型的基础上总结一些初始化的规则。

在没有显式初始化的情况下，外部变量与静态变量默认被初始化为 0；而自动变量与寄存

器变量则具有未定义的初值（即无用数据）。

标量变量可在定义时被初始化，其通过在变量名后带一个等号和一个表达式来完成：

```
int x = 1;
char squote = '\'';
long day = 1000L * 60L * 24L; /* 一天的毫秒数 */
```

对于外部变量和静态变量，其初始化值必须是一个常量表达式；初始化只进行一次（概念上是在程序开始执行前完成）。对于自动变量和寄存器变量，每次进入其所在函数或程序块时都会对其进行初始化。

对于自动变量和寄存器变量，其初始化值并不限于常量：它可以是包含任意已定义值（甚至函数调用）的表达式。例如，3.3 节中的二分查找程序可以被编写为：

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    …
}
```

代替之前的

```
    int low, high, mid;

    low = 0;
    high = n - 1;
```

在效果上，自动变量的初始化相当于赋值语句的简写形式。倾向于哪种形式很大程度上是个人喜好问题。我们通常采用显式赋值的形式，因为声明中的初始化值更难察看，且离变量使用的位置更远。

数组的初始化可以通过在其声明后带上用花括号括上的一列用逗号分隔开的初始化值来完成。例如，以每月的天数来初始化数组 days：

```
int days [] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

当数组的大小缺省时，编译器会统计初始化值的个数来得出数组的长度（在本例中为 12）。

如果初始化值的个数少于数组既定的大小，外部、静态及自动数组变量所缺少的元素将被设置为零；个数多于既定大小则是错误的。不能指定单个初始化值对多个数组元素的重复初始化，也无法在不提供前面所有元素初始化值的情况下直接对数组中间的元素进行初始化。

字符数组的初始化是一个特例；其可以用一个字符串替代上述使用花括号和逗号的记法：

```
char pattern[] = "ould";
```

是下面这种更长但等价的语句的简写形式：

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

此例中，数组的大小为 5（4 个字符加上 1 个结束符'\0'）。

4.10 递归

C 函数可以递归调用，即一个函数可以直接或间接地调用它自己。考虑将一个数打印为字符串的情况，正如之前提到的，数字生成次序的不对：先得到的是低位数字，然后才是高位数字；而它们必须以相反的次序打印出来。

解决这个问题有两种方法。一种是将所生成的数字存放在一个数组中，然后按照相反的次序打印它们，这是 3.6 节中函数 itoa 所采用的方法。另一种是采用递归，此方法中函数 printd 首先调用其自身来处理所有打头的数字，然后再打印结尾的数字。同样，这个版本在处理最大负数时也会出错。

```
#include <stdio.h>

/* printd: 将 n 按十进制数进行打印 */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

当函数递归调用自身时，每次调用都会得到一个新的全部自动变量的集合，该集合与之前调用获得的变量集合彼此独立。这样，在 printd(123) 中，第一个 printd 接到的参数 n 为 123，它将 12 传给第二个 printd，后者再将 1 传给第三个 printd。第三层的 printd 打印 1，然后返回第二层；该层的 printd 打印 2，然后返回第一层；第一层的 printd 打印 3 并终止执行。

递归的另一个较好的例子是快速排序（quicksort）。该算法是 C. A. R. Hoare 于 1962 年发明的。对于一个给定的数组，选取一个元素，并以该元素为界将其他元素分为两个子集，一个子集的所有元素都小于选定元素，另一个子集的所有元素都大于或等于该元素，然后对两个子集递归执行同一过程。当某个子集的元素少于 2 个时，该子集就不需要再进行排序，于是递归就终止了。

我们的快速排序版本不是最快的，但它是其中最简单的版本之一。我们选取每个（子）数组的中间元素进行划分。

```
/* qsort: 将 v[left] … v[right] 按递增次序排序 */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)  /* 假如数组元素少于 2 个 */
        return;         /* 则不做任何操作 */
    swap(v, left, (left + right)/2); /* 将划分元素移动 */
    last = left;                     /* 至 v[0] */
    for (i = left+1; i <= right; i++)    /* 对其余元素进行划分 */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);       /* 将划分元素移回适当位置 */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

由于交换操作在 qsort 中出现了 3 次，因此我们将其作为一个独立的函数。

```
/* swap: 将 v[i]与 v[j]互换 */
void swap(int v[], int i, int j)
{
```

```
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

标准库中包含了 qsort 的一个版本，它可以对任意类型的对象排序。

递归程序不能节省存储空间，因为它必须在某处维护一个存放待处理值的栈。它也不会执行得更快。但是相对于与之等价的非递归程序，递归代码更加紧凑，编写它和理解它常常要容易得多。递归对于处理像树之类递归定义的数据结构尤其方便；在 6.5 节我们会见到一个非常好的例子。

练习 4-12. 运用 printd 的思想编写一个递归版本的 itoa 函数;即通过递归调用将一个整数转换为字符串形式。

练习 4-13. 编写一个递归版本的 reverse(s)函数，该函数将字符串 s 本身颠倒过来。

4.11 C 预处理器

C 语言的预处理器提供了一定的语言功能，预处理从概念上讲是编译过程中单独执行的第一个步骤。其中两个最常用的特性是#include（用于在编译过程中包含某个文件的内容）和#define（以任意设定的字符序列替代一个标记）。在本节中描述的其他特性还包括条件编译及带参数的宏。

4.11.1 文件包含

文件包含功能使得处理大量#define 语句以及声明更加方便。任何形如：

```
#include “文件名”
```

或

```
#include <文件名>
```

的源代码行都会被替换为由文件名指定的文件的内容。如果是文件名由引号引上，那么通常从包含该文件的源程序所存放的位置开始查找该文件；如果没有查找到，或者文件名由尖括号括上，则按照 C 语言的具体实现所定义的规则查找该文件。被包含的文件本身也可以包含#include 代码行。

源文件的开始处常常有多个#include 行,用来包含常用的#define 语句和 extern 声明，或者用于访问如<stdio.h>等头文件内库函数的函数原型声明。（严格地说，这些要包含的内容并不一定以文件的形式存在，头文件的访问方式依赖于具体实现。）

#include 是大型程序中将声明紧密关联在一起的常用手段。它保证提供给所有源文件的是一致的定义和变量声明，这样可以消除掉那些与一致性有关的棘手错误。很自然地，当某个包含文件发生变化时，所有依赖于它的文件都必须重新编译。

4.11.2 宏替换

形如

```
#define  名字  替换文本
```

的宏定义给出了最简单的一种宏替换——其后出现的标记名字都将被替代为替换文本。#define 中的名字具有与变量名相同的形式；替换文本则是任意的。通常替换文本就是宏代码行的后面部分，但是一个长的宏定义可能连续存放为多个行(通过每行的末尾放置一个 \ 来表示续行)。#define 定义的名字的作用域是从其定义位置开始到该被编译文件的末尾为止。宏定义中可以使用之前的宏定义。宏只对标记进行替换，而在带引号的字符串中不会发生替换。例如，如果 YES 是一个宏定义名字，对于 printf("YES")或者 YESMAN 都不会进行宏替换。

任何宏定义的名字都可以被配上任意的替代文本。例如，

```
#define  forever  for (;;)      /* 无限循环 */
```

定义了一个新"关键字"forever，表示一个无限循环。

还可以定义带参数的宏，这样宏的不同调用能具有不同的替换文本。例如，定义一个称为 max 的宏：

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

尽管它看上去像一个函数调用，但 max 在使用时会扩展为代码行。每次出现的形式参数（在这里是 A 或 B）都会被替换为相应的实在参数。因此代码行

```
x = max(p+q, r+s);
```

将被替换为

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

由于对所有实在参数的处理都是一致的，因此这个宏能用于任何数据类型；对于不同的数据类型，max 宏只需要一种，但 max 函数则需要有多种。

如果仔细考虑上述对 max 的扩展，你会发现一些陷阱。其中表达式的求值进行了两次；这对于包含递增操作符或输入及输出等有附带效应的表达式是一件坏事。例如，

```
max(i++, j++)    /* 错了！ */
```

会将较大的值递增两次。还要适当小心地用括号来确保求值的既定顺序；请考虑下面的宏

```
#define  square(x)  x * x   /* 错了！  */
```

在调用 suqare(z+1) 时会出现什么样的情况。

但宏是有价值的，一个实用的例子来自于<stdio.h>，该头文件中的 getchar 和 putchar 常被定义为宏，目的是避免因每处理一个字符都要进行一次函数调用而带来的运行时开销。<ctype.h>中的函数常常也是通过宏实现的。

可以使用 #undef 取消名字的宏定义，这通常用于保证某个例程是一个真正的函数而不是宏：

```
#undef getchar

int getchar(void) { … }
```

带引号的字符串中的形式参数不会被替代。然而，如果替换文本中的某个参数名之前带有一个 #，此组合在该形参被替换为实参时将被扩展为带引号的字符串。这一方法可以与字符串连接配合使用，例如构建一个打印调试信息的宏：

```
#define  dprint(expr)  printf(#expr " = %g\n", expr)
```

当该宏被调用时，譬如

```
dprint(x/y);
```

它被扩展为

```
printf("x/y" " = %g\n", x/y);
```

而这两个字符串是相连接的，因此效果等同于

```
printf("x/y = %g\n", x/y);
```

实在参数中的 " 将被替换为 \"，\ 将被替换为 \\，因此扩展后得到的是合法的字符串常量。

预处理器的操作符 ## 提供了一种将实在参数拼接起来的方法。如果一个形参在替换文本中与 ## 相邻，那么该形参会被替换为对应的实参，而该 ## 及其周边的空白符都会被移除，所得结果将被重新扫描。例如，下面的宏将它的两个参数拼接起来：

```
#define paste(front, back)  front ## back
```

这样 paste(name, 1) 就创建出标记 name1。

嵌套使用 ## 的规则比较晦涩；进一步的细节请参见附录 A。

练习 4-14. 定义一个宏 swap(t, x, y)，它将类型为 t 的两个参数 x、y 彼此互换。（使用程序块结构会有所帮助。）

4.11.3 条件包含

预处理本身可以通过（在预处理过程中求值的）条件语句来进行控制。这提供了一种可根据编译过程中的条件取值来选择性地包含代码的方法。

#if 语句对一个整型常量表达式求值（表达式中不得包含 sizeof、强制类型转换以及枚举常量），如果表达式的值不为零，那么从该行后一直到 #endif 、#elif 或者 #else 之间的行都会被包含到代码中。（预处理语句#elif 的作用类似于 else if。）在#if 语句中，可以使用表达式 defined(*名字*)，如果*名字*已被定义，则该表达式取值为 1，否则为 0。

例如，为了确保文件 hdr.h 的内容只会被包含一次，可以像下面这样将文件内容用条件语句包裹起来：

```
#if !defined(HDR)
#define HDR

/* hdr.h 的内容放在这里 */

#endif
```

hdr.h 文件在第一次被包含时会定义名字 HDR；而之后的包含会因发现该名字已定义而直接跳转到#endif 处。类似做法能用于避免对文件的多次包含。如果一致地使用这种做法，那么每个头文件自身都可以包含它所依赖的任何头文件，而头文件的使用者不必处理其间的相互依赖关系。

下面的预处理序列通过检测名字 SYSTEM 来决定包含哪个版本的头文件：

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
```

```
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

#ifdef 与#ifndef 语句专用于测试某个名字是否已被定义。前面有关#if 的第一个例子可以被编写为：

```
#ifndef HDR
#define HDR

/* hdr.h 的内容放在这里 */

#endif
```

第 5 章 指针与数组

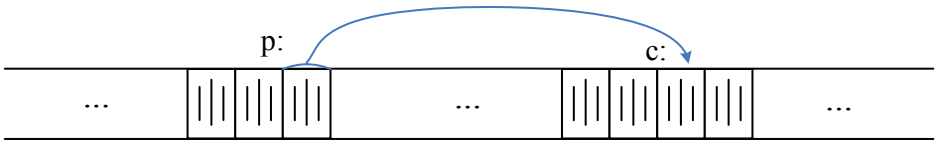
指针是存放某个变量的地址的变量。C 语言中大量采用了指针，一部分原因是在某些情况下指针是表示计算的唯一方法，另一部分原因是使用指针编写代码通常比用其他方法更加简洁高效。指针与数组存在着紧密的联系；本章还将探究它们的这种联系并说明如何利用这一特性。

与 goto 语句一样，指针也被认为是构建无法理解的程序的一种诡异途径。这的确是非常可能的，如果它们没有被小心地使用，极易出现未指向所期望地址的指针。然而，在一定规则下使用指针却能够让代码变得简单明了，这也是我们试图展现的方面。

ANSI C 的一个主要变化是明确了指针的操控规则，将优秀程序员已经运用的以及优秀编译器已经增强的特性正式确立下来。此外，void * 类型（空指针）替代 char * 成为合法的通用指针类型。

5.1 指针和地址

我们以一个描述内存组织的简化图例开始讨论。典型的机器拥有一列连续编号或编址的内存单元，这些单元可单独操控，也可按连续的单元组操控。普遍的情况是单个字节是 char，双字节单元可当作一个 short 型整数，而连续四个字节则构成一个 long 型整数。指针是能够存放一个地址的存储单元组（通常为双字节或 4 字节）。这样，假如 c 是一个 char，p 是一个指向 c 的指针，则这种情况可表示如下：



一元运算符 & 用于取一个对象的地址，因此语句

```
p = &c;
```

将 c 的地址赋给变量 p，并称 p“指向”c。& 运算符只能应用于内存中的对象——变量以及数组元素。它不能被用于表达式、常量或者寄存器变量。

一元运算符 * 是间接（indirection）或解引用（dereferencing）运算符；当其应用于指针时即访问此指针所指向的对象。假设 x 与 y 是整数，ip 是指向整数的指针。下面的代码说明了如何声明一个指针以及 & 和 * 的用法：

```
int x = 1, y = 2, z[10];
int *ip;        /* ip 是指向 int 型的指针 */

ip = &x;        /* ip 指向了 x */
y = *ip;        /* y 的值变为 1 */
*ip = 0;        /* x 的值变为 0 */
ip = &z[0];     /* ip 指向了 z[0] */
```

x、y 与 z 的声明我们从一开始就已经见过了。指针 ip 的声明方式

```
int *ip;
```

的目的是便于记忆；它表明表达式 *ip 是一个 int 型数。这种声明变量的语法模仿了含有此

变量的表达式的语法。同样的理由也用于函数的声明之中。例如：

```
double *dp, atof(char *);
```

表明表达式 *dp 与 atof(s)的值都为 double 类型，且 atof 的参数是指向 char 的指针。

读者可能会发现一个隐含的规则，即指针必须指向一个特定类型的对象：每个指针都指向某个具体的数据类型。(存在一个例外：一个"指向 void 的指针"用于保存任何类型的指针，但其无法解引用自身。我们将在 5.11 节介绍这个概念。)

假如 ip 指向整数 x，那么在任何出现 x 的上下文中，x 的位置都能够用*ip 替换，因此

```
*ip = *ip + 10;
```

将 *ip 的值增加了 10。

一元运算符 * 和 & 的优先级较算术运算符高，因此赋值语句

```
y = *ip + 1
```

首先取出 ip 指向的对象，增加 1，并将结果赋给 y；而

```
*ip += 1
```

将 ip 指向的对象增 1，其等同于

```
++*ip
```

及

```
(*ip)++
```

最后这个例子中括号是必须的；否则表达式会将 ip 本身（而不是 ip 所指向的对象）增 1，这是由于 * 和 ++ 等一元运算符是由右至左接合的缘故。

最后，由于指针也是一种变量，它们也可不经过解引用而直接使用。例如 iq 是另一个指向 int 的指针，

```
iq = ip
```

将 ip 的内容复制到 iq 中，从而使 iq 指向 ip 所指的对象。

5.2 指针和函数参数

由于 C 中的函数采用按值传递参数，因此没有直接的途径可以让被调函数改变调用者函数的变量。例如，排序程序可能使用一个 swap 函数互换两个乱序的元素。使用语句

```
swap(a, b);
```

无法达到目的。这里 swap 函数的定义是

```
void swap(int x, int y) /* 错了！ */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

由于是按值调用，swap 不能改变调用它的程序中的参数 a 与 b。上面的函数仅仅交换了 a 和 b 的副本。

获得所期望效果的方式是让调用者传递需修改变量的指针：

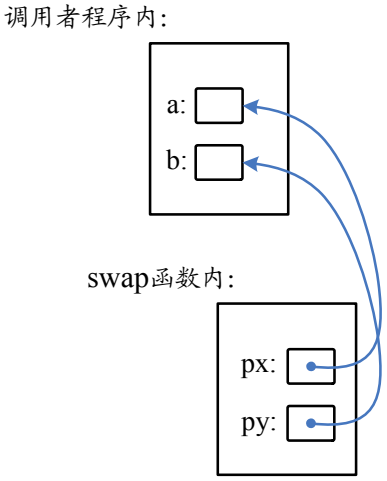
```
swap(&a, &b);
```

运算符 & 得到变量的地址，因此 &a 即为 a 的指针。这里 swap 函数的参数被声明为指针，并通过它们间接访问真正需要操作的变量。

```
void swap(int *px, int *py)    /* 互换 *px 与 *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

用图表示即：



指针参数使得一个函数能够访问及修改调用它的函数中的对象。例如，考虑函数 getint，它转换输入的任意格式的字符流，将其分解为整数值，每次调用得到一个整数。getint 需要返回它找到的数值，并当所有输入处理完毕时提示文件结束（EOF）。这两种值只能通过不同的路径传输，因为无论使用什么值代表 EOF，都可能与输入的整数值相同。

一个解决方案是让 getint 用函数返回值传递文件结束状态，并用一个指针参数存放转换的整数返回给调用者。此方案也被用于 scanf；参见 7.4 节。

下面的循环把调用 getint 所得到的整数填到一个数组中：

```
int n, array[SIZE], getint(int *);

for (n = 0; n < size && getint(&array[n]) != EOF; n++)
    ;
```

每次调用将 array[n]设置为本次在输入中找到的整数，然后将 n 递增。注意其中的关键是将 array[n]的地址传送给 getint。否则无法让 getint 将转换后的整数传回给调用者。

本书的 getint 版本返回 EOF 表示文件结束，返回零表示本次输入不是数字，返回正值表示输入包含了一个有效的数字。

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: 取得输入中整数并放到 *pn 中 */
int getint(int *pn)
```

```
{
    int c, sign;

    while (isspace(c = getch()))  /* 略过空白符 */
        ;
    if (isdigit(c) && c != EOF && c != '*' && c != '-') {
        ungetch(c);  /* 它不是数字 */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

在整个 getint 中，*pn 都作为一个普通的 int 变量使用。我们还使用了 getch 和 ungetch（在 4.3 节中描述），这样一个必须额外读出的字符能被压回输入中。

练习 5-1. 本节所给代码中 getint 将其后未带数字的 + 或 - 按有效的表达式 0 处理。将其修正，并将这一字符压回输入中。

练习 5-2. 编写 getint 的浮点数变体 getfloat。getfloat 的函数返回值应为什么类型？

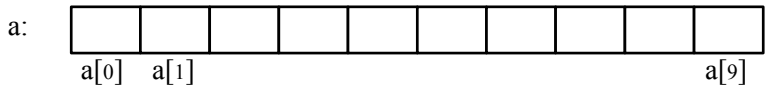
5.3 指针和数组

C 语言中指针和数组之间有很强的联系，强到应当将它们一起讨论。任何用数组下标来完成的操作都同样可以用指针来完成。通常使用指针速度会更快，但在某种程度上也更难于理解——至少对于新手来说是这样。

声明

```
int a[10];
```

定义了一个大小为 10 的数组 a，也就是包含了 10 个连续对象的“整块儿”，对象分别称为 a[0]、a[1]、…、a[9]。



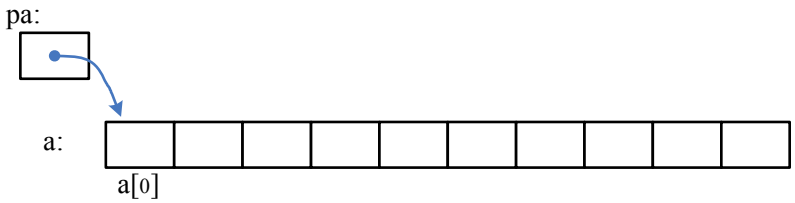
符号 a[i]代表了数组中的第 i 个元素。如果 pa 是一个整数指针，声明为

```
int *pa;
```

那么赋值语句

```
pa = &a[0];
```

将 pa 设为指向 a 的第 0 个元素；即 pa 包含了 a[0]的地址。



那现在赋值语句

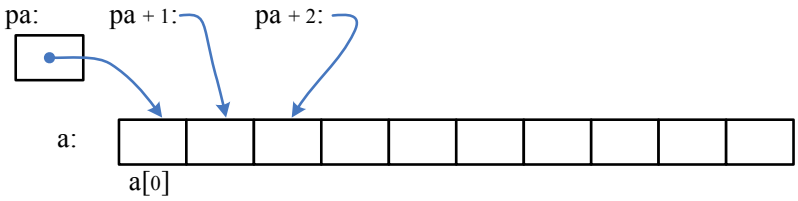
```
x = *pa;
```

会将 a[0]的内容复制到 x。

如果 pa 指向了一个数组中的某一具体元素，那么根据定义，“pa + 1”将指向下一个元素，“pa + i”将指向 pa 之后的第 i 个元素，而“pa - i”指向 pa 之前的第 i 个元素。因此，如果 pa 指向 a[0]，那么

```
*(pa + 1)
```

代表了 a[1]的内容，pa + i 是 a[i]的地址，*(pa + i)是 a[i]的内容。



无论数组 a 中的变量的类型和大小如何，上述规则都是成立的。“将指针加 1”以及指针的所有其他算术运算的意义就是：pa + 1 指向 pa 的下一个对象，pa + i 指向 pa 的向后的第 i 个对象。

数组下标与指针运算之间的关联相当紧密。根据定义，数组类型的变量或表达式的值是此数组的第 0 个元素的地址。因此，如下赋值

```
pa = &a[0];
```

之后 pa 与 a 拥有同样的值。由于数组名称等同于其起始元素的位置，赋值 pa=&a[0]也可被写为

```
pa = a;
```

a[i]也可以被写成*(a+i)，这至少在初次见到时会令人十分惊讶。C 语言对 a[i] 求值时会立即将它转换为*(a+1)；这两种形式是等价的。将运算符 & 应用到等式两边，则得到&a[i]与 a+i 也是等价的：a+i 是 a 之后第 i 个元素的地址。反过来，如果 pa 是一个指针，在表达式中它也可以与下标一起使用；pa[i]等同于*(pa+i)。简言之，“数组+下标”表达式与“指针+偏移”表达式是等价的。

数组名与指针间有一个区别必须牢记：指针是一个变量，所以 pa=a 和 pa++都是合法的；但数组名却不是变量，因而 a=pa 和 a++等写法都是不合法的。

当一个数组名被传递给函数时，实际传递的是起始元素的位置，在该函数中，对应参数是一个局部变量，因此一个数组名参数是一个指针，即一个存放地址的变量。我们能够利用这一事实编写另一个版本的 strlen，其用于计算一个字符串的长度。

```
/* strlen: 返回字符串 s 的长度 */
int strlen(char *s)
{
```

```
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

由于 s 是一个指针，对其递增是完全合法的；s++不会对 strlen 函数的调用者中的字符串产生影响，只会增加 strlen 自己的指针拷贝。这就意味着像

```
strlen("hello, world"); /* 字符串常量 */
strlen(array);          /* 字符数组 char array[100] */
strlen(ptr);            /* 字符指针 char *ptr */
```

等调用都能工作。

作为函数定义中的形参，

```
char s[];
```

与

```
char *s;
```

是等同的；我们更倾向于使用后者，因为其更明显地标明此参数是一个指针。当数组名被传递给函数时，函数可根据便利程度来认为传递的是数组还是指针，并照此使用。甚至可以两种形式都使用，假如这样看起来清晰恰当。

通过传递数组内元素的起始位置，可以仅传递部分数组给函数。例如，a 是一个数组，

```
f(&a[2])
```

及

```
f(a+2)
```

都向函数 f 传递以 a[2]作为起始地址的部分数组。而 f 的参数声明可以写作

```
f(int arr[]) { … }
```

或

```
f(int *arr) { … }
```

因此对于 f 而言，参数所对应的实际只是数组的一部分没有关系。

如果确定元素存在，也可以向前索引一个数组；像 p[-1]、p[-2]等等在语法上都是合法的，它们对应于 p[0]之前紧挨着的元素。当然，引用数组边界外的对象则是非法的。

5.4 地址运算

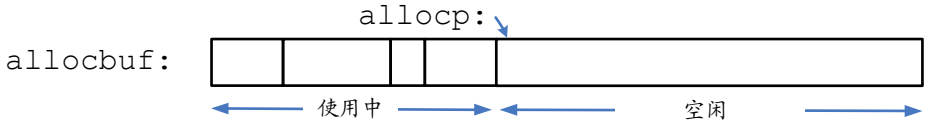
如果 p 是指向数组某个元素的指针，则 p++ 使 p 指向下一个元素；p+=i 使 p 指向当前元素之后的第 i 个元素。它们及其他类似的语句是形式最简单的指针/地址运算。

C 语言的地址运算方法是一致且有规律的；指针、数组、地址运算整合在一起是 C 语言的一大优点。下面我们通过编写一个初级的存储分配器来进行说明。第一个是 alloc(n)，它返回指针 p，p 指向连续的 n 个字符空间。该空间可被 alloc 的调用者用于存放字符。第二个是 afree(p)，它释放用 alloc 申请的存储，使之能够被重新（申请）使用。这些程序之所以"初级"是由于 afree 的调用顺序必须与 alloc 的顺序相反。也就是说，由 alloc 和 afree 管理的存储是一个栈——一个后进先出的队列。标准库提供的类似函数 malloc 和 free 则没有这个限制；在 8.7 节我们将介绍如何实现它们。

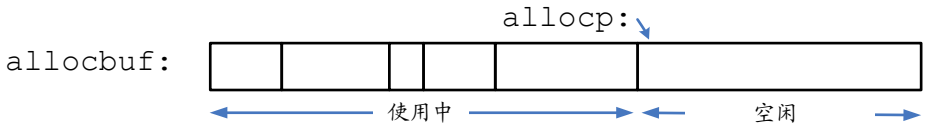
最容易的实现方法是让 alloc 管理一个较大的字符数组，我们将其称为 allocbuf。此数组由 alloc 和 afree 私有使用。由于这两个函数的参数用的是指针，而不是数组下标，其他程序都无需知道该数组的名字，因此可以在包含 alloc 和 afree 的那个源文件中用 static 声明该数组，这样它就不会被外部程序见到。实际实现中，这个数组甚至可以没有名字，它可能由（通过 malloc 调用或者向操作系统请求而获得的）匿名存储块的指针所替代。

另一所需的信息是 allocbuf 已被使用的量。我们用一个称为 allocp 的指针指向头一个空闲元素。当 alloc 接到 n 个字符的申请时，它检查 allocbuf 是否剩余有足够的空间。如果有，alloc 返回 allocp 的当前值（譬如，空闲块的起始位置），并将其增加 n，指向后面的空闲区域。如果没有足够空间，则 alloc 返回零值。假如 p 在 allocbuf 的范围之内，afree(p) 的工作只是将 allocp 设置为 p。

调用alloc之前:



调用alloc之后:




```
#define ALLOCSIZE 10000  /* 可用空间的大小 */

static char allocbuf[ALLOCSIZE];      /* 用于分配的存储 */
static char *allocp = allocbuf;       /* 头一个空闲的位置 */

char *alloc(int n)  /* 返回指向 n 个字符的指针 */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) {  /* 满足 */
        allocp += n;
        return allocp - n;  /* 分配前 p 所指的位置 */
    } else     /* 没有足够空间 */
        return 0;
}

void afree(char *p)  /* 释放 p 指向的存储 */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

通常指针可以像任何其他变量一样初始化，虽然有意义的值只有零值或者之前定义的某个恰当类型的数据的地址。声明

```
static char *allocp = allocbuf;
```

定义 allocp 为字符型指针，并初始化为指向 allocbuf 的开头，这也是程序一开始时的头一个空闲位置。由于数组名也是第零个元素的地址，这个声明也可以被写为

```
static char *allocp = &allocbuf[0];
```

条件测试

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* 满足 */
```

检查是否有足够的空间满足 n 个字符的请求。如果足够，allocp 的新值最多为 allocbuf 的末尾元素多 1。如果请求可以被满足，alloc 返回指向一整段字符的起始位置的指针（注意这个函数声明本身）。如果不满足，alloc 的返回必须表示已经没有剩余空间了。C 语言保证零值决不会是一个有效的数据地址，因此返回一个零值可以作为异常事件的信号，在本例中即表示空间不足。

指针和整数不能互换。零值是唯一的例外：零值常量可以被赋给一个指针，而指针可以与零值常量相比较。零值常常用符号常量NULL所替换，更明晰的标示出这是用于指针的特殊赋值。NULL定义在 <stddef.h> 中[①]。从现在起我们将使用NULL来表示指针零值。

条件测试如

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* 满足 */
```

和

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

展示了指针运算的几个重要方面。首先，指针可以在某些情况下进行比较。如果 p 和 q 分别指向属于同一数组的元素，则关系运算符如==、!=、<、>= 等等都能正确地工作。例如，若 p 指向的元素在 q 指向的元素之前，则

```
p < q
```

为真。任何指针与零值进行相等或不相等的比较都是有意义的。但是没有指向同一个数组的指针相互进行比较或者运算的行为则是未定义的。（存在一个例外：超过数组末尾的第一个元素可以用于指针运算。）

其次，我们已经看到指针和整数可以进行加减，表达式

```
p + n
```

表示 p 当前所指向对象之后的第 n 个对象的地址；无论 p 所指对象是什么类型都是这样。n 会根据 p 所指对象的大小（这取决于 p 的声明）进行扩展。例如，若 int 是 4 个字节，那么 int 型指针运算中 n 将扩大 4 倍。

指针减法也是有效的：如果 p 和 q 指向属于同一数组的元素并且 p < q，那么 q - p + 1 就是就是从 p 至 q 的元素的数目。这一特性可以被用于编写另一个版本的 strlen 函数：

```
/* strlen: 返回字符串 s 的长度 */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

在函数的声明部分，p 被初始化为 s，即指向字符串的首字符。在 while 循环中，字符被依次检查，直到碰见末尾的'\0'。由于 p 是指向字符的指针，每次 p++ 使 p 前进指向下一个字符，p - s 给出了 p 前进的字符数——这正是字符串的长度。（字符串中的字符数可能太大而不能用一个 int 来存放。头文件 <stddef.h> 中定义了一个 ptrdiff_t 类型，足够容纳两个指针的（正负）差值。然而，假如我们非常谨慎，我们应该使用 size_t 作为 strlen 的返回类型，使此程序与标准库的版本一致。size_t 是由 sizeof 运算符返回的无符号整数类型。）

指针运算是一致的：如果我们处理的是比字符占用更多的存储空间的浮点数，并且 p 是一

[①] 根据标准，NULL 在<stddef.h>中定义，尽管它也出现在如<stdio.h>、<stdlib.h>等头文件中。

个浮点指针，那么 p++将指向下一个浮点数。这样我们只需要将 alloc 和 afree 中所有的 char 都替换为 float，就能编写出另一个用于处理浮点数而不是字符的 alloc 版本。所有的指针操作都自动将所指对象的大小考虑在内。

有效的指针运算包括将指针（值）赋给同一类型的指针，指针和整数进行加减，指向属于同一数组的元素的指针相比较或相减，以及指针赋零值或与零值比较。所有其他的指针运算都是非法的。例如两个指针相加、相乘或相除，指针与浮点数或双精度数相加，甚至将某个类型的指针（void * 除外）在没有强制转换的情况下赋给另一个类型的指针等等都是不合法的。

5.5 字符指针与函数

字符串常量，如

```
"I am a string"
```

是一个字符数组。在内部表示中，此数组用空字符'\0'结束，这样程序才能找到它的结尾。数组的存储长度也因此比双引号间的字符数目大 1。

字符串常量最常见的或许是用作函数的参数，就像

```
printf("hello, world\n");
```

当类似这样的一个字符串出现在程序中时，对它的访问通过一个字符指针进行；printf 接收到的是指向该字符数组起始位置的指针。就是说，字符串常量是通过其第一个元素的指针来访问的。

字符串常量不必一定用作函数参数。如果 pmessage 声明为

```
char *pmessage;
```

那么语句

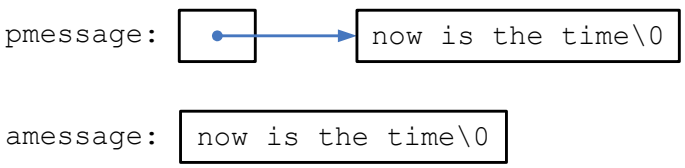
```
pmessage = "now is the time";
```

将这个字符串数组的指针赋给 pmessage。这**并非**对字符串的拷贝，复制的只是指针。C 语言没有提供任何将字符串作为一个单元处理的操作。

下面的定义存在一个重要的区别：

```
char amessage[] = "now is the time"; /* 数组 */
char *pmessage = "now is the time";  /* 指针 */
```

amessage 是一个数组，大小刚好足够容纳用于初始化它的对应字符串和'\0'。此数组包含的字符可能变化，但 amessage 始终对应着同一个存储空间。而后者 pmessage 是一个指针，其被初始化为指向一个字符串常量；这个指针之后可能被修改而指向不同的地方，但如果你试图修改后者初始化字符串的内容，则结果是未定义的。



我们将通过对标准库的两个有用函数的几个修改版本的考察，来详细说明指针和数组的多个方面。第一个函数是 strcpy(s,t)，其将字符串 t 复制到字符串 s 中。用 s = t 来表达当然很好，但这样复制的不是字符串而是指针。为了复制字符串，我们需要一个循环。第一个是数组版本：

```
/* strcpy: 将 t 复制到 s; 数组下标版本 */
```

```
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

对应地，下面是 strcpy 的指针版本：

```
/* strcpy: 将 t 复制到 s; 指针版本 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

由于参数是按值传递的，strcpy 可按其想要的任何方式使用参数 s 和 t。这里将它们看作初始化好的指针比较方便，它们沿数组一次前进一个字符，直到 t 的结束符'\0'被复制到 s 为止。

实际中，strcpy 不会像上述程序那样编写。有经验的 C 编程者会更喜欢

```
/* strcpy: 将 t 复制到 s; 指针版本 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

这个程序在循环的条件检测部分递增 s 和 t。*t++的值为 t 增加之前所指向的字符；后缀 ++ 在字符被获取之后才改变 t。同样地，该字符被存放到原 s 的位置后 s 才被递增。该字符还作为与'\0'相比较的值来控制循环。其最终效果就是字符从 t 复制到 s，直到包含结尾的'\0'为止。

作为最后一步精简，请注意与'\0'的比较其实是多余的，因为该判定只是"这个表达式是否为零"。因此该函数可能会按以下方式编写：

```
/* strcpy: 将 t 复制到 s; 指针版本 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

这种表示法尽管初看起来可能显得难于理解，但其方便性是很明显的。应该掌握这种惯用法，因为你经常会在 C 程序中见到它。

标准库（<string.h>）中的 strcpy 将目的字符串作为函数值返回。

我们考察的第二个程序是 strcmp(s, t)，其按照字母序比较字符串 s 和 t，并在 s 小于、等于或大于 t 时返回负数、零或者正数。返回值是 s 和 t 第一个不同的字符相减的差值。

```
/* strcmp: 如果 s<t 返回<0，如果 s==t 返回 0，如果 s>t 返回>0 */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
```

```
    return s[i] - t[i];
}
```

对应 strcmp 的指针版本：

```
/* strcmp: 如果 s<t 返回<0，如果 s==t 返回 0，如果 s>t 返回>0 */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

由于 ++ 和 -- 既是前缀运算符又是后缀运算符，因此也会出现 * 和 ++ 及 -- 的其他组合，尽管要少见一些。例如：

```
*--p
```

首先将 p 减 1，然后再获取 p 所指向的字符。实际上，下面这对表达式

```
*p++ = val;   /* 将 val 压入栈 */
val = *--p;   /* 将栈顶元素弹出到 val 中 */
```

是栈压入和弹出的标准惯用法；见 4.3 节。

头文件<string.h>包含有本节所述的函数的声明，以及标准库中各种类型的其他字符串处理函数的声明。

练习 5-3. 编写 strcat 函数的指针版本。此函数我们在第 2 章描述过：strcat(s, t)将字符串 t 复制到 s 的末尾处。

练习 5-4. 编写函数 strend(s, t)，如果字符串 t 出现在字符串 s 的末尾处，函数返回 1，否则返回 0。

练习 5-5. 编写库函数 strncpy、strncat 和 strncmp，这些函数只处理它们字符串参数的最多前 n 个字符。例如，strncpy(s, t, n)将 t 的最多前 n 个字符复制到 s 中。它们的全面描述参见附录 B。

练习 5-6. 以指针来替代数组下标，重写之前章节和练习中的相应程序。可能的好选择包括 getline（第 1、4 章），atoi、itoa 及其变体（第 2、3、4 章），以及 strindex 和 gettop（第 4 章）。

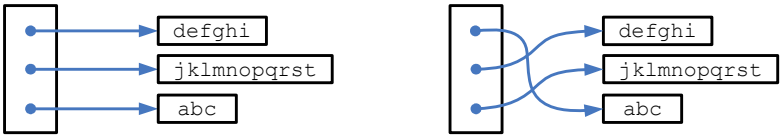
5.6 指针数组；指向指针的指针

由于指针本身也是变量，它们可以像其他变量那样存放在数组中。让我们编写一个将一组文本行按字母序排序的程序来说明这一点。这是 UNIX 程序 sort 去掉其他功能的版本。

在第 3 章我们给出了一个 Shell 函数 sort 用于整数数组排序，并在第 4 章用快速排序对其进行了改进。在这里相同的算法一样有效，只是现在我们必须处理文本行，它们的长度各异，且与整数不同，不能用单个操作来比较或移动。我们需要一个数据表示法来高效便利地处理变长的文本行。

我们引入指针数组来解决此问题。如果待排序的文本行首尾相连存放在一个大的字符数组中，则每一行可通过指向它的首字符的指针来访问。这些指针可以存放在一个数组中。两个文本行可通过将其指针传递到 strcmp 来进行比较。当两个乱序的行需要交换时，交换的是指针

数组中对应的指针，而不是交换文本行本身。



这样就消除了移动文本行本身而带来的复杂存储管理和高开销这一对共生问题。

排序过程共分为三个步骤：

```
读入输入的所有文本行
将它们排序
将它们顺序打印出来
```

通常情况下，最好是将程序按照这种自然划分而分为多个函数，并通过一个主程序来控制其他函数。让我们将排序步骤稍稍推后，先集中讨论数据结构和输入输出部分。

输入函数需要收集和保存每一行字符，并建立一个指向这些行的指针数组。它还需累计输入行数，因为排序和打印需要这一信息。由于输入函数只能处理有限数量的输入行，它可返回某个非法行数（如-1）来表示出现了过多输入的情况。

输出函数仅需按照指针数组中的顺序打印文本行。

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

#define MAXLINES 5000       /* 能被排序的最大行数 */

char *lineptr[MAXLINES];    /* 指向文本行的指针 */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* 将输入行排序 */
main()
{
    int nlines;       /* 读入的输入行数 */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}

#define MAXLEN 1000     /* 任一输入行的最大长度 */
int getline(char *, int);
char *alloc(int);

/* readlines: 读取输入行 */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
```

```
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* 删除换行符 */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines: 写输出行 */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

函数 getline 来自 1.9 节。

一个主要的新事物是 lineptr 的声明：

```
char *lineptr[MAXLINES]
```

说明 lineptr 是一个包含 MAXLINES 个元素的数组，每个元素都是指向字符的指针。也就是说，lineptr[i]是一个字符型指针，*lineptr[i]是其指向的字符，即所保存的第 i 个文本行的首字符。

由于 lineptr 是数组的名字，它可以按照指针来对待，就像我们在之前的例子中所做的那样，因此 writelines 可以改写为：

```
/* writelines: 写输出行 */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

初始的*lineptr 指向第一行；每增 1 就使它前进到下一行的指针，同时 nlines 计数减 1。

输入和输出搞定之后，我们进一步考虑排序部分。第 4 章的快速排序（quick-sort）需要微小的改变：声明需要修改，比较操作必须通过调用 strcmp 来完成。算法则保持不变，这让我们对程序仍能工作有了一定的把握。

```
/* qsort: 将 v[left]…v[right]排为递增次序 */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right)  /* 如果数组元素小于 2 个 */
        return;         /* 则什么也不做 */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
```

```
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

同样地，swap 程序只需要一小点改变：

```
/* swap: 将 v[i]和 v[j]互换 */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

由于 v（指代 lineptr）的任何元素都是字符指针，因此 temp 必须也是，这样才能够相互复制。

练习 5-7. 重写 readlines，将文本行存放在由 main 提供的数组中，而不是调用 alloc 来获得存储。这个程序会比改写之前快多少？

5.7 多维数组

C 语言提供了矩形多维数组，尽管实际中它们用得远比指针数组要少。在本节中我们将展示它们的一些特性。

考虑日期转换的问题，将某月某日转换为某年的第几天，以及反向的转换。例如 3 月 1 日是非闰年的第 60 天，是闰年的第 61 天。我们定义两个函数进行转换：day_of_year 将某月某日转换为某年的第几天，而 month_day 将某年的第几天转换为某月某日。由于后一个函数要计算两个值，其参数月和日应为指针：

```
month_day(1988, 60, &m, &d)
```

将 m 置为 2，d 置为 29（2 月 29 日）。

这两个函数都需要同样的信息，一份每月天数的表（"九月里有三十天……"）。由于闰年和非闰年有不同的每月天数，将它们分作二维数组的两行比在计算过程中考虑 2 月份的变化要容易一些。用于执行转换的数组和函数如下：

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: 从月/日转换为一年的第几天 */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i]
```

```
    return day;
}

/* month_day: 将一年的第几天转换为月/日 */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

回忆逻辑表达式的运算，例如上面闰年的计算，其结果不是 0（假）就是 1（真），因而它可被用作数组 daytab 的下标。

数组 daytab 必须在 day_of_year 和 month_day 的外部声明，这样两个函数都能使用它。我们将 daytab 设为 char 类型，是为了说明 char 类型用于存放非字符的小整数也是合法的。

daytab 是我们碰到的第一个二维数组。在 C 中，二维数组实际上是一个一维数组，其每个元素都是一个数组。因而下标被写为：

```
daytab[i][j]  /* [行][列] */
```

而不是

```
daytab[i, j]  /* 错 */
```

除了这一标记上的区别，二维数组的处理方式与其他语言基本一致。元素按行存放，所以当元素按照存储顺序访问时，最右边的下标（即列）变化得最快。

数组以一列用花括号括起来的初始值进行初始化；二维数组的各行用对应的子列初始化。我们让数组 daytab 用一列 0 开头，这样月份就可以使用自然的 1 到 12 而不是 0 到 11。由于这里空间不是首要考虑的问题，这样做比调整下标更加直观。

如果要将一个二维数组传递给函数，函数的参数声明中必须包括数组的列数；而行数则无关，因为就像之前那样，传递的是数组行的指针，这里每一行是包含 13 个 int 的一维数组。在这个具体例子中，它是指向包含 13 个 int 的数组对象的指针。因此，如果要将数组 daytab 传递给函数 f，f 的声明将会是：

```
f(int daytab[2][13]) { … }
```

由于行数无关紧要，也可以是

```
f(int daytab[][13]) { … }
```

或者是

```
f(int (*daytab)[13]) { … }
```

此声明表示其参数是指向包含 13 个整数的数组的指针。这里圆括号是必须的，因为括号 [] 比 * 的优先级高。没有圆括号，声明

```
int *daytab[13]
```

就是一个包含 13 个整数指针的数组。更一般地，只有数组的第一个维度（下标）是可以不指定的，而其他维度都必须指定。

5.12 节有对复杂声明的进一步讨论。

练习 5-8．在 day_of_year 和 month_day 中没有错误检测，请修正这一缺陷。

5.8 指针数组的初始化

考虑函数 month_name(n)的编写，该函数返回包含第 n 月的名字的字符串。这里应用内部静态数组十分理想。month_name 包含一个私有的字符串数组，并返回指向正确名字串的指针。本节说明这个名字数组如何进行初始化。

其语法与之前的初始化类似：

```
/* month_name: 返回第 n 个月的名字 */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "Apirl", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n>12) ? name[0] : name[n];
}
```

声明 name 是一个字符指针数组，与排序用例中的 lineptr 是同一类型。初始化值是一列字符串，每个串被分配到数组中的相应位置。第 i 个串中的字符被放在某个地方，而指向它们的指针存放在 name[i]中。由于数组 name 的大小没有指定，编译器累计初始化值的个数并填入确切的数值。

5.9 指针与多维数组的比较

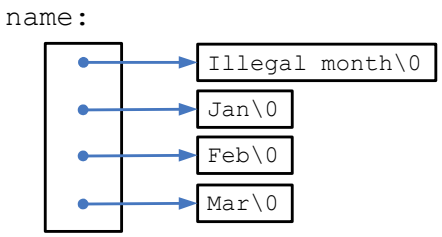
C 语言初学者有时会混淆二维数组与指针数组的区别，例如前一个例子中的 name。假设有定义

```
int a[10][20];
int *b[10];
```

那么语法上 a[3][4]和 b[3][4]都合法地引用了一个 int。但 a 是一个真正的二维数组：200 个 int 大小的空间已被预留了出来，传统的矩形下标计算公式 20×行＋列 用于得到元素 a[行][列]。而对于 b，该定义仅仅分配了 10 个指针，且没有初始化它们；初始化必须显式地完成，或者通过静态分配或者通过代码初始化。假设 b 每个元素指向一个 20 个元素的数组，那么将被预留的有 200 个 int，以及指针的 10 个单元。指针数组的一个重要优点是数组的每行可以有不同的长度。这样，b 的每个元素不必都指向具有 20 个元素的向量；可以有些指向 2 个元素，有些 50 个元素，有些则根本没有元素。

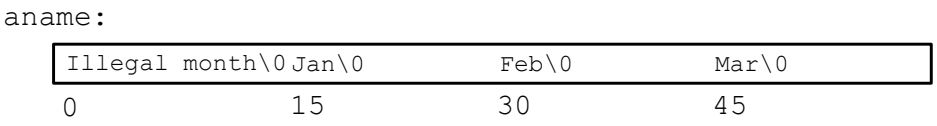
尽管我们只以整数进行了讨论，但指针数组最常用于存放长度各异的字符串，就像在函数 month_name 中的那样。指针数组的声明和图示如下：

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```



与二维数组相较：

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



练习 5-9. 重写函数 day_of_year 和 month_day，使用指针替代下标。

5.10 命令行参数

在支持 C 语言的环境中，存在一种在程序开始执行时将命令行参数传递给它的方法。当 main 被调用时，它有两个参数。第一个（传统上称为 argc，参数计数）是运行程序时命令行参数的个数；第二个（argv，参数向量）是一个指针，指向包含参数字符串的数组，其中每个串对应一个参数。我们惯常使用多级指针来操作这些字符串。

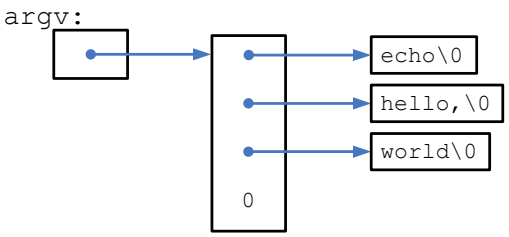
最简单的示例是程序 echo，它将其命令行参数用空格分隔并回显在一行上。即，命令

```
echo  hello, world
```

打印输出为

```
hello, world
```

按照常规，argv[0]是被调用程序的名字，所以 argc 至少为 1。如果 argc 是 1，那么在程序名之后没有命令行参数。在上面的例子中，argc 为 3，argv[0]、argv[1]和 argv[2]分别是"echo"、"hello,"和"world"。 可选参数的第一个是 argv[1]，最后一个是 argv[argc - 1]；此外，标准要求 argv[argc]是一个空指针。



echo 的第一个版本将 argv 看作一个字符指针数组：

```
#include <stdio.h>

/* 回显命令行参数; 版本 1 */
main(int argc, char *argv[])
{
    int i;
```

```
    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

由于 argv 是指向一个指针数组的指针，我们可以操作指针而不用数组下标。下面这个变体基于递增 argv，它是指向字符指针的指针，同时 argc 计数递减：

```
#include <stdio.h>

/* 回显命令行参数; 版本 2 */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

由于 argv 是指向参数字符串数组起始位置的指针，将它加 1（++argv）使它指向 argv[1] 而不是 argv[0]。每次递增将它顺移至下一个参数；*argv 则为指向对应参数的指针。与此同时，argc 被递减；当它变为 0 时，就完成了所有参数的打印。

我们可将 printf 的声明写为另一形式：

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

此语句说明 printf 的格式化参数也可以是表达式。

作为第二个例子，我们将 4.1 节的模式查找程序进行一些加强。如果读者还记得，我们把搜索模式深放在程序内部，这显然不是一种令人满意的安排。我们以 UNIX 程序 grep 为范本修改程序，使其可通过命令行的第一个参数来指定待匹配的模式。

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: 打印与第 1 个参数指定模式相匹配的行 */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

标准库函数 strstr(s, t) 返回字符串 s 中首次出现字符串 t 的位置的指针，或在未出现时返回 NULL。它在 <string.h> 中声明。

这一命令行参数模型现在可被着意加强来进一步说明指针的构造。假设我们希望有两个可

选参数。第一个表示“打印所有不匹配模式的行”；第二个表示“在每一个打印行之前加上该行的行号”。

在 UNIX 系统上 C 程序有一个共同的约定：用减号开头的参数是一个可选标记或参数。如果我们选择 -x（表示“除外”）提示反意执行，以及 -n（“编号”）来要求加上行号，那么命令

```
find -x -n 模式
```

将会打印所有不匹配模式的行，并在前面加上该行的行号。

可选参数应该允许以任意的次序出现，而程序的其他部分应该与参数出现的个数无关。并且，如果可选参数能够合并则会便于使用，就如

```
find -xn 模式
```

这是对应的程序：

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: 打印与第一个（非可选）参数相匹配的行 */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: 非法选项 %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("用法: find -x -n 模式\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

在每一个可选参数前 argc 被递减同时 argv 被递增。在循环的最后，如果没有错误，argc 表示有多少参数没有处理，同时 argv 指向第一个未处理的参数。因此 argc 应该为 1，而*argv 则应该指向待匹配模式。请注意*++argv 是一个指向参数串的指针，所以(*++argv)[0]是该参数串的第一个字符。（另一种有效形式是**++argv。）因为[]比*和++结合得更紧，所以圆括号是必须的；否则表达式将被看作*++(argv[0])。事实上这正是我们在内层循环中使用的形式，其任务是沿着特定的参数字符串前进，在内层循环中，表达式*++argv[0]增加的是指针 argv[0]！

很少有人使用比它们更加复杂的指针表达式，如果有这种情况，将它们分作两步或三步会更直观一些。

练习 5-10. 编写程序 expr，它对来自命令行的一个逆波兰式求值，这里每个运算符或操作数都是分开的参数，例如：

```
expr 2 3 4 + *
```

对 2×(3＋4) 求值。

练习 5-11. 修改程序 entab 和 detab（第 1 章作为练习编写的程序），使其接受一组制表符停止位作为参数。如果没有参数则使用缺省的制表设置。

练习 5-12. 扩展 entab 和 detab，使其接受缩略形式

```
entab -m +n
```

其表示从第 m 列开始，每 n 列一个制表符停止位，选择（对使用者而言）方便的缺省行为。

练习 5-13. 编写程序 tail，它打印其输入的最后 n 行。n 设有缺省值，比如说为 10，但它可通过一个可选参数来改变，这样

```
tail -n
```

打印最后的 n 行。无论输入以及 n 值是如何的不合理，此程序都应该具有合理的行为。编写此程序让它最好地利用可获得的存储；文本行应该像 5.6 节的排序程序中那样存放，而不是放在固定大小的二维数组中。

5.11 函数指针

在 C 语言中，函数本身不是变量，但定义函数的指针是可以的，它可被赋值、放在数组内、传递给函数、由函数返回等等。我们将通过修改本章先前编写的排序程序来对此进行阐述。修改后的函数如果给出可选参数 -n，它将以数值序代替字母序进行输入行的排序。

一个排序通常包括三个部分——用于确定任意两个对象次序的比较部分，用于倒转它们的次序的交换部分，以及进行比较和交换直到所有对象已经有序的排序算法部分。排序算法独立于比较和交换操作，所以通过传递不同的比较和交换程序给排序算法程序，我们就可以安排不同准则进行排序。这正是我们在新排序程序中所采用的方法。

像之前那样，两行的字母序比较通过 strcmp 来完成；我们还需要一个 numcmp 例程基于数值来比较两行并且返回与 strcmp 相同类别的比较结果标志。这些函数在 main 函数之前声明，一个指向其中某一恰当函数的指针被传递到 qsort 中。我们略去了参数的错误处理，这样我们可以集中精力考虑主要问题。

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

```
#define MAXLINES 5000      /* 可被排序的最大行数 */
char *lineptr[MAXLINES];   /* 指向文本行的指针 */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
       int (*comp)(void *, void *));
int numcmp(char *, char *);

/* 将输入行进行排序 */
main(int argc, char *argv[])
{
    int nlines;          /* 输入的行数 */
    int numeric = 0;     /* 如果按数值排序则为 1 */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
           (int (*)(void*, void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

在对 qsort 的调用中，strcmp 和 numcmp 是函数的地址。由于已经知道它们是函数，& 运算符不是必需的，就像在数组名之前不需要用 & 一样。

改写后的 qsort 可以处理任意的数据类型，而不是仅能处理字符串。按照 qsort 函数原型的表述，qsort 的参数包括一个指针数组、两个整数、以及一个有 2 个指针参数的函数。通用指针类型 void * 用于这些指针参数。任何指针都可被转换为 void * 并转换回来，且不丢失信息，所以我们可用强制转换为 void * 的参数来调用 qsort。比较函数的参数使用了这种特意的函数参数变换方式。通常说来这对实际的表述没有影响，但需要确认编译器认为这些都 OK。

```
/* qsort: 将 v[left]…v[right]按递增次序排列 */
void qsort(void *v[], int left, int right,
       int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right)  /* 如果数组包含元素小于两个 */
        return;         /* 则什么也不做 */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
```

```
}
```

这些声明应该仔细研究一下。qsort 的第四个参数是

```
int (*comp)(void *, void *)
```

它表示 comp 是一个函数，该函数有两个 void * 参数，并返回一个 int 值。

在代码行

```
if ((*comp)(v[i], v[left]) < 0)
```

中 comp 的使用与是符合其声明的：comp 是一个函数的指针，*comp 是该函数，而

```
(*comp)(v[i], v[left])
```

则是对此函数的调用。这里括号是需要的，这样成员才能够按正确的方式关联；如果没有它们，

```
int *comp(void *, void *)  /* 错了！ */
```

表示 comp 是一个函数，其返回一个 int 指针，和原意大不相同。

我们已经给出了 strcmp，它对两个字符串进行比较。这里介绍 numcmp，其比较两个字符串的前导数值（该值通过调用 atof 计算得到）：

```
#include <stdlib.h>

/* numcmp: 按数值比较 s1 与 s2 */
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

用于交换两个指针的 swap 函数和本章之前所展示的完全一样，只是声明变成了 void *。

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

还有多种选项可以加到这个排序程序中,其中一些构成了有挑战性的习题。

练习 5-14. 修改排序程序使其可处理标记 -r，它表明按逆序（降序）进行排序。请确认 -r 可以与 -n 一起工作。

练习 5-15. 增加 -f 选项，将大小写合并考虑，这样在排序时大小写不作区分；例如，a 和 A 是相等的。

练习 5-16. 增加 -d（“字典序”）选项，使程序仅比较字母、数字和空格。请确保它可以与 -f 协同工作。

练习 5-17. 增加字段处理功能，这样可以按多个字段对行进行排序，每个字段根据独立的选项

集完成排序。（本书的索引以 -df 对索引类型排序，并以 -n 对页码排序。）

5.12 复杂声明

C 语言中声明所用的语法有时会倍受批评，尤其是那些包含函数指针的声明语法。该语法试图让一个声明与对应的用法相匹配；简单情况下它工作得很好，但较复杂的情况下则可能令人困惑，这可能是由于声明不能按照从左到右的顺序理解，也可能是因为使用了太多的圆扩号。声明

```
int *f();     /* f: 函数返回指向 int 型的指针 */
```

与

```
int (*pf)(); /* pf: 指向返回 int 的函数的指针 */
```

之间的区别就说明了这个问题：* 是一个前缀运算符，它的优先级低于 ()，因此必须用圆括号来强制恰当的结合。

虽然真正复杂的声明在实际中极少出现，但是，知道如何理解它们并且在必要时如何创建它们仍然重要。合成声明的一个好方法是使用 typedef 分步骤完成。typedef 将在 6.7 节讨论。作为另一种选择，本节将给出一对程序，它们将有效的 C 语言声明和对应的文字描述进行互相转换。转换的文字描述可以从左至右地解读。

第一个程序 dcl 要复杂一些。它将 C 语言声明转换为文字描述，就如下面的例子：

```
char **argv
    argv: pointer to pointer to char  /* 指向字符指针的指针 */
int (*daytab)[13]
    daytab: pointer to array[13] of int
    /* 指向含 13 个 int 元素的数组的指针 */
int *daytab[13]
    daytab: array[13] of pointer to int
    /* 包含 13 个指向 int 的指针元素的数组 */
void *comp()
    comp: function returning pointer to void
    /* 返回 void 型指针的函数 */
void (*comp)()
    comp: pointer to function returning void
    /* 指向返回 void 的函数的指针 */
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returening char
    /* x 是一个函数，其返回值为指向数组的指针，
    该数组元素是指向返回 char 的函数的指针 */
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
    /* x 是包含 3 个元素的数组，该数组元素是指向一个函数的指针，
    该函数返回值为指向包含 5 个 char 元素的数组的指针 */
```

dcl 基于声明符所用语法编写，该语法的精确描述在附录 A 的 8.5 节；这里是一个简化形

式：

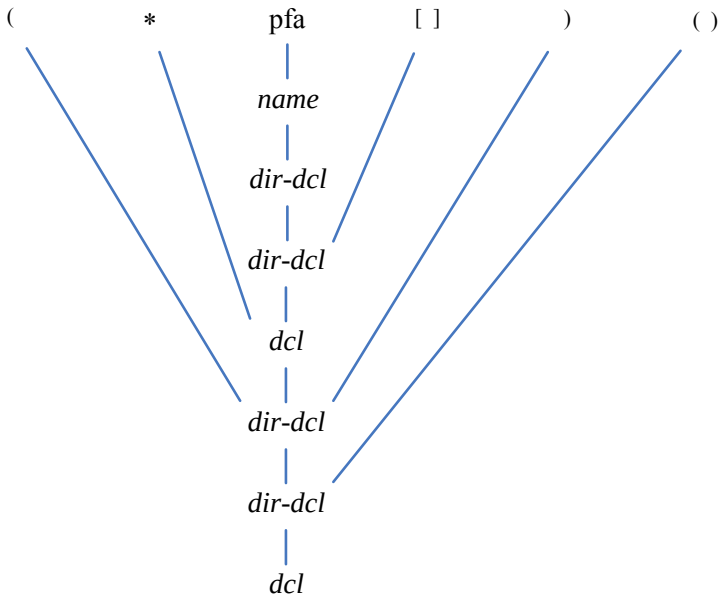
```
dcl:          optional *'s direct-dcl
direct-dcl:   name
              (dcl)
              direct-dcl()
              direct-dcl[optional size]
```

用文字描述就是：*dcl* 是一个 *direct-dcl*，该 *direct-dcl* 也可能前缀有一个或多个 * 。*direct-dcl* 是 *name*（名字），或者是用圆括号括起来的 *dcl*，或者是其后紧跟一个圆括号的 *direct-dcl*，或者是其后紧跟一个方括号（括号内可能包含大小）的 *direct-dcl*。

这个语法可用来解析声明。例如，考虑如下声明：

```
(*pfa[])()
```

pfa 将被辨识为一个 *name*，即一个 *direct-dcl*。那么 pfa[]同样是一个 *direct-dcl*。这样*pfa[]就被辨识为一个 *dcl*，因此 (*pfa[]) 是一个 *direct-dcl*。这样 (*pfa[])() 就是一个 *direct-dcl* 也即一个 *dcl*。我们还可使用一个解析树来演示这一解析过程（其中 *direct-dcl* 被缩写为 *dir-dcl*）：



dcl 程序的核心是一对函数—dcl 和 dirdcl，它们根据上述语法对声明进行解析。由于该语法是递归定义的，当这对函数辨识声明的片段时将互相递归调用；这样的程序被称为递归下降解析器。

```
/* dcl: 解析一个声明 */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; )    /* 计算 * 的个数 */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}

/* dirdcl: 解析一个直接声明 */
void dirdcl(void)
```

```
{
    int type;

    if (tokentype == '(') {          /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            printf("error: missing )\n");
    } else if (tokentype == NAME) /* 变量名字 */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " function returning");
        else {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        }
}
```

由于这些程序的目的在于演示，而不要求尽善尽美，因此 dcl 有明显的限制。它仅可处理如 char 或 int 这样的简单数据类型。它没有处理函数中的参数类型，或者如 const 这样的限定符。不合逻辑的空白符会造成解析错乱。错误恢复方面它没有做太多工作，所以无效声明也会造成解析错乱。这些方面的改进将留作练习。

下面是全局变量和主函数：

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

#define  MAXTOKEN  100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;              /* 解析中最新遇到的标记的类型 */
char token[MAXTOKEN];       /* 最新遇到的标记字符串*/
char name[MAXTOKEN];        /* 标识符名称 */
char datatype[MAXTOKEN];    /* 数据类型 = char、int 等等 */
char out[1000];             /* 输出字符串 */

main()     /* 将声明转换为文字 */
{
    while (gettoken() != EOF) {    /* 输入行中的第 1 个标记 */
        strcpy(datatype, token);   /* 是数据类型 */
        out[0] = '\0';
        dcl();        /* 解析输入行的剩余部分 */
        if (tokentype != '\n')
            printf("syntax error\n");  /* 语法出错 */
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

函数 gettoken 跳过空格符和制表符，找到输入中的下一个标记；“标记”可以是名字、一对圆括号、一对方括号（可能包含数字），或者是其他的任意单个字符。

```
int gettoken(void)  /* 返回紧接着的标记 */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}
```

getch 和 ungetch 在第 4 章中讨论过。

反向的转换要简单一些，尤其是不需要担心生成多余的圆括号。程序 undcl 将类似“x 是一个函数，该函数返回一个指向数组的指针，该数组元素是指向返回 char 的函数的指针”的文字描述，我们将其表述为

```
x () * [] * () char
```

转换为

```
char (*(*x())[])()
```

这种缩写的输入语法使得我们可以复用 gettoken 函数。undcl 也使用了与 dcl 相同的外部变量。

```
/* undcl: 将文字描述转换为声明 */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
```

```
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
```

练习 5-18. 使 dcl 可以从输入错误中恢复。

练习 5-19. 修改 undcl 使其不会增加冗余的圆括号到声明中。

练习 5-20. 扩展 dcl，使其可处理包含函数参数类型、const 等限定符等等的声明。

第 6 章 结构

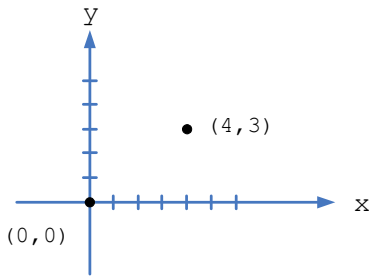
结构是一个或多个变量的集合，这些变量的类型可能不同，为方便而将它们并为一组用一个单独的名字操控。（某些语言中结构被称为“记录（record）”，例如著名的 Pascal。）结构有助于组织复杂的数据，尤其在大型程序中，因为它们允许将相关联的一组变量作为一个单元使用，而不是作为分离的对象处理。

职员薪资表是说明结构的一个传统例子：雇员用一组属性来描述，例如名字、住址、社会保障号码、薪金等等。其中一些属性可能又是结构：名字有多个组成部分；相应地，住址、甚至薪金也是这样。另一个对 C 语言更典型的例子来自图形学：一个点是一对坐标值，一个矩形是两个点等等。

ANSI 标准作出的主要修改是定义了结构赋值——结构可被复制、赋值、传递给函数、以及被函数返回。这一特性大多数编译器已经支持多年，现在被精确地定义下来。结构和数组类型的自动变量现在也能够被初始化了。

6.1 结构的基础知识

让我们创建几个适用于图形学的结构。最基本的对象是点，我们假定它有 x 坐标值和 y 坐标值，且都是整数。



这两个分量可以被放到一个像这样声明的结构中：

```
struct point {
    int x;
    int y;
};
```

关键字 struct 引入结构声明，结构声明是被花括号括上的一列声明。在关键字 struct 之后可带有一个被称为**结构标记**的可选名字（在这里即 point）。该标记命名了该类型的结构，并在此后可做为声明中花括号部分的缩写使用。

在结构内命名的变量被称为**成员**。结构成员、标记和普通变量（即非成员变量）可以拥有相同的名字而不冲突，因为它们总是能够通过上下文进行区分。并且，相同的成员名字可以出现在不同的结构中，尽管就编程风格而言通常只有关系紧密的对象才使用相同的名字。

一个 struct 声明定义了一个类型。在标志成员列表结束的右花括号后可以带有一列变量，就像任何基本类型那样。譬如，

```
struct { … } x, y, z;
```

与

```
int x, y, z;
```

在语法上可以类比：两者都将 x、y 和 z 声明为所命名的类型，并且为它们预留了存储空间。

不带变量列表的结构声明并不预留存储空间；它只是描述了结构的模版（即一个结构的形态），然而如果结构声明已被加上标记，该标记则可用于此后结构实例的定义。例如，按照上面的 point 声明，

```
sturct point pt;
```

定义了一个 struct point 类型的结构变量 pt。一个结构可通过在其定义之后紧随的初始值列表来进行初始化。每一初始值都是赋给结构成员的常量表达式：

```
struct point maxpt = { 320, 200 };
```

一个自动结构变量也可以通过赋值或者调用某个返回同类型结构的函数来进行初始化。

在表达式中，通过如下构成形式对某个特定结构的成员进行引用：

```
结构名称.成员
```

结构成员运算符“.”连接结构名称和成员名称。例如，打印 pt 这个点的坐标可以用

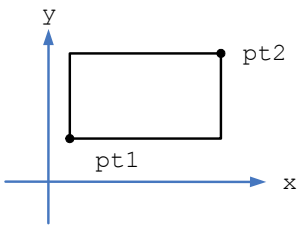
```
printf("%d, %d", pt.x, pt.y)
```

计算从原点 (0,0) 至 pt 的距离可以用

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

结构可以嵌套。矩形的一种表示方法就是代表其对角的两个点：




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

结构 rect 包含了两个 point 结构。如果我们将 screen 声明为

```
struct rect screen;
```

则

```
screen.pt1.x
```

引用了 screen 的成员 pt1 的 x 坐标。

6.2 结构和函数

对结构的操作中，仅有“以结构为单元进行复制或赋值”、“用 & 获取结构地址”、以及“访问结构成员”等操作是合法的。复制和赋值也包括向函数传递参数以及从函数返回值。结构不可以进行比较。可以用一个成员常值列表初始化结构；一个自动结构变量也可以通过赋值进行初始化。

我们通过编写一些操控点和矩形的函数来深入了解结构。至少有三种可能的编写方法：分别传递结构的组件、传递整个结构、或者传递结构的指针。每种都有其优点和缺点。

第一个函数，makepoint，其输入两个整数并返回一个 point 结构：

```
/* makepoint: 以 x 和 y 为坐标构造一个点 */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

注意参数名和同名的成员之间没有冲突；实际上名字的重复强调了它们的联系。

现在 makepoint 可用于动态地初始化任何结构，或者为函数提供结构参数：

```
struct rect screen;
struct point middle;
struct point make point(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, XMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
              (screen.pt1.x + screen.pt2.x)/2);
```

接下来是对点进行算术运算的函数集。例如：

```
/* addpoint: 两个点相加 */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

在这里参数和返回值都是结构。我们之所以递增 p1 中的成员而不是显式地使用一个临时变量，目的是强调结构参数像其他参数那样是按值传递的。

另一个例子是函数 ptinrect，其测试一个点是否在一个矩形之内，这里我们采用了惯例，即矩形包含它的左侧边和底边，而不包含顶边和右侧边。

```
/* ptinrect: 如果 p 在 r 之内，返回 1，否则返回 0 */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

这里假定矩形按照规范形式表述，即 pt1 的坐标值小于 pt2 的坐标值。接下来的函数返回一个确保为规范形式的矩形：

```
#define min(a, b)   ((a) < (b) ? (a) : (b))
#define max(a, b)   ((a) > (b) ? (a) : (b))

/* canonrect: 将矩形的坐标规范化 */
struct rect canonrect(struct rect r)
{
    struct rect temp;
```

```
    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

如果需要将一个大型结构传递给一个函数，通常传递指针要比复制整个结构的效率更高。结构指针就像其他变量的指针一样。声明

```
struct point *pp;
```

表示 pp 是一个 struct point 结构类型的指针。如果 pp 指向一个 point 结构，那么*pp 即是这个结构，(*pp).x 和(*pp).y 则是其成员。要使用 pp，我们可能编写如下代码：

```
struct point origin, *pp;

pp = &origin;
printf("origin is (%d, %d)\n", (*pp).x, (*pp).y);
```

(*pp).x 中的圆括号是必须的，因为结构成员运算符 . 的优先级高于 * 。表达式 *pp.x 的意义是 *(pp.x) ，由于这里 x 并不是一个指针，因此该表达式是非法的。

结构指针使用得相当频繁，以至于 C 语言提供了另一种简写记法。如果 p 是一个结构指针，那么

```
p->结构成员
```

即引用了指定的成员。(运算符 -> 为一个负号紧接着 > 号。)所以上面的代码可以改写为

```
printf("origin is (%d, %d)\n", pp->x, pp->y);
```

运算符 . 和 -> 都为自左向右结合，因此假如有

```
struct rect r, *rp = &r;
```

那么以下四个表达式是等价的：

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

结构运算符 . 和 -> 与用于函数调用的 () 和用于下标的 [] 一起，处于运算符优先等级的顶端，因而结合得非常紧密。例如，给定声明

```
struct {
    int len;
    char *str;
} *p;
```

则

```
++p->len
```

递增的是 len，而不是 p，因为其隐含的带圆括号的表达式是++(p->len)。圆括号可用于改变结合关系：(++p)->len 在访问 len 之前先递增 p，而(p++)->len 在之后才递增 p。(最后这组圆括号不是必需的。)

同样地，*p->str 获取 str 指向的内容；*p->str++ 在访问它指向的内容之后将 str 递增（就像*s++ 那样)；(*p->str)++递增 str 指向的内容；而 *p++->str 在访问 str 指向的内容之后将 p 递增。

6.3 结构数组

请考虑编写一个程序来计算每个 C 关键字出现的次数。我们需要一个字符串数组来存放关键字的名字，和一个整数数组来存放计数。一种可能是使用两个并列的数组，keyword 和 keycount，即如

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

但正是数组并列这一事实提示了另一种不同的构成方式——结构数组。每个关键字项都是一对变量：

```
char *word;
int count;
```

因此有一组这样的变量对。结构声明

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

声明了一个结构类型 key，定义了此结构类型的数组 keytab，并为它们预留了存储空间。此数组的每个元素都是一个结构。这一声明也可写为

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

由于结构 keytab 包含了一组名字常值，最容易的方法是将其设为外部变量，并在定义时一次性初始化完毕。此结构的初始化类似于早先的例子——定义之后跟随着花括号内的一组初始化值：

```
struct key {
    char *word;
    int count;
} keytab[] = {
    “auto”, 0,
    “break”, 0,
    “case”, 0,
    “char”, 0,
    “const”, 0,
    “continue”, 0,
    “default”, 0,
    /* … */
    “unsigned”, 0,
    “void”, 0,
    “volatile”, 0,
    “while”, 0,
};
```

与结构成员对应，初始化值两两成对而列。若按每“行”（即每个结构）将初始化值用花括号括起来则会更加精确。如

```
{ “auto”, 0 },
{ “break”, 0 },
{ “case”, 0 },
```

…

但是当初始化值是简单变量或字符串，且所有值都给出时，内部的花括号不是必需的。与通常一样，如果数组 keytab 提供了初始化值并且 [] 内为空，那么其项数将通过计算确定。

此关键字计算程序首先是 keytab 的定义。主函数通过重复调用函数 getword 来读取输入，每次调用获取一个词。每个词都在 keytab 中进行查找，使用的是在第 3 章中编写的二分查找函数。关键字列表必须按照升序排列。

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

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* 计算 C 的关键字 */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch: 在 tab[0]至 tab[n-1]中查找关键字 */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

稍后我们将描述函数 getword；现在只用知道 getword 每次调用找到一个词，并将其拷贝到该函数第一参数所指定的数组中。

常量 NKEYS 为 keytab 中关键字的个数。虽然可以用手工计数，但通过机器来做要容易和

安全得多，在关键字列表可能改变时尤其如此。一种可能是用一个空指针作为初始化列表的结尾，这样可按 keytab 循环直到找到结尾。

但是这不是必需的，由于数组的大小在编译时会完全确定。数组的大小是其一项的大小乘以项数，所以关键字数组的项数就是

```
keytab 的大小 / 结构 key 的大小
```

C 提供了一个叫做 sizeof 的编译时用的一元运算符，其可用于计算任一对象的大小。表达式

```
sizeof 对象
```

以及

```
sizeof (类型名)
```

生成一个整数，其等于所指定对象或类型的按字节计的大小。（严格地说，sizeof 产生一个类型为 size_t 的无符号整数值，该类型在头文件 <stddef.h> 中定义。）对象可以是变量、数组或者结构；类型名可以是像 int 或 double 等的基本类型名，或者是结构或指针等派生类型名。

在我们的例子中，关键字的数目就是数组的大小除以其中一个元素的大小。此计算方法被用在一个设置 NKEYS 值的 #define 声明里：

```
#define NKEYS    (sizeof keytab / sizeof(struct key))
```

它的另一种写法是用数组大小除以一个指定元素的大小：

```
#define NKEYS    (sizeof keytab / sizeof keytab[0])
```

这样写的好处是假如类型变化也不需要进行改变。

sizeof 不能被用于 #if 行中，因为预处理器并不解析类型名。但 #define 中的表达式却不由预处理器处理，因此这里的代码是合法的。

现在考虑 getword 函数。我们编写了一个比本程序所需功能更为通用的 getword 函数，但其并不复杂。getword 从输入中获取最新到来的“词”，这里的词可以是以字母开头的由字母或数字组成的串，或者是单个非空白字符。函数返回值为该词的第一个字符、或者为表示文件结束的 EOF、或者是非字母的字符本身。

```
/* getword: 从输入中得到接下来的词或字符 */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c))
        *w = '\0';
        return c;
    }
    for ( ; --lim > 1; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
```

```
}
```

getword 使用了第 4 章编写的 getch 和 ungetch。当一个字母数字标记收集结束时，getword 已经多取了一个字符，于是调用 ungetch 将该字符推回输入以便于下一次调用。getword 还使用 isspace 来跳过空白符、使用 isalpha 来识别字母、使用 isalnum 来识别字母和数字；它们都来自于标准头文件 <stype.h>。

练习 6-1. 我们的 getword 版本不能正确地处理下划线、字符串常量、注释、以及预处理控制行。请编写一个更好的版本。

6.4 结构指针

为了阐明对结构指针与结构数组的一些相关考虑，让我们再次编写关键字计数程序，这次用指针来替代数组下标。

外部声明 keytab 不用改变，但是 main 和 binsearch 都需要修改。

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* 计算 C 的关键字; 指针版本 */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if((p = binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", key->count, p->word);
    return 0;
}

/* binsearch: 在 tab[0]至 tab[n-1]中查找关键字 */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high - low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
```

```
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

这里有几件事情值得一提。首先，binsearch 的声明必须标明它返回一个 struct key 指针，而不是返回整数；这在 binsearch 和其函数原型中都要进行声明。如果 binsearch 找到了关键字，它返回指向该关键字的指针；如果没有找到，则返回 NULL。

其次，keytab 的元素现在通过指针来访问。这一点要求 binsearch 有明显的改变。

low 和 high 的初始化值现在分别是指向表开头的指针和指向表末尾之后第一个位置的指针。

中间元素的计算不再是

```
mid = (low + high) / 2  /* 错误 */
```

那么简单了，因为两个指针相加是非法的。但是，它们相减却是合法的，high - low 即为元素的个数，于是

```
mid = low + (high - low) / 2
```

使 mid 指向 low 与 high 中间的位置。

最重要的改变是调整算法来确保其不会产生非法指针或者访问超出数组范围的元素。问题在于 &tab[-1] 和 &tab[n] 都在数组 tab 的范围之外。前者是完全非法的，而解引用后者是非法的。然而，C 语言的定义保证了涉及数组尾部之后第一个元素的指针运算（即&tab[n]）会正常地工作。

在 main 中我们用

```
for (p = keytab; p < keytab + NKEYS; p++)
```

如果 p 是指向结构的指针，对 p 的运算将考虑到结构的大小，因此 p++ 使得 p 增加恰当的大小而指向结构数组的下一个元素，测试部分也会让循环正确地停止。

然而，不要假定结构的大小就等于其成员的大小之和。由于不同对象的对齐要求，结构中可能存在未命名的“空洞”。例如，如果 char 是 1 字节、int 是 4 字节，那么结构

```
struct {
    char c;
    int i;
};
```

或许需要 8 个字节，而不是 5 个。sizeof 运算符返回正确的值。

最后，附带一个关于程序格式的问题：当函数返回像结构指针这样的复杂类型时，就如

```
struct key * binsearch(char *word, struct key *tab, int n)
```

语句中的函数名在文本编辑器中会较难浏览和查找，另一种时常使用的风格是：

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

这是个人喜好问题；选择你所喜欢的形式并一直使用它。

6.5 自引用结构

假设我们需要处理更为通用的问题——统计输入中所有单词出现的次数。由于单词列表无法事先知道，我们不能方便地将其排序并使用二分查找；也不能在每个单词到来时使用线性查找，看它是否已经出现过，这样程序会非常耗时。（更精确地说，其运行时间很可能随着输入单词数的增加而线性增长。）那我们如何组织数据才能高效地处理一个任意单词的列表呢？

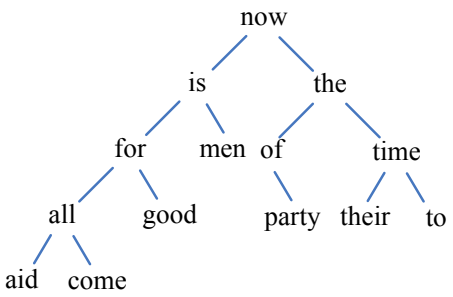
一个解决方案是在每个单词到来时将其按序排放到正确的位置，从而始终将目前已出现的单词保持有序。这不应该通过在一个线性数组中平移单词来完成，虽然可以那样做，但也会非常耗时。我们代之使用一种称为二叉树的数据结构。

这颗树对于每个不同的单词都有一个“节点”；每个节点包含

一个指向单词文本的指针
一个出现次数的计数
一个指向左孩子节点的指针
一个指向右孩子节点的指针

没有节点会有多于两个孩子；它或许只有一个或零个孩子。

节点按这样的形式组织：任一节点的左子树仅包含那些（按字母序）小于该节点单词的单词，而右子树则仅包含比其大的单词。下面就是句子“now is the time for all good men to come to the aid of their party”通过插入每个遇到的单词而构建成的树：



为了确认一个新碰到的单词是否已经存在于这棵树中，需要自根开始，将此新单词与该节点的单词相比较。假如匹配，问题的答案是肯定的。如果该新单词较树中的单词小，则到节点的左孩子继续查找，否则到其右孩子处查找。如果要查找的方向没有孩子，则该新单词不在树中，这个空位就是加入该新单词的恰当位置。此过程是递归的，因为自任一节点的查找也包括了在它某一孩子中的查找。相应地，插入和打印采用递归过程也会最自然。

回过来考虑节点的描述问题，节点可方便地表述为包含 4 个成员的结构：

```
struct tnode {           /* 树的节点 */
    char *word;          /* 指向单词的文本 */
    int count;           /* 出现的次数 */
    struct tnode *left;     /* 左孩子 */
    struct tnode *right;    /* 右孩子 */
};
```

节点的这种递归声明或许看起来令人提心吊胆，但它是正确的。一个结构包含它自己的实例是非法的，但

```
struct tnode *left;
```

将 left 声明为 tnode 的指针，而不是 tnode 本身。

偶尔地，人们会需要自引用结构的一种变体：两个结构相互引用。其构建方式如下：

```
struct t {
    …
    struct s *p;  /* p 指向一个 s */
};
struct s {
    …
    struct t *q;  /* q 指向一个 t */
};
```

有了如 getword 等几个已经编写过的函数支持，整个程序的代码出人意料地小。主函数使用 getword 读取单词，并使用 addtree 将它们放置到树中。

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* 单词频度统计 */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

addtree 是递归函数。单词被 main 函数递交到树的顶层（根）。在每一层，该单词与存放在节点内的单词相比较，并通过递归调用 addtree 下传到其左子树或右子树，最后该单词要么与树中已有的某单词相匹配（在这种情况下计数会被递增），要么碰到一个空指针，这表明必须创建一个节点并将其加入此树中。如果创建了新的节点，addtree 会返回一个指向它的指针，该指针会被加到它的父节点中。

```
struct tnode *talloc(void);
char *strdup(char  *);

/* addtree: 将一个包含 w 的节点加到 p 或 p 之下的位置 */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {     /* 出现了新单词 */
        p = talloc();    /* 创建一个新节点 */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;      /* 重复出现的单词 */
    else if (cond < 0)   /* 小于则进入左子树 */
        p->left = addtree(p->left, w);
```

```
    else                    /* 大于则进入右子树 */
        p->right = addtree(p->right, w);
    return p;
}
```

新节点的存储空间通过函数 talloc 获得，该函数返回一个指针，指向适合存放单个树节点的空间。新单词通过 strdup 拷贝到一个隐藏空间。（我们很快就会讨论这些函数。）节点上的计数被初始化，两个孩子被设为空。这部分代码仅在新节点被加入时在树的叶节点处执行。我们省略了 strdup 及 talloc 的返回值出错检查（这样做并不明智）。

treeprint 按序打印这棵树；对于每个节点，它打印左子树（所有小于此节点单词的单词），然后是此节点的单词本身，然后是右子树（所有较大的单词）。如果你还拿不准递归是如何工作的，那么请模拟如上所述的 treeprint 对树的操作。

```
/* treeprint: 按序打印树 p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

一个实用的注释：如果由于单词没有以随机的大小次序到达而使树变得“不平衡”，那么程序运行的时间会大大增加。最坏的情况是，假如单词已经有序，那么此程序就是对线性查找的一个高代价的模拟。存在一些不受此最坏情况影响的二叉树通用构建法则，但此处我们不作讨论。

在我们结束这个例子之前，还值得简短讨论一下与存储空间分配器相关的一个枝节问题。很显然，在一个程序中只期望存在一个存储空间分配器，即便它要分配不同类型的对象。但是假如一个分配器需要处理，举例来说，对字符指针的请求和对 struct tnode 指针的请求，就出现了两个问题：第一，如何满足多数实际机器都存在的要求，即某些类型的对象必须满足一些对齐限制（例如，整数常常必须位于偶地址）？第二，什么样的声明才能应对一个分配器必须按需返回不同类型的指针这一实际问题？

对齐要求一般容易满足，这要以一些空间浪费作为代价，以确保分配器总是返回满足所有对齐限制的指针。第 5 章的 alloc 不提供任何对齐保证，因此我们将使用标准库函数 malloc，它可以做到这点。在第 8 章中我们将展示一种实现 malloc 的方法。

类似malloc的函数的类型声明对于任何严格对待类型检查的语言来说都是一个恼人的问题。在C中，正确的方法是将malloc声明为返回一个void型指针，并显式地将指针强制转换为所期望的类型 [①]。malloc以及相关的函数在标准头文件 <stdlib.h> 中声明。这样talloc就可被编写为：

```
#include <stdlib.h>

/* talloc: 构建一个 tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

[①] 就 1988-1989 ANSI/ISO 标准而言，void * 到 ALMOSTANYTYPE * 的转换是自动的，因此这种强制转换并不必要；而且如果 malloc 或其替代者未声明为返回 void *，显式的强制转换还会掩盖这一不经意的错误。另一方面，在 ANSI 标准之前，这种强制转换则是必须的。

strdup 所做的只是将其参数给出的字符串复制到一个安全的地方，此安全空间通过调用 malloc 获得：

```
char *strdup(char *s)   /* 构建一个 s 的副本 */
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 为了存放'\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

若没有空间可以分配，malloc 返回 NULL；strdup 将此值传回，把错误处理留给它的调用者。

通过调用 malloc 获得的存储空间可以通过调用 free 释放以供重新分配；请参见 7、8 两章。

练习 6-2. 编写一个程序，它读取一个 C 程序并按字母序将变量名分组打印出来，前 6 个字符相同（而之后有所不同）的变量分为一组。不要计算字符串和注释内的单词。将 6 改为一个参数，可以通过命令行进行设置。

练习 6-3. 编写一个交叉引用程序，打印一个文档中所有单词的列表，并打印每个单词出现的行的行数列表。去掉如"the"、"and"等干扰性单词。

练习 6-4. 编写一个程序，将其输入中的不同单词按出现频度降序排列打印出来。每个单词之前是它出现的次数。

6.6 表的查询

在本节我们将编写一个表查询软件包的内部构成程序，以此阐述结构的更多方面。这是有可能在微处理器的或编译器的符号表管理程序中找到的典型代码。例如，考虑 #define 语句，当碰到如

```
#define IN 1
```

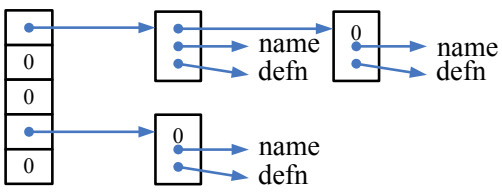
的行时，名字 IN 以及替换文本 1 被存放到一个表中。之后，当名字 IN 出现在如

```
state = IN;
```

的声明中时，它必须被替换为 1。

有两个函数用于操作名称和替换文本行。install(s, t) 将名字 s 和替换文本 t 记录在表中；s 和 t 都是字符串。lookup(s) 在该表中查找 s，并返回 s 所在位置的指针，如果它不在表中，则返回 NULL。

所用算法是哈希查找法——到来的名字被转换为一个小的非负整数，该整数作为一个指针数组的下标。每个数组元素指向一个（描述名字的块的）链表的开头，该链表中的名字都具有对应的哈希值（即该下标）。如果没有名字被哈希为该下标值，那么对应的数组元素为 NULL。



块列表中的块是一个结构，其包含了名字的指针、替换文本的指针以及列表中后续块的指针。后续指针为空则标志这个列表结束。

```
struct nlist {       /* 表项: */
    struct nlist *next; /* 链表中的后续项 */
    char *name;         /* 所定义的名字 */
    char *defn;         /* 替换文本 */
};
```

指针数组即

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE];  /* 指针表 */
```

lookup 和 install 都要用到哈希函数，该函数通过将字符串的每一字符值与由该字符之前其余字符混杂而成的值相加，最后将所得混合值按数组大小取模，并返回余值。这不是可能的最优哈希函数，但是它既短小又有效。

```
/* hash: 形成字符串 s 的 hash 值 */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

无符号运算保证了哈希值是非负值。

此哈希过程产生出数组 hashtab 中的一个起始下标；如果被哈希的字符串会在某个位置找到，那么它一定位于此处起始的块列表中。这个查找通过 lookup 来执行，如果 lookup 找到了已经存在的项，它返回指向该项的指针，否则返回 NULL。

```
/* lookup: 在 hashtab 中查找 s */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;    /* 找到 */
    return NULL;          /* 未找到 */
}
```

lookup 中的 for 循环是遍历链表的标准惯用法：

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    …
```

install 使用 lookup 来确认待加入的名字是否已经存在；如果存在，新的定义将替换掉旧的定义。否则，一个新项将被创建。如果由于任何理由没有保存新项的空间，install 将返回 NULL。

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: 将 (name, defn) 放到 hashtab 中 */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
```

```
    unsigned hashval;

    if ((np = lookup(name)) == NULL) {   /* 未找到 */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else    /* 已经有了 */
        free((void *) np->defn);  /* 将原有的 defn 释放 */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

练习 6-5．编写一个 undef 函数，它从 lookup 与 install 维护的表中删除某个名字及定义。

练习 6-6．基于本节的函数，实现一个简单的（譬如没有参数）适用于 C 程序的 #define 处理器版本。你可能还会发现 getch 和 ungetch 有帮助。

6.7 类型定义（Typedef）

C 语言提供了一个称为 typedef 的功能用于创建新的数据类型名。例如，声明

```
typedef int Length;
```

使得名字 Length 成为 int 的同义词。此 Length 类型可被用于声明、强制转换等等，用法与 int 类型完全一致：

```
Length len, maxlen;
Length *lengths[];
```

类似地，声明

```
typedef char *String;
```

使 String 成为 char *（即字符指针）的同义词，随后它即可用于声明和强制转换：

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

注意 typedef 所声明的类型出现在变量名的位置，而不是直接跟在关键字 typedef 之后。typedef 在语法上与 extern、static 等存储类相似。我们让 typedef 定义的名字用大写开头，以示差别。

作为一个更复杂的例子，我们可将本章先前所述的树节点用 typedef 来定义：

```
typedef struct tnode *Treeptr;

typedef struct tnode {  /* 树节点 */
    char *word;         /* 指向单词的文本 */
    int count;          /* 出现次数 */
    Treeptr left;       /* 左孩子 */
    Treeptr right;      /* 右孩子 */
} Treenode;
```

这里创建了两个新的类型关键字 Treenode（一个结构）和 Treeptr（该结构的指针）。这样

函数 talloc 就可改写为：

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

必须强调的是 typedef 声明并没有创造任何意义上的新类型；它只是为现有的类型增加了一个新名字。它也没有创造任何新的语义：用这种方式声明的变量与由显式拼写的类型所声明的变量的特性完全一致。从效果上看，typedef 与 #define 类似，不同之处是由于它可以被编译器解释，因而可以应付那些超出预处理器能力的文本替换。例如，

```
typedef int (*PFI)(char *, char *);
```

创建了类型 PFI，它表示"指向返回 int 的（且拥有两个 char * 参数的）函数的指针"，其可被用于第 5 章的排序程序这样的上下文中，譬如

```
PFI strcmp, numcmp;
```

撇开纯粹的审美因素，采用 typedef 有两个主要的理由。第一个是将程序参数化以应对移植性问题。假如 typedef 用于那些可能与机器相关的数据类型，那么当程序被移植时只有 typedef 的定义需要改变。一种普遍情况是使用 typedef 名字代表不同的整数类型，这样可以为每类主机选择一个恰当的 short、int 和 long 的集合。像来自标准库中的 size_t 和 ptrdiff_t 类型就是实际的例子。

采用 typedef 的第二个目的是为程序提供更好的说明性——一个叫做 Treeptr 的类型会比一个只是声明为某复杂结构的指针类型要更容易理解。

6.8 联合（Union）

联合是一个变量，它可以（在不同的时间）包含不同类型和大小的对象，并由编译器来追踪大小和对齐要求。联合提供了一种在单个存储区域操控不同类型的数据的途径，且不需要在程序中嵌入任何与具体机器相关的信息。它相当于 Pascal 中的可变记录。

在编译器的符号表管理器中有可能会找到一个例子：假如一个常量可以是一个 int、或一个 float、或一个字符指针；特定常量的值必须存放在正确类型的变量中，然而假如一个值无论何种类型都占用相同大小的存储空间，那么仍然是最方便的表管理方式。这正是联合的目的——一个单独的变量可以合法地包含多种类型中的任意一个。联合的语法基于结构：

```
unison u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

变量 u 将会大到足以包含这三个类型中最大的一个；其实际的大小依赖于具体实现。u 可以被赋给这些类型中的任意一种，然后用于表达式，只要它的用法是一致的：从 u 获取的类型必须是最新存放的类型。跟踪最新存放在联合中的是哪种类型是编程人员的责任；假如联合以一种类型存放并以另一种类型获取，则其结果依赖于具体实现。

在语法上，联合的成员就像结构一样通过

```
联合名.成员
```

或

```
联合指针->成员
```

进行访问。如果变量 utype 被用于跟踪 u 的当前存放的类型，那么或许会看到如下这样的代码：

```
if (utype == INT)
   printf("%d\n", u.ival);
else if (utype == FLOAT)
   printf("%f\n", u.fval);
else if (utype == STRING)
   printf("%s\n", u.sval);
else
   printf("bad type %d in  utype\n", utype);
```

联合可以出现在结构和数组中，反之亦然。对结构中的联合（或者反过来）的成员的访问方式与嵌套结构的访问方式相同。例如，在如下定义的结构数组中

```
struct {
   char *name;
   int flags;
   int utype;
   union {
      int ival;
      float fval;
      char *sval;
   } u;
} symtab[NSYM];
```

对成员 ival 的引用通过

```
symtab[i].u.ival
```

完成，而对字符串 sval 的首字符的引用通过下面两者之一完成。

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

就效果而言，联合就是一个其所有成员相对于基本位置的偏移都是零的结构，该结构大到足以容纳其中"最宽"的成员，并且其对齐方式适合于联合中所有的类型。对结构的操作同样允许用于联合：可以按整个单元进行赋值或复制、获取地址、以及访问成员。

联合只可以使用其第一个成员类型的值进行初始化；即上面描述的联合 u 只能使用整数值进行初始化。

第 8 章中的存储空间分配器展示了联合是如何能够用来强制一个变量对齐到某个特定类型的存储空间边界上的。

6.9 位域（Bit-fields）

当存储空间极为宝贵时，可能有必要将几个对象压缩到单个机器字中；一种普遍的用法是类似编译器符号表等应用中的按位标记集合。一些外部强制数据格式（譬如对硬件设备的接口）也常常要求具有获取一个字的片段的能力。

想象某个编译器操作符号表时的片段。程序中的每个标识符都附带有一定信息，例如，它是否是一个关键字，是否是外部变量且（或）是静态变量，等等。将这些信息编码的最紧凑方式就是在单个 char 或 int 之内的一组按位标记。

做到这样的常用方法是定义一组与相关比特位对应的"掩码"，就如

```
#define KEYWORD     01
#define EXTERNAL    02
#define STATIC      04
```

或者

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

这些数字必须是 2 的幂次。这样对这些位的访问就变成了使用第 2 章所描述的移位、掩码、反码等运算符"按位操控"的问题。

某些惯用法出现得很频繁：

```
flags |= EXTERNAL | STATIC;
```

将 flags 中的 EXTERNAL 位与 STATIC 位打开，而

```
flags &= ~(EXTERNAL | STATIC);
```

将它们关闭，以及

```
if ((flags & (EXTERNAL | STATIC)) == 0) …
```

当这两个位都关闭时为真。

尽管这些惯用法很容易被掌握，但另一种选择是 C 语言提供的可直接定义和访问字内位域的能力，而无需使用位逻辑运算符。**位域**（bit-field），或简称为**域**（field），是位于一个我们称之为“字”的存储单元内的一组相邻的比特位，该存储单元字由具体实现定义。定义和访问域的语法基于结构。例如，上面 #define 的符号表可以被替换为三个域的定义：

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

此处定义了一个称为 flags 的变量，它包含三个 1 比特位的域。冒号之后的数代表了该域的比特位宽度。这些域被声明为 unsigned int 以确保它们是无符号的量。

单个域的引用方式与其他结构成员一样：flags.is_keyword、flags.is_extern 等等。域就像小的整数，并且就像其他整数一样可以参与算术表达式的运算。于是之前的例子就可以更自然地写成：

```
flags.is_extern = flags.is_static = 1;
```

用于打开这两个位；

```
flags.is_extern = flags.is_static = 0;
```

用于关闭这两个位；而

```
if (flags.is_extern == 0 && flags.is_static == 0)
    …
```

用于测试它们。

关于域的几乎所有的内容都依赖于具体实现。一个域是否可以跨字的边界由具体实现定义。域不一定要有名字；未命名域（只有冒号和宽度）用于空间填充。特殊宽度 0 可以被用于强制对齐到下一个字的边界。

域在一些机器中从左至右分配，而在另一些机器中从右至左分配。这意味着尽管域对于维护内部定义的数据结构很有用，但当获取外部定义的数据的片断时，哪一端先出现的问题需要仔细考虑；依赖于这些因素的程序是不可移植的。域可以仅用 int 进行声明；但为了移植性，请显式地指定 signed 或者 unsigned。它们不是数组，也没有地址，因此 & 运算符不能作用于它们。

第 7 章 输入与输出

输入/输出功能并不是 C 语言本身的组成部分，所以到目前为止，我们并没有过多地强调它们。然而，程序与所在环境之间的交互方式比之前展示的要复杂许多。本章将描述标准库——一组为 C 程序提供输入/输出、字符串处理、存储管理、数学运算以及其他一些功能服务的函数集合。讨论的重点将放在输入/输出上。

ANSI 标准精确地定义了这些库函数，所以在任何可使用 C 语言的系统中这些函数形式都相互兼容。如果程序与系统交互的部分仅仅使用了标准库提供的功能，那么这些程序不经修改就可以从一个系统移植到另一个系统。

这些库函数的特性分别声明在十几个头文件中，我们已经见过其中一些，如 <stdio.h>、<string.h> 和 <ctype.h>。我们不打算将整个标准库都罗列于此，因为我们更关心如何使用标准库编写 C 程序。附录 B 对标准库进行了详细的描述。

7.1 标准输入/输出

正如第 1 章所述，标准库实现了一个简单的文本输入/输出模型。文本流由一连串文本行组成，每行以一个换行符结尾。如果系统不是这种运作方式，标准库将负责使该系统看上去像是这样运作的。例如，标准库可能会在输入端将回车符和新行符转换为换行符，而在输出端进行逆向转换。

最简单的输入机制是使用 getchar 函数从标准输入（一般为键盘）中一次读取一个字符：

```
int getchar(void)
```

getchar 函数在每次被调用时返回下一个输入字符；若遇到文件末尾，则返回 EOF。符号常量 EOF 在 <stdio.h> 中定义，其值一般为-1，但应当使用 EOF 而不是-1 来测试文件是否结束，这样可使程序与 EOF 的特定值无关。

在许多环境中，可通过重定向符 < 实现输入重定向，以一个文件替代键盘输入：如果程序 prog 使用了 getchar，那么命令行

```
prog <infile
```

将使得 prog 从文件 infile（而不是从键盘）中读取字符。输入的变换是以一种 prog 本身毫不觉察的方式完成的，而且字符串"<infile"也不包含在 argv 的命令行参数中。如果输入通过管道机制来自另一程序，那么输入的变换同样不可见：例如，在某些系统中，命令行

```
otherprog | prog
```

将运行两个程序 otherprog 和 prog，并将 otherprog 的标准输出连接至 prog 的标准输入。

函数

```
int putchar(int)
```

用于输出数据：putchar(c)将字符 c 送至标准输出（缺省为屏幕显示）。正常情况下 putchar 返回所输出的字符；如果发生错误，则返回 EOF。同样，输出通常也能够用 >文件名 重定向到一个文件：例如，如果程序 prog 使用了 putchar，那么命令行

```
prog >outfile
```

会使得 prog 将写到标准输出的数据转而写入文件 outfile 中。如果系统支持管道，那么

```
prog | anotherprog
```

将把 prog 的标准输出连接至 anotherprog 的标准输入。

函数 printf 也向标准输出上输出数据。程序可能交替调用 putchar 和 printf，输出将按照调用的顺序依次产生。

每一个使用输入/输出库函数的源文件都必须在首次使用之前包含如下代码行：

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

当文件名用尖括号 < 和 > 括上时，预处理器将在某些标准位置查找这个头文件（例如，在 UNIX 系统中，典型的位置是在目录/usr/include 中）。

许多程序仅从一个输入流中读取数据并只向一个输出流写出数据。对于这类程序，使用函数 getchar、putchar 和 printf 来实现输入/输出可能已完全够用，用于起步当然也足够了，这当重定向被用于将某一程序的输出连接至另一程序的输入的情况时尤其如此。例如，考察下面的 lower 程序，它将输入转换为小写字母的形式：

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

main()    /* lower: 将输入转换为小写形式 */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

函数 tolower 在头文件 <ctype.h> 中定义；它将大写字母转换为小写形式，并将其余字符原样返回。正如之前提及，头文件 <stdio.h> 中的 getchar 和 putchar 以及 <ctype.h> 中的 tolower 等“函数”常常为宏，这样可避免为每个字符都进行函数调用的开销。我们将在 8.5 节介绍它们的实现方法。无论 <ctype.h> 中的函数在给定机器上如何实现，使用这些函数的程序都不必了解字符集的知识。

练习 7-1. 编写一个程序，根据程序调用名（放在 argv[0]中），实现将大写字母转换为小写字母或将小写字母转换为大写字母的功能。

7.2 格式化输出——printf 函数

输出函数 printf 用于将内部数值转换为字符的形式。在前面的章节中我们曾非正式地使用过这个函数。本节涵盖了它最典型的一些用法，但并不完全；关于 printf 的完整描述参见附录 B。

```
int printf(char *format, 参数1, 参数2, …)
```

函数 printf 通过输出格式 format 的控制，将其参数进行转换与格式化，并在标准输出上打印出来。它的返回值为所打印的字符数。

格式化字符串包含两种类型的对象：普通字符和转换规格说明。在输出时普通字符原样不动复制到输出流中，而转换规格说明并不直接输出到输出流中，而是用于控制 printf 中后续参数的转换和打印。每一个转换规格说明都由一个百分号 % 引入，并以一个转换字符结束。在字符%和转换字符中间可能依次包含如下成分：

- 负号，用于指定被转换的参数按照左对齐的形式输出。
- 数，用于指定最小字段宽度。被转换后的变元将在不小于最小字段宽的宽度中打印出来。如果必要，字段左面（或右面，如果使用左对齐）多余的字符位置用空格填充以保证最小字段宽度。
- 小数点，用于将字段宽和精度分开。
- 数，用于表示精度，即指定一个字符串中所要打印的最大字符数，或一个浮点数小数点后的位数，或一个整数最少输出的数字数目。
- 字母 h 或 l，h 表示将整数作为 short 类型打印，l 表示将整数作为 long 类型打印。

表 7-1 中列出了所有转换字符。如果 % 后面的字符不是一个转换规格说明，则该行为是未定义的。

表 7-1 printf 函数的基本转换字符

字符	变元类型；输出形式
d, i	int 类型；十进制数
o	unsigned int 类型；无符号八进制数（不以 0 开头）
x, X	unsigned int 类型；无符号十六进制数（不以 0x 或 0X 开头），10~15 这六个数分别用 a~f 或 A~F 表示
u	unsigned int 类型；无符号十进制数
c	int 类型；单个字符
s	char*类型；打印字符串的字符直至遇到空字符（'\0'）或已打印了由精度指定的字符数为止
f	double 类型；十进制小数表示法：[-]m.dddddd，其中 d 的个数由精度指定（缺省为 6）
e, E	double 类型；指数形式：[-]m.dddddd e±xx 或[-]m.dddddd E±xx，其中 d 的数目由精度指定（缺省为 6）
g, G	double 类型；如果指数小于-4 或大于等于精度值，则用%e 或%E 格式输出，否则用%f 格式输出。尾部的 0 和小数点不打印
p	void * 类型；指针（取决于具体实现）
%	不转换参数；打印一个百分号%

转换规格说明中，宽度或精度可用星号 * 表示，此时，宽度或精度的值通过转换下一个参数（必须为 int 类型）来计算。例如，为了从字符串 s 中最多打印 max 个字符，可用如下语句：

```
printf("%.*s", max, s);
```

格式转换的大部分内容已经在先前几章中有过阐述，但与字符串相关的精度部分我们还未介绍。下表展示了在打印字符串"hello, world"（12 个字符）时各种转换规格说明所产生的效果。我们在字段的前后加上冒号，这样可以清晰地显示出字段的宽度。

```
:%s:            :hello, world:
:%10s:          :hello, world:
:%.10s:         :hello, wor:
:%-10s:         :hello, world:
:%.15s:         :hello, world:
:%-15s:         :hello, world   :
:%15.10s:       :     hello, wor:
:%-15.10s:      :hello, wor     :
```

警告：函数 printf 用它的第一个参数来判断其后的参数数目以及它们的类型。如果参数不够或者参数类型错误，那么函数处理将会混乱，使用者也会得到错误的结果。请注意如下两

个调用之间的差别：

```
printf(s);          /* 如果字符串 s 含有字符%，输出将出错 */
printf("%s", s);    /* 安全做法 */
```

函数 sprintf 进行与函数 printf 同样的转换，但它将输出存放在一个字符串中：

```
int sprintf(char *string, char *format, 参数1, 参数2, …)
```

与之前一样，sprintf 函数根据 format 对参数序列进行格式化，只不过 sprintf 将结果放到 string 中而不是标准输出中。string 的大小必须足以存放输出结果。

练习 7-2. 编写一个程序，以合理的方式打印任意输入的内容。该程序至少应能根据用户习惯以八进制或十六进制打印非显示字符，并将过长的文本行进行拆分。

7.3 变长参数表

本节以实现 printf 函数的一个迷你版本为例，介绍如何以可移植的方式编写能处理可变长度的参数表的函数。由于重点在于参数的处理，因此函数 minprintf 只处理格式字符串和参数表，格式转换则通过调用 printf 实现。

函数 printf 的正确说明形式如下：

```
int printf(char *fmt, …)
```

其中，声明 … 表示对应参数的数量和类型可能会变化。声明 … 只能在参数表的最后位置出现。由于不必让 minprintf 像函数 printf 那样返回实际输出的字符数，我们将它声明为如下形式：

```
void minprintf(char *fmt, …)
```

关键技巧在于 minprintf 如何遍历一个甚至连名字也没有的参数表。标准头文件 <stdarg.h> 中包含了一组宏定义，它们对如何依次处理参数表进行了定义。此头文件的实现因机器不同而变化，但其提供的接口是一致的。

类型 va_list 用于声明一个依次引用每个参数的变量。在函数 minprintf 中，该变量被称为 ap，意思是"argument pointer（参数指针）"。宏 va_start 将 ap 初始化，使其指向第一个无名参数。在 ap 被使用之前，这个宏必须被调用一次。参数表中必须至少包括一个有名参数；va_start 用最后一个有名参数来确定起始位置。

每调用一次 va_arg，它就返回一个参数，并将 ap 指向下一个参数。va_arg 使用一个类型名以确定返回何种类型的对象以及 ap 移动的步长。最后，必须在函数返回之前调用 va_end，完成必要的清理工作。

上述特性构成了实现我们的简化 printf 的基础：

```
#include <stdarg.h>

/* minprintf: 带有可变参数表的迷你 printf 函数 */
void minprintf (char *fmt, …)
{
    va_list ap;  /* 依次指向每一个无名参数 */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt);  /* 使 ap 指向第一个无名参数 */
```

```
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap);  /* 完成之后的清理工作 */
}
```

练习 7-3. 改写 minprintf 函数，使其能处理 printf 函数的更多功能。

7.4 格式化输入——scanf 函数

scanf 是与输出函数 printf 相对应的输入函数，它从反方向提供了许多与 printf 同样的转换功能，其函数原型如下：

```
int scanf(char *format, …)
```

scanf 从标准输入中读取字符序列，根据 format 中的转换规格说明对其进行解析，并将结果存放在其余的参数中。格式参数 format 将在下面讨论；其余每个参数都必须是指针，用于指明对应的转换输入应该存放的位置。与 printf 一样，本节只总结了 scanf 的一些最有用的特性，并未列出全部内容。

当scanf的格式串全匹配光或者当一些输入无法与控制说明相匹配时，它就停止运行，并返回成功匹配并赋值的输入项的个数。返回值可作为判断已获得输入值的输入项数的依据。如果是文件结尾，则返回EOF；注意这与返回 0 值不同，0 表示下一个输入字符和格式串中的第一个说明不匹配。下次对此函数的调用将从上次转换[①]的最后一个字符的下一字符开始继续搜索。

还存在一个输入函数 sscanf，其用于从一个字符串（而不是标准输入）中读取字符序列：

```
int sscanf(char *string, char *format, 参数1, 参数2, …)
```

它按照 format 中的格式从 string 中读取字符序列，并通过参数 1、参数 2 等等存放转换结果。这些参数必须是指针。

输入格式串一般都包含转换规格说明，用以控制输入的转换。格式化串可能包含如下成分：

[①] 此处的“转换”可理解为“匹配成功”，译者注。

- 空格或制表符，在处理过程中将被忽略。
- 普通字符序列（不包括%），用于匹配输入流中下面尚待读入的非空白字符序列。
- 转换规格说明，由一个%、一个可选的*（赋值屏蔽字符）、一个可选的整数（用于指定字段最大宽度）、一个可选的 h、l 或 L（用于指明转换对象的宽度）以及一个转换字符组成。

每一转换规格说明用于依次指导每一输入字段的转换。通常转换结果会放到相应参数所指向的变量中。但是，如果转换规格说明中包含表示赋值屏蔽的字符 * ，则会跳过该输入字段，不进行赋值。输入字段被定义为一个由非空白符组成的字符串，字段范围或者直至下一个空白符，或者直至字段所指定的最大宽度。这意味着 scanf 会越过行边界查找它的输入，因为换行符也是空白符（空白符包括空格、横向制表符、换行符、回车符、纵向制表符以及换页符）。

转换字符用于指明如何解释输入字段。按照 C 语言的按值调用的语义，对应的参数必须是一个指针。转换字符如表 7-2 中所示：

表 7-2. scanf 函数的基本的转换字符

字符	输入数据；参数类型
d	十进制整数；int *类型
i	整数；int *类型。可以是八进制（以 0 开头）或十六进制整数（以 0x 或 0X 开头）
o	无符号八进制整数（以 0 开头或不以 0 开头）；unsigned int *类型
u	无符号十进制整数；unsigned int *类型
x, X	无符号十六进制整数（以或不以 0x 或 0X 开头）；unsigned int *类型
c	字符；char *类型。将下（几）个输入字符（缺省字符数为 1）存放到指定的位置。该转换规格说明不跳过空白符，若要读取下一个非空白字符，使用%1s
s	字符串（不加引号）；char *类型，指向一个字符数组，其足以存放该字符串以及在该串末尾加上的一个空字符（'\0'）
e, f, g	可带前缀正负号、小数点及指数部分的十进制浮点数；float *类型
%	字符%本身；不进行任何赋值操作

在转换字符 d、i、o、u 及 x 之前可以加上字符 h 或 l。h 用于表明参数列表中的对应项是一个指向 short 而不是 int 的指针，l 则表明对应项为指向 long 的指针。类似地，转换字符 e、f 及 g 之前可以加上 l，表明参数列表中的对应项是一个指向 double 而不是 float 的指针。

作为第一个例子，下面用函数 scanf 进行输入转换来改写第 4 章的原始计算器程序：

```
#include <stdio.h>

main()  /* 原始计算器程序 */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

假设我们希望读取包含如下日期形式的输入行：

```
25 Dec 1988
```

则相应的 scanf 语句为：

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

monthname 没有加取址运算符&，因为数组名本身就是指针。

普通字符（串）也可出现在 scanf 的格式串中，但其必须与输入中同样的字符（串）相匹配。因此可用下列 scanf 语句读取形如 mm/dd/yy 的日期数据：

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
```

scanf 忽略格式字符串中的空格和制表符；此外，在查找输入值时，它也会略过输入中的空白符（空格、制表符、换行符等等）。为了读取没有固定格式的输入，通常最好是每次读入一行，然后用 sscanf 逐个读入。例如，假设我们希望读取可能包含上述任一形式的日期的行时，可以编写如下代码：

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line);  /* 形如 25 Dec 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line);  /* 形如 mm/dd/yy */
    else
        printf("invalid: %s\n", line);  /* 无效日期形式 */
}
```

scanf 可与其他输入函数混合调用。scanf 调用之后下一输入函数的调用将从该 scanf 未读取的首个字符处开始读取数据。

最后提醒读者：scanf 和 sscanf 函数的参数都必须是指针。最常见的错误是将输入语句写成

```
scanf("%d", n);
```

其正确形式应为

```
scanf("%d", &n);
```

这类错误在编译阶段通常检测不到。

练习 7-4．参照上一节的 minprintf 函数，编写一个自己的 scanf 版本。

练习 7-5．改写第 4 章的后缀计算器，用 scanf 和（或）sscanf 完成输入及数字转换。

7.5 文件访问

到目前为止，我们所讨论的例子都是从标准输入读取数据并向标准输出写入数据。标准输入和标准输出是由程序所在的操作系统为程序自动定义的。

下面这一步要编写一个程序，对尚未与之关联的文件进行访问。cat 是一个可说明这种操作需求的程序，它把一组有名文件的内容串联起来并输出到标准输出。cat 可用于将文件打印到屏幕上，也可作为那些不能通过名字访问文件的程序的通用输入收集器。例如，命令

```
cat x.c y.c
```

将文件 x.c 和 y.c 的内容（而不包含任何其他内容）打印到标准输出上。

问题在于怎样使得这些有名文件能够被读取——即怎样将用户所用的外部文件名字和读取这些文件的语句联系起来。

方法很简单：一个文件在可被读写之前，必须用库函数 fopen 打开。fopen 用诸如 x.c 或 y.c 这样的外部名字与操作系统进行某些必要的处理和协商（我们不必关心其细节），并返回一个之后可用于文件读写的指针。

该指针称为**文件指针**，它指向一个包含文件信息的结构，例如：缓冲区的位置、缓冲区中当前所指向的字符位置、文件是正在读或还是正在写、文件是否出错或是否已经达到文件末尾等等。用户不必知道这些细节，因为 <stdio.h> 的定义中包括了一个称为 FILE 的结构声明，使用文件指针只需声明即可，例如：

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

上述语句表示 fp 是一个指向 FILE 的指针，fopen 返回一个指向 FILE 的指针。注意 FILE 像 int 一样是类型名，而不是结构标记，它由 typedef 定义。（fopen 在 UNIX 系统中的实现细节将在 8.5 节讨论。）

程序可以这样调用 fopen 函数：

```
fp = fopen(name, mode);
```

fopen 的第一个参数是一个字符串，包含待打开文件的名字。第二个参数是**访问模式**，它也是一个字符串，表示调用者希望如何使用文件。允许的访问模式包括读（"r"）、写（"w"）以及追加（"a"）。一些系统还区分文本和二进制文件，对后者的访问必须在模式串中添加字符"b"。

如果以写或追加方式打开一个不存在的文件，那么如有可能，系统将创建该文件。当以写方式打开一个已存在的文件时，该文件的原有内容将被丢弃，而以追加方式打开时文件原有内容将被保留。试图读一个不存在的文件会导致错误，也有其他原因可能导致错误，比如试图读一个无读取权限的文件。如果有错误发生，fopen 将返回 NULL。（错误可以更精确地进行识别；参见附录 B 第 1 节末尾关于错误处理函数的讨论。）

一旦文件被打开，接下来就需要某种途径来读写文件了。有几种可能的方法，getc 和 putc 是其中最简单的。getc 返回文件中的下一个字符，它需要文件指针来告知是哪个文件。

```
int getc(FILE *fp)
```

getc 返回 fp 所指向的输入流中的下一个字符；如果到达文件末尾或发生错误，则返回 EOF。

putc 是一个输出函数：

```
int putc(int c, FILE *fp)
```

putc 将字符 c 写入 fp 指向的文件并返回所写入的字符；如果有错误出现，则返回 EOF。正如 getchar 和 putchar 一样，getc 和 putc 可以是宏而不是函数。

当一个 C 程序开始运行时，操作系统环境负责打开 3 个文件并提供相应的文件指针。这些文件是标准输入、标准输出和标准错误，对应的文件指针分别为 stdin、stdout 和 stderr，它们在 <stdio.h> 中声明。通常情况下，stdin 指向键盘，而 stdout 和 stderr 则指向显示器；但正如 7.1 节中所述，stdin 和 stdout 可以被重定向到文件或管道。

getchar 和 putchar 可以用 getc、putc、stdin 及 stdout 定义如下：

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

为了格式化文件的输入输出，可以使用函数 fscanf 和 fprintf。它们与 scanf 和 printf 的区别仅在于它们的第一个参数是指向待读写文件的指针，格式串则是第二个参数。

```
int fscanf(FILE *fp, char *format, …)
int fprintf(FILE *fp, char *format, …)
```

掌握上述预备知识之后，我们现在就能编写将一组文件串联起来的 cat 程序了。该程序的设计与其他许多程序类似：若命令行中带有参数，则将它们作为文件名依次处理，若没有参数，则使用标准输入。

```
#include <stdio.h>

/* cat: 将一组文件串联起来，版本 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc == 1)  /* 如果未带参数，则复制标准输入 */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: can't open %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy: 将文件 ifp 复制到 ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

文件指针 stdin 与 stdout 是 FILE *类型的对象。但它们是常量而不是变量，因而不可能对其进行赋值。

函数

```
int fclose(FILE *fp)
```

的功能与 fopen 相反，它断开由 fopen 建立的文件指针和外部名字之间的连接，释放文件指针以供其他文件使用。由于大多数操作系统对于一个程序可同时打开的文件数都存在一定限制，所以当文件指针不再需要时释放它是一个好的思路，就像我们在 cat 程序中所做的那样。对输出文件使用 fclose 还有另一原因：它将缓冲区中 putc 正在收集的输出数据刷到文件中。当一个程序正常终止时，系统自动对该程序打开的每个文件调用 fclose 函数。（如果不再需要使用 stdin 与 stdout，可以将其关闭。它们也能通过库函数 freopen 重新指定。）

7.6 错误处理——stderr 和 exit 函数

cat 程序对错误的处理并不理想。假如由于某些原因而造成某个文件无法访问，那么相应的诊断信息将会在连接输出的末尾处打印出来。如果程序输出至屏幕，这种处理方式尚可接受，

但当输出至一个文件或通过管道向另一程序输入时则会出现问题。

为了更好地处理这类情况，程序配备了与 stdin 和 stdout 类似、被称为 stderr 的第二种输出流。即便重定向了标准输出，写入 stderr 的输出通常仍会显示到屏幕上。

下面我们改写 cat 程序，使其将错误信息写到标准错误 stderr 上。

```
#include <stdio.h>

/* cat: 将多个文件拼接到一起，版本 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0];  /* 程序名，供错误处理使用 */

    if (argc == 1)  /* 如果没有参数，则从标准输入拷贝 */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                        prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: error writing stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

此程序通过两种途径报错。首先，将 fprintf 生成的诊断信息输出到 stderr，于是诊断信息就会显示到屏幕而不是消失在管道或进入某个输出文件中。诊断信息包含了 argv[0]中的程序名，这样当此程序和其他程序一起使用时就可以辨别出错误的来源。

其次，此程序使用了标准库函数 exit，当它被调用时将终止程序执行。任何调用上述程序的进程都能获得该程序调用 exit 时的参数，因此可通过另一个将该程序作为子进程的程序来测试该程序是否成功执行。按照惯例，返回 0 值表示一切正常，而非 0 值通常表示出现了异常情况。exit 为每个已打开的输出文件调用 fclose 函数，将所有缓冲的输出刷入相应的文件中。

在主程序 main 中，语句 return expr 等价于 exit(expr)。但是 exit 有其优点：它可被其他函数调用，并可用类似第 5 章中介绍的模式查找函数来找到这些调用。

如果流 fp 中发生了错误，函数 ferror 返回一个非 0 值。

```
int ferror(FILE *fp)
```

尽管输出错误很少遇到，但它们确实会发生（例如，当磁盘满时），因此一个产品级程序应当检测这样的错误。

函数 feof(FILE *)与 ferror 类似，如果遇到文件结尾，它返回一个非 0 值。

```
int feof(FILE *fp)
```

本节编写的小程序旨在说明问题，因此我们不太关心它的退出状态，但对于任何正式的程序而言，都应特别注意返回合理有用的状态值。

7.7 行输入与行输出

标准库提供了一个输入函数 fgets，它和之前几章中用到的函数 getline 类似：

```
char *fgets(char *line, int maxline, FILE *fp)
```

fgets 从 fp 所指向的文件中读取下一行输入（包括换行符），并将其存放到字符数组 line 中；最多可读取 maxline-1 个字符，读取的行以'\0'结尾。通常情况下，fgets 返回 line，但如果遇到了文件末尾或发生了错误，则返回 NULL（我们编写的 getline 函数返回行的长度，这个值更有用，当它为 0 时意味着到达了文件末尾）。

对于输出，函数 fputs 将一个字符串（不一定包含换行符）写入一个文件中：

```
int fputs(char *line, FILE *fp)
```

如果发生错误，该函数返回 EOF，否则返回一个非负值。

库函数 gets 和 puts 的功能与 fgets 和 fputs 类似，但它们是对 stdin 和 stdout 进行操作。有一点注意不要混淆，gets 在读入字符串时将丢弃末尾的'\n'，而 puts 则要在写出字符串的末尾添加'\n'。

类似 fgets 和 fputs 这样的函数并不特别，下列代码就拷贝自作者所用系统中标准库内 fgets 和 fputs 的代码：

```
/* fgets: 从 iop 所指向的文件中最多读取 n 个字符 */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: 将字符串 s 输出到 iop 所指向的文件 */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

ANSI C 标准规定：ferror 函数在遇到错误时返回非 0 值；而 fputs 函数在遇到错误时返回 EOF，否则返回一个非负整数值。

使用 fgets 很容易实现我们的 getline 函数：

```
/* getline: 读入一行并返回其长度 */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
```

```
        return strlen(line);
}
```

练习 7-6．编写一个程序，比较两个文件并打印它们第一个不相同的行。

练习 7-7．修改第 5 章的模式查找程序，使其从一组有名文件中读取输入，并在没有文件作为参数时从标准输入中读取输入。当找到一个匹配行时，是否应该将对应的文件名打印出来？

练习 7-8．编写一个程序，打印一组文件；每个文件从新的一页开始打印，包括文件标题和每页的页数。

7.8 其他函数

标准库提供了功能宽泛的函数集合。本节将对其中最常用的函数进行一个简要概述。更多细节和许多这里没有介绍的函数请参见附录 B。

7.8.1 字符串处理函数

前面已经提到过字符串函数 strlen、strcpy、strcat 和 strcmp，它们都定义在头文件 <string.h> 中。在下列函数中，s 与 t 的类型为 char *，而 c 与 n 的类型为 int。

strcat(s,t)	将字符串 t 连接到字符串 s 的末尾
strncat(s,t,n)	将字符串 t 中前 n 个字符连接到字符串 s 的末尾
strcmp(s,t)	视字符串 s 小于、等于或大于字符串 t 而返回负整数、0 或正整数
strncmp(s,t,n)	与 strcmp 相同，但只处理前 n 个字符
strcpy(s,t)	将字符串 t 复制到字符串 s
strncpy(s,t,n)	将字符串 t 中最多前 n 个字符复制到字符串 s
strlen(s)	返回字符串 s 的长度
strchr(s,c)	返回字符串 s 中首次出现 c 的位置指针，若未找到则返回 NULL
strrchr(s,c)	返回字符串 s 中末次出现 c 的位置指针，若未找到则返回 NULL

7.8.2 字符类测试和转换函数

头文件 <ctype.h> 中的一些函数用于字符的测试和转换。在下列函数中，c 是可用于表示 unsigned char（无符号字符）或 EOF 的一个 int 型数。函数的返回值类型为 int。

isalpha(c)	若 c 是字母，则返回非 0 值，否则返回 0
isupper(c)	若 c 是大写字母，则返回非 0 值，否则返回 0
islower(c)	若 c 是小写字母，则返回非 0 值，否则返回 0
isdigit(c)	若 c 是数字，则返回非 0 值，否则返回 0
isspace(c)	若 c 是空格、横向制表符、换行符、回车符和纵向制表符，则返回非 0 值，否则返回 0
toupper(c)	返回 c 的大写形式
tolower(c)	返回 c 的小写形式

7.8.3 ungetc 函数

标准库提供了一个称为 ungetc 的函数，它是第 4 章中编写的 ungetch 函数的一个功能上限制更严的版本。

```
int ungetc(int c, FILE *fp)
```

该函数将字符 c 写回文件 fp。如果执行成功则返回 c，否则返回 EOF。每个文件只能接收一个写回字符。ungetc 函数可以和诸如 scanf、getc 或 getchar 等任何输入函数配合使用。

7.8.4 命令执行函数

函数 system(char *s)执行包含在字符串 s 中的命令，然后继续执行当前程序。s 的内容很大程度上取决于所用的操作系统。例如，在 UNIX 系统中，语句

```
system("date");
```

将执行程序 date，它在标准输出上打印当天的日期和时间。system 函数返回一个来自被执行命令的与系统相关的整数状态。在 UNIX 系统中，该返回状态是 exit 所返回的数值。

7.8.5 存储管理函数

函数 malloc 和 calloc 用于动态分配内存块。语句

```
void *malloc(size_t n)
```

在分配成功时，返回一个指向 n 字节大小的未初始化的存储空间的指针，否则返回 NULL。而语句

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

在分配成功时，返回一个指向足以容纳由 n 个指定大小的对象组成的数组的存储空间的指针，否则返回 NULL。该存储空间被初始化为全 0。

由 malloc 或 calloc 函数返回的指针能自动地根据所分配对象的类型进行适当的对齐调整，但它必须强制转换为恰当的类型，例如：

```
int *ip;

ip = (int *)calloc(n, sizeof(int));
```

free(p)函数用于释放 p 所指向的存储空间，其中 p 是此前通过调用 malloc 或 calloc 函数而得到的指针。存储空间的释放顺序没有什么限制，但释放一个不是通过调用 malloc 或 calloc 函数而得到的指针所指向的存储空间是一个可怕的错误。

使用已经释放的存储空间同样是错误的。下面所示的是一个典型的不正确的代码段，它是用于释放链表中每项所占存储空间的循环：

```
for (p = head; p != NULL; p = p->next)  /* 错了! */
    free(p);
```

正确的处理方法是在释放之前将所有必要的信息先保存起来：

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
```

```
}
```

8.7 节给出了一个类似于 malloc 函数的存储分配程序的实现，该程序分配的存储块可以以任意顺序释放。

7.8.6 数学函数

头文件 <math.h> 中声明了二十多个数学函数，下面所列的是一些经常使用的函数。每个函数带有一个或两个 double 型参数，并且返回一个 double 类型的值。

sin(x)	x 的正弦函数，其中 x 用弧度表示
cos(x)	x 的余弦函数，其中 x 用弧度表示
atan2(y,x)	y/x 的反正切函数，该函数值用弧度表示
exp(x)	指数函数 e^x
log(x)	x 的自然对数（以 e 为底）函数（x > 0）
log10(x)	x 的常用对数（以 10 为底）函数（x > 0）
pow(x, y)	x^y
sqrt(x)	x 的平方根函数（x ⩾ 0）
fabs(x)	x 的绝对值函数

7.8.7 随机数发生器函数

函数 rand()用于生成介于 0 和 RAND_MAX 之间的伪随机整数序列，其中 RAND_MAX 在头文件 <stdlib.h> 中定义。下面是一种生成大于等于 0 但小于 1 的随机浮点数的方法：

```
#define frand() ((double)rand() / (RAND_MAX+1.0))
```

（如果所用函数库中已提供了一个用于生成浮点随机数的函数，那么它可能比上面这个函数具有更好的统计特性。）

函数 srand(unsigned)设置 rand 所用的随机数生成算法的种子值。2.7 节中给出了遵循标准的 rand 和 srand 函数的可移植的实现。

练习 7-9. 诸如 isupper 这样的函数能够以节省空间或者节省时间的方式实现。试探究这两种可能性。

第8章 UNIX 系统接口

UNIX 操作系统通过一系列的系统调用提供服务，这些系统调用实际上是操作系统内的函数，它们可以被用户程序调用。本章将介绍如何在 C 语言程序中使用一些重要的系统调用。如果读者使用的是 UNIX，本章将会对你有直接的帮助，这是因为，我们经常需要借助于系统调用以获得最高的效率，或者访问标准库中没有的某些功能。但是，即使读者是在其它操作系统上使用 C 语言，本章的例子也将会帮助你对 C 语言程序设计有更深入的了解。不同系统中的代码具有相似性，只是一些细节上有区别而已。因为 ANSI C 标准函数库是以 UNIX 系统为基础建立起来的，所以，学习本章中的程序还将有助于更好地理解标准库。

本章的内容包括 3 个主要部分，输入 / 输出、文件系统和存储分配。其中，前两部分的内容要求读者对 UNIX 系统的外部特性有一定的了解。

第 7 章介绍的输入 / 输出接口对任何操作系统都是一样的。在任何特定的系统中，标准库函数的实现必须通过宿主系统提供的功能来实现。接下来的几节将介绍 UNIX 系统中用于输入和输出的系统调用，并介绍如何通过它们实现标准库。

8.1. 文件描述符

在 UNIX 操作系统中，所有的外围设备（包括键盘和显示器）都被看作是文件系统中的文件，因此，所有的输入 / 输出都要通过读文件或写文件完成。也就是说，通过一个单一的接口就可以处理外围设备和程序之间的所有通信。

通常情况下，在读或写文件之前，必须先将这个意图通知系统，该过程称为打开文件。如果是写一个文件，则可能需要先创建该文件，也可能需要丢弃该文件中原先已存在的内容。系统检查你的权力（该文件是否存在？是否有访问它的权限？），如果一切正常，操作采统将向程序返回一个小的非负整数，该整数称为文件描述符。任何时候对文件的输入 / 输出都是通过文件描述符标识文件，而不是通过文件名标识文件。（文件描述符类似于标准库中的文件指针或 MS-DOS 中的文件句柄。）系统负责维护已打开文件的所有信息，用户程序只能通过文件描述符引用文件，

因为大多数的输入 / 输出是通过键盘和显示器来实现的，为了方便起见，UNIX 对此做了特别的安排。当命令解释程序（即“ shell”）运行一个程序的时候，它将打开 3 个文件，对应的文件描述符分别为 0，1，2，依次表示标准输入，标准输出和标准错误。如果程序从文件 0 中读，对 1 和 2 进行写，就可以进行输 / 输出而不必关心打开文件的问题。

程序的使用者可通过<和>重定向程序的 I/O：

```
prog < 输入文件名 > 输出文件名
```

这种情况下，shell 把文件描述符 0 和 1 的默认赋值改变为指定的文件。通常，文件描述符 2 仍与显示器相关联，这样，出错信息会输出到显示器上。与管道相关的输入 / 输出也有类似的特性。在任何情况下，文件赋值的改变都不是由程序完成的，而是由 shell 完成的。只要程序使用文件 0 作为输入，文件 1 和 2 作为输出，它就不会知道程序的输入从哪里来，并输出

到哪里去。

8.2. 低级 I/O——read 和 write

输入与输出是通过 read 和 write 系统调用实现的。在 C 语言程序中，可以通过函数 read 和 write 访问这两个系统调用。这两个函数中，第一个参数是文件描述符，第二个参数是程序中存放读或写的数据的字符数组，第三个参数是要传输的字节数。

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

每个调用返回实际传输的字节数。在读文件时，函数的返回值可能会小于请求的字节数。如果返回值为 0，则表示已到达文件的结尾；如果返回值为-1，则表示发生了某种错误。在写文件时，返回值是实际写入的字节数。如果返回值与请求写入的字节数不相等，则说明发生了错误。

在一次调用中，读出或写入的数据的字节数可以为任意大小。最常用的值为 1，即每次读出或写入 1 个字符（无缓冲），或是类似于 1024～4096 这样的与外围设备的物理块大小相应的值。用更大的值调用该函数可以获得更高的效率，因为系统调用的次数减少了。

结合以上的讨论，我们可以编写一个简单的程序，将输入复制到输出，这与第 1 章中的复制程序在功能上相同。程序可以将任意输入复制到任意输出，因为输入 / 输出可以重定向到任何文件或设备。

```
#include "syscalls.h"

main()  /* copy input to output */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

我们已经将系统调用的函数原型集中放在一个头文件 syscalls.h 中，因此，本章中的程序都将包含该头文件。不过，该文件的名字不是标准的。

参数 BUFSIZ 也已经在 syscalls.h 头文件中定义。对于所使用的操作系统来说，该值是一个较合适的数值。如果文件大小不是 BUFSIZ 的倍数，则对 read 的某次调用会返回一个较小的字节数，write 再按这个字节数写，此后再调用 read 将返回 0。

为了更好地掌握有关概念，下面来说明如何用 read 和 write 构造类似于 getchar、putchar 等的高级函数。例如，以下是 getchar 函数的一个版本，它通过每次从标准输入读入一个字符来实现无缓冲输入。

```
#include "syscalls.h"

/* getchar:  unbuffered single character input */
int getchar(void)
{
```

```
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

其中，c 必须是一个 char 类型的变量，因为 read 函数需要一个字符指针类型的参数（&c）。在返回语句中将 c 转换为 unsigned char 类型可以消除符号扩展问题。

getchar 的第二个版本一次读入一组字符，但每次只输出一个字符。

```
#include "syscalls.h"

/* getchar:  simple buffered version */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) {  /* buffer is empty */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

如果要在包含头文件<stdio.h>的情况下编译这些版本的 getchar 函数，就有必要用 #undef 预处理指令取消名字 getchar 的宏定义，因为在头文件中，getchar 是以宏方式实现的。

8.3. open、creat、close 和 unlink

除了默认的标准输入、标准输出和标准错误文件外，其它文件都必须在读或写之前显式地打开。系统调用 open 和 creat 用于实现该功能。

open 与第 7 章讨论的 fopen 相似，不同的是，前者返回一个文件描述符，它仅仅只是一个 int 类型的数值。而后者返回一个文件指针。如果发生错误，open 将返回-1。

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

与 fopen 一样，参数 name 是一个包含文件名的字符串。第二个参数 flags 是一个 int 类型的值，它说明以何种方式打开文件，主要的几个值如下所示：

```
O_RDONLY    以只读方式打开文件
O_WRONLY    以只写方式打开文件
O_RDWR      以读写方式打开文件
```

在 System V UNIX 系统中，这些常量在头文件<fcntl.h>中定义，而在 Berkeley（BSD）版本中则在<sys/file.h>中定义。

可以使用下列语句打开一个文件以执行读操作：

```
fd = open(name, O_RDONLY,0);
```

在本章的讨论中，open 的参数 perms 的值始终为 0。

如果用 open 打开一个不存在的文件，则将导致错误。可以使用 creat 系统调用创建新文件或覆盖已有的旧文件，如下所示：

```
int creat(char *name, int perms);
fd = creat(name, perms);
```

如果 creat 成功地创建了文件，它将返回一个文件描述符，否则返回-1。如果此文件已存在，creat 将把该文件的长度截断为 0，从而丢弃原先已有的内容。使用 creat 创建一个已存在的文件不会导致错误。

如果要创建的文件不存在，则 creat 用参数 perms 指定的权限创建文件。在 UNIX 文件系统中，每个文件对应一个 9 比特的权限信息，它们分别控制文件的所有者、所有者组和其他成员对文件的读、写和执行访问。因此，通过一个 3 位的八进制数就可方便地说明不同的权限，例如，0755 说明文件的所有者可以对它进行读、写和执行操作，而所有者组和其他成员只能进行读和执行操作。

下面通过一个简化的 UNIX 程序 cp 说明 creat 的用法。该程序将一个文件复制到另一个文件。我们编写的这个版本仅仅只能复制一个文件，不允许用目录作为第二个参数，并且，目标文件的权限不是通过复制获得的，而是重新定义的。

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666    /* RW for owner, group, others */

void error(char *,  ...);

/* cp:  copy f1 to f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Usage: cp from to");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: can't create %s, mode %03o",
            argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error on file %s", argv[2]);
    return 0;
}
```

该程序创建的输出文件具有固定的权限 0666。利用 8.6 节中将要讨论的 stat 系统调用，可以获得一个已存在文件的模式，并将此模式赋值给它的副本。

注意，函数 error 类似于函数 printf，在调用时可带变长参数表。下面通过 error 函

数的实现说明如何使用 printf 函数家族的另一个成员 vprintf。标准库函数 vprintf 函数与 printf 函数类似，所不同的是，它用一个参数取代了变长参数表，且此参数通过调用 va_start 宏进行初始化。同样，vfprintf 和 vsprintf 函数分别与 fprintf 和 sprintf 函数类似。

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

/* error:  print an error message and die */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

一个程序同时打开的文件数是有限制的（通常为 20）。相应地，如果一个程序需要同时处理许多文件，那么它必须重用文件描述符。函数 close（int fd）用来断开文件描述符和已打开文件之间的连接，并释放此文件描述符，以供其它文件使用。close 函数与标准库中的 fclose 函数相对应，但它不需要清洗（flush）缓冲区。如果程序通过 exit 函数退出或从主程序中返回，所有打开的文件将被关闭。

函数 unlink(char *name)将文件 name 从文件系统中删除，它对应于标准库函数 remove。

练习 8-1　用 read、write、open 和 close 系统调用代替标准库中功能等价的函数，重写第 7 章的 cat 程序，并通过实验比较两个版本的相对执行速度。

8.4. 随机访问——lseek

输入 / 输出通常是顺序进行的：每次调用 read 和 write 进行读写的位置紧跟在前一次操作的位置之后。但是，有时候需要以任意顺序访问文件，系统调用 lseek 可以在文件中任意移动位置而不实际读写任何数据：

```
long lseek(int fd, long offset, int origin);
```

将文件描述符为 fd 的文件的当前位置设置为 offset，其中，offset 是相对于 orgin 指定的位置而言的。随后进行的读写操作将从此位置开始，origin 的值可以为 0、1 或 2，分别用于指定 offset 从文件开始、从当前位置或从文件结束处开始算起。例如，为了向一个文件的尾部添加内容（在 UNIX shell 程序中使用重定向符>>或在系统调用 fopen 中使用参数“a”），则在写操作之前必须使用下列系统调用找到文件的末尾：

```
lseek(fd, 0L, 2);
```

若要返回文件的开始处（即反绕），则可以使用下列调用：

```
lseek(fd, 0L, 0);
```

请注意，参数 0L 也可写为(long)0，或仅仅写为 0，但是系统调用 lseek 的声明必须保持一致。

使用 lseek 系统调用时，可以将文件视为一个大数组，其代价是访问速度会慢一些。例如，下面的函数将从文件的任意位置读入任意数目的字节，它返回读入的字节数，若发生错误，则返回-1。

```
#include "syscalls.h"

/*get:  read n bytes from position pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0) /* get to pos */
        return read(fd, buf, n);
    else
        return -1;
}
```

lseek 系统调用返回一个 long 类型的值，此值表示文件的新位置，若发生错误，则返回-1。标准库函数 fseek 与系统调用 lseek 类似，所不同的是，前者的第一个参数是 FILE *类型，且在发生错误时返回一个非 0 值。

8.5. 实例——fopen 和 getc 函数的实现

下面以标准库函数 fopen 和 getc 的一种实现方法为例来说明如何将这些系统调用结合起来使用。

我们回忆一下，标准库中的文件不是通过文件描述符描述的，而是使用文件指针描述的。文件指针是一个指向包含文件各种信息的结构的指针，该结构包含下列内容：一个指向缓冲区的指针，通过它可以一次读入文件的一大块内容；一个记录缓冲区中剩余的字符数的计数器；一个指向缓冲区中下一个字符的指针；文件描述符；描述读 / 写模式的标志；描述错误状态的标志等。

描述文件的数据结构包含在头文件<stdio.h>中，任何需要使用标准输入 / 输出库中函数的程序都必须在源文件中包含这个头文件（通过#include 指令包含头文件）。此文件也被库中的其它函数包含。在下面这段典型的<stdio.h>代码段中，只供标准库中其它函数所使用的名字以下划线开始，因此一般不会与用户程序中的名字冲突。所有的标准库函数都遵循该约定。

```
#define NULL     0
#define EOF      (-1)
#define BUFSIZ   1024
#define OPEN_MAX  20    /* max #files open at once */

typedef struct _iobuf {
    int  cnt;      /* characters left */
    char *ptr;     /* next character position */
    char *base;    /* location of buffer */
    int  flag;     /* mode of file access */
    int  fd;       /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];
```

```
#define stdin  (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])

enum _flags {
    _READ   = 01,   /* file open for reading */
    _WRITE  = 02,   /* file open for writing */
    _UNBUF  = 04,   /* file is unbuffered */
    _EOF    = 010,  /* EOF has occurred on this file */
    _ERR    = 020   /* error occurred on this file */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)    ((p)->flag & _EOF) != 0)
#define ferror(p)   ((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)

#define getc(p)   (--(p)->cnt >= 0 \
             ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 \
             ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()   getc(stdin)
#define putcher(x)  putc((x), stdout)
```

宏 getc 一般先将计数器减 1，将指针移到下一个位置，然后返回字符。（前面讲过，一个长的#define 语句可用反斜杠分成几行。）但是，如果计数值变为负值，getc 就调用函数 _fillbuf 填充缓冲区，重新初始化结构的内容，并返回一个字符。返回的字符为 unsigned 类型。以确保所有的字符为正值。

尽管在这里我们并不想讨论一些细节，但程序中还是给出了 putc 函数的定义，以表明它的操作与 getc 函数非常类似，当缓冲区满时，它将调用函数_flushbuf。此外，我们还在其中包含了访问错误输出、文件结束状态和文件描述符的宏。

下面我们来着手编写函数 fopen。fopen 函数的主要功能是打开文件，定位到合适的位置，设置标志位以指示相应的状态。它不分配任何缓冲区空间，缓冲区的分配是在第一次读文件时由函数_fillbuf 完成的。

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666    /* RW for owner, group, others */

FILE *fopen(char *name, char *mode)
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break;       /* found free slot */
    if (fp >= _iob + OPEN_MAX)   /* no free slots */
```

```
        return NULL;

    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1)        /* couldn't access name */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

该版本的 fopen 函数没有涉及标准 C 的所有访问模式，但是，加入这些模式并不需要增加多少代码。特别是，该版本的 fopen 不能识别表示二进制访问方式的 b 标志，这是因为，在 UNIX 系统中这种方式是没有意义的。同时，它也不能识别允许同时进行读和写的+标志。

对于某一特定的文件，第一次调用 getc 函数时计数值为 0，这样就必须调用一次函数 _fillbuf。如果_fillbuf 发现文件不是以读方式打开的，它将立即返回 EOF；否则，它将试图分配一个缓冲区（如果读操作是以缓冲方式进行的话）。

建立缓冲区后，_fillbuf 调用 read 填充此缓冲区，设置计数值和指针，并返回缓冲区中的第一个字符。随后进行的_fillbuf 调用会发现缓冲区已经分配。

```
#include "syscalls.h"

/* _fillbuf:  allocate and fill input buffer */
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag&(_READ|_EOF_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL)    /* no buffer yet */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF;      /* can't get buffer */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}
```

最后一件事情便是如何执行这些函数。我们必须定义和初始化数组_iob 中的 stdin、

stdout 和 stderr 值：

```
FILE _iob[OPEN_MAX] = {   /* stdin, stdout, stderr */
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE, | _UNBUF, 2 }
};
```

该结构中 flag 部分的初值表明,将对 stdin 执行读操作、对 stdout 执行写操作、对 stderr 执行缓冲方式的写操作。

练习 8-2　用字段代替显式的按位操作，重写 fopen 和_fillbuf 函数。比较相应代码的长度和执行速度。

练习 8-3　设计并编写函数_flushbuf、fflush 和 fclose。

练习 8-4　标准库函数

```
int fseek(FILE *fp, long offset, int origin)
```

类似于函数 lseek，所不同的是，该函数中的 fp 是一个文件指针而不是文件描述符，且返回值是一个 int 类型的状态而非位置值。编写函数 fseek，并确保该函数与库中其它函数使用的缓冲能够协同工作。

8.6. 实例——目录列表

我们常常还需要对文件系统执行另一种操作，以获得文件的有关信息，而不是读取文件的具体内容。目录列表程序便是其中的一个例子，比如 UNIX 命令 ls，它打印一个目录中的文件名以及其它一些可选信息，如文件长度、访问权限等等。MS-DOS 操作系统中的 dir 命令也有类似的功能。

由于 UNIX 中的目录就是一种文件，因此，ls 只需要读此文件就可获得所有的文件名。但是，如果需要获取文件的其它信息，比如长度等，就需要使用系统调用。在其它一些系统中，甚至获取文件名也需要使用系统调用，例如在 MS-DOS 系统中即如此。无论实现方式是否同具体的系统有关，我们需要提供一种与系统无关的访问文件信息的途径。

以下将通过程序 fsize 说明这一点。fsize 程序是 ls 命令的一个特殊形式，它打印命令行参数表中指定的所有文件的长度。如果其中一个文件是目录，则 fsize 程序将对此目录递归调用自身。如果命令行中没有任何参数，则 fsize 程序处理当前目录。

我们首先回顾 UNIX 文件系统的结构。在 UNIX 系统中，目录就是文件，它包含了一个文件名列表和一些指示文件位置的信息。“位置”是一个指向其它表（即 i 结点表）的索引。文件的 i 结点是存放除文件名以外的所有文件信息的地方。目录项通常仅包含两个条目：文件名和 i 结点编号。

遗憾的是，在不同版本的系统中，目录的格式和确切的内容是不一样的。因此，为了分离出不可移植的部分，我们把任务分成两部分。外层定义了一个称为 Dirent 的结构和 3 个函数 opendir、readdir 和 closedir，它们提供与系统无关的对目录项中的名字和 i 结点编号的访问。我们将利用此接口编写 fsize 程序，然后说明如何在与 Version 7 和 System V UNIX 系统的目录结构相同的系统上实现这些函数。其它情况留作练习。

结构Dirent包含i结点编号和文件名。文件名的最大长度由NAMZ_MAX设定，NAME_MAX的值由系统决定。opendir返回一个指向称为DIR的结构的指针，该结构与结构FILE类似，它将被readdir和closedir使用。所有这些信息存放在头文件dirent.h中。

```
#define NAME_MAX   14  /* longest filename component; */
                           /* system-dependent */

typedef struct {      /* portable directory entry */
   long ino;                /* inode number */
   char name[NAME_MAX+1];    /* name + '\0' terminator */
} Dirent;

typedef struct {      /* minimal DIR: no buffering, etc. */
   int fd;             /* file descriptor for the directory */
   Dirent d;           /* the directory entry */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

系统调用stat以文件名作为参数，返回文件的i结点中的所有信息；若出错，则返回-1。如下所示：

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

它用文件name的i结点信息填充结构stbuf。头文件<sys/stat.h>中包含了描述stat的返回值的结构。该结构的一个典型形式如下所示：

```
struct stat   /* inode information returned by stat */
{
   dev_t    st_dev;     /* device of inode */
   ino_t    st_ino;     /* inode number */
   short    st_mode;    /* mode bits */
   short    st_nlink;   /* number of links to file */
   short    st_uid;     /* owners user id */
   short    st_gid;     /* owners group id */
   dev_t    st_rdev;    /* for special files */
   off_t    st_size;    /* file size in characters */
   time_t   st_atime;   /* time last accessed */
   time_t   st_mtime;   /* time last modified */
   time_t   st_ctime;   /* time originally created */
};
```

该结构中大部分的值已在注释中进行了解释。dev_t和ino_t等类型在头文件<sys/types.h>中定义，程序中必须包含此文件。

st_mode项包含了描述文件的一系列标志，这些标志在<sys/stat.h>中定义。我们只需要处理文件类型的有关部分：

```
#define S_IFMT   0160000  /* type of file: */
#define S_IFDIR  0040000  /* directory */
#define S_IFCHR  0020000  /* character special */
```

```
#define S_IFBLK   0060000  /* block special */
#define S_IFREG   0010000  /* regular */
/* ... */
```

下面我们来着手编写程序 fsize。如果由 stat 调用获得的模式说明某文件不是一个目录，就很容易获得该文件的长度，并直接输出。但是，如果文件是一个目录，则必须逐个处理目录中的文件。由于该目录可能包含子目录，因此该过程是递归的。

主程序 main 处理命令行参数，并将每个参数传递给函数 fsize。

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>     /* flags for read and write */
#include <sys/types.h>  /* typedefs */
#include <sys/stat.h>   /* structure returned by stat */
#include "dirent.h"

void fsize(char *)

/* print file name */
main(int argc, char **argv)
{
    if (argc == 1)  /* default: current directory */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

函数fsize打印文件的长度。但是，如果此文件是一个目录，则fsize首先调用dirwalk函数处理它所包含的所有文件。注意如何使用文件<sys/stat.h>中的标志名 S_IFMT 和 S_IFDIR 来判定文件是不是一个目录。括号是必须的，因为&运算符的优先级低于==运算符的优先级。

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize:  print the name of file "name" */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

函数 dirwalk 是一个通用的函数，它对目录中的每个文件都调用函数 fcn 一次。它首先打开目录，循环遍历其中的每个文件，并对每个文件调用该函数，然后关闭目录返回。因为 fsize 函数对每个目录都要调用 dirwalk 函数，所以这两个函数是相互递归调用的。

```
#define MAX_PATH 1024

/* dirwalk:  apply fcn to all files in dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: can't open %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
            || strcmp(dp->name, ".."))
            continue;    /* skip self and parent */
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
            fprintf(stderr, "dirwalk: name %s %s too long\n",
                dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

每次调用 `readdir` 都将返回一个指针，它指向下一个文件的信息。如果目录中已没有待处理的文件，该函数将返回 `NULL`。每个目录都包含自身“ `.`”和父目录“ `..`”的项目，在处理时必须跳过它们，否则将会导致无限循环。

到现在这一步为止，代码与目录的格式无关。下一步要做的事情就是在某个具体的系统上提供一个 `opendir`、`readdir` 和 `closedir` 的最简单版本。以下的函数适用于 Version 7 和 System V UNIX 系统，它们使用了头文件（`sys/dir.h`>中的目录信息，如下所示：

```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif
struct direct {   /* directory entry */
    ino_t d_ino;          /* inode number */
    char  d_name[DIRSIZ];  /* long name does not have '\0' */
};
```

某些版本的系统支持更长的文件名和更复杂的目录结构。

类型 `ino_t` 是使用 `typedef` 定义的类型，它用于描述 i 结点表的索引。在我们通常使用的系统中，此类型为 `unsigned short`，但是这种信息不应在程序中使用。因为不同的系统中该类型可能不同，所以使用 `typedef` 定义要好一些。所有的“系统”类型可以在文件 `<sys/types.h>`）中找到。

`opendir` 函数首先打开目录，验证此文件是一个目录（调用系统调用 `fstat`，它与 `stat` 类似，但它以文件描述符作为参数），然后分配一个目录结构，并保存信息：

```
int fstat(int fd, struct stat *);
```

```
/* opendir:  open a directory for readdir calls */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
     || fstat(fd, &stbuf) == -1
     || (stbuf.st_mode & S_IFMT) != S_IFDIR
     || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}
```

closedir 函数用于关闭目录文件并释放内存空间：

```
/* closedir:  close directory opened by opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

最后，函数 readdir 使用 read 系统调用读取每个目录项。如果某个目录位置当前没有使用（因为删除了一个文件），则它的 i 结点编号为 0，并跳过该位置。否则，将 i 结点编号和目录名放在一个 static 类型的结构中，并给用户返回一个指向此结构的指针。每次调用 readdir 函数将覆盖前一次调用获得的信息。

```
#include <sys/dir.h>   /* local directory structure */

/* readdir:  read directory entries in sequence */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf;  /* local directory structure */
    static Dirent  d;     /* return: portable structure */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
                == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0) /* slot not in use */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0';  /* ensure termination */
        return &d;
    }
    return NULL;
}
```

尽管 fsize 程序非常特殊，但是它的确说明了一些重要的思想。首先，许多程序并不是“系统程序”，它们仅仅使用由操作系统维护的信息。对于这样的程序，很重要的一点是，信息的表示仅出现在标准头文件中，使用它们的程序只需要在文件中包含这些头文件即可，而不需要包含相应的声明。其次，有可能为与系统相关的对象创建一个与系统无关的接口。标

准库中的函数就是很好的例子。

练习 8-5　　修改 `fsize` 程序，打印 i 结点项中包含的其它信息。

8.7.　实例——存储分配程序

我们在第 5 章给出了一个功能有限的面向栈的存储分配程序。本节将要编写的版本没有限制，可以以任意次序调用 `malloc` 和 `free`。`malloc` 在必要时调用操作系统以获取更多的存储空间。这些程序说明了通过一种与系统无关的方式编写与系统有关的代码时应考虑的问题，同时也展示了结构、联合和 `typedef` 的实际应用。

`malloc` 并不是从一个在编译时就确定的固定大小的数组中分配存储空间，而是在需要时向操作系统申请空间。因为程序中的某些地方可能不通过 `malloc` 调用申请空间（也就是说，通过其它方式申请空间），所以，`malloc` 管理的空间不一定是连续的。这样，空闲存储空间以空闲块链表的方式组织，每个块包含一个长度、一个指向下一块的指针以及一个指向自身存储空间的指针。这些块按照存储地址的升序组织，最后一块（最高地址）指向第一块（参见图 8-1）。

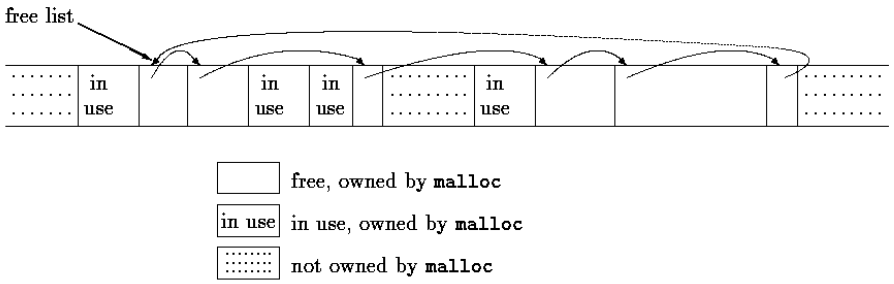


图 8-1

当有申请请求时，`malloc` 将扫描空闲块链表，直到找到一个足够大的块为止。该算法称为“首次适应”（first fit）；与之相对的算法是“最佳适应”（best fit），它寻找满足条件的最小块。如果该块恰好与请求的大小相符合，则将它从链表中移走并返回给用户。如果该块太大，则将它分成两部分：大小合适的块返回给用户，剩下的部分留在空闲块链表中。如果找不到一个足够大的块，则向操作系统申请一个大块并加入到空闲块链表中。

释放过程也是首先搜索空闲块链表，以找到可以插入被释放块的合适位置。如果与被释放块相邻的任一边是一个空闲块，则将这两个块合成一个更大的块，这样存储空间不会有太多的碎片。因为空闲块链表是以地址的递增顺序链接在一起的，所以很容易判断相邻的块是否空闲。

我们在第 5 章中曾提出了这样的问题，即确保由 `malloc` 函数返回的存储空间满足将要保存的对象的对齐要求。虽然机器类型各异，但是，每个特定的机器都有一个最受限的类型：如果最受限的类型可以存储在某个特定的地址中，则其它所有的类型也可以存放在此地址中。在某些机器中，最受限的类型是 `double` 类型；而在另外一些机器中，最受限的类型是 `int` 或 `long` 类型。

空闲块包含一个指向链表中下一个块的指针、一个块大小的记录和一个指向空闲空间本

身的指针。位于块开始处的控制信息称为“头部“。为了简化块的对齐，所有块的大小都必须是头部大小的整数倍，且头部已正确地对齐。这是通过一个联合实现的，该联合包含所需的头部结构以及一个对齐要求最受限的类型的实例，在下面这段程序中，我们假定 long 类型为最受限的类型：

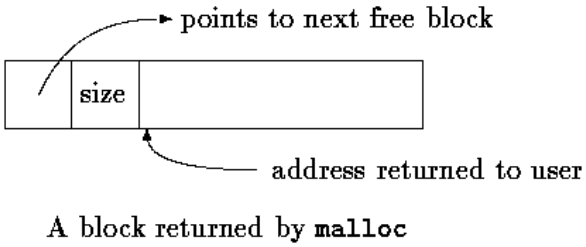
```
typedef long Align;    /* for alignment to long boundary */

union header {        /* block header */
   struct {
      union header *ptr; /* next block if on free list */
      unsigned size;    /* size of this block */
   } s;
   Align x;          /* force alignment of blocks */
};

typedef union header Header;
```

在该联合中，Align 字段永远不会被使用，它仅仅用于强制每个头部在最坏的情况下满足对齐要求。

在 malloc 函数中，请求的长度（以字符为单位）将被舍入，以保证它是头部大小的整数倍。实际分配的块将多包含一个单元，用于头部本身。实际分配的块的大小将被记录在头部的 size 字段中。malloc 函数返回的指引将指向空闲空间，而不是块的头部。用户可对获得的存储空间进行任何操作，但是，如果在分配的存储空间之外写入数据，则可能会破坏块链表。图 8-2 表示由 malloc 返回的块。



A block returned by **malloc**

图 8-2　malloc 返回的块

其中的 size 字段是必需的，因为由 malloc 函数控制的块不一定是连续的，这样就不可能通过指针算术运算计算其大小。

变量 base 表示空闲块链表的头部。第一次调用 malloc 函数时，freep 为 NULL，系统将创建一个退化的空闲块链表，它只包含一个大小为 0 的块，且该块指向它自己。任何情况下，当请求空闲空间时，都将搜索空闲块链表。搜索从上一次找到空闲块的地方（freep）开始。该策略可以保证链表是均匀的。如果找到的块太大，则将其尾部返回给用户，这样，初始块的头部只需要修改 size 字段即可。在任何情况下，返回给用户的指针都指向块内的空闲存储空间，即比指向头部的指针大一个单元。

```
static Header base;      /* empty list to get started */
static Header *freep = NULL;    /* start of free list */

/* malloc:  general-purpose storage allocator */
void *malloc(unsigned nbytes)
```

```
{
    Header *p, *prevp;
    Header *moreroce(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
    if ((prevp = freep) == NULL) {   /* no free list yet */
        base.s.ptr = freeptr = prevptr = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {  /* big enough */
            if (p->s.size == nunits)  /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {            /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)  /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* none left */
    }
}
```

函数 morecore 用于向操作系统请求存储空间，其实现细节因系统的不同而不同。因为向系统请求存储空间是一个开销很大的操作，因此，我们不希望每次调用 malloc 函数时都执行该操作，基于这个考虑，morecore 函数请求至少 NALLOC 个单元。这个较大的块将根据需要分成较小的块。在设置完 size 字段之后，morecore 函数调用 free 函数把多余的存储空间插入到空闲区域中。

UNIX 系统调用 sbrk(n)返回一个指针，该指针指向 n 个字节的存储空间。如果没有空闲空间，尽管返回 NULL 可能更好一些，但 sbrk 调用返回-1。必须将-1 强制转换为 char * 类型，以便与返回值进行比较。而且，强制类型转换使得该函数不会受不同机器中指针表示的不同的影响。但是，这里仍然假定，由 sbrk 调用返回的指向不同块的多个指针之间可以进行有意义的比较。ANSI 标准并没有保证这一点，它只允许指向同一个数组的指针间的比较。因此，只有在一般指针间的比较操作有意义的机器上，该版本的 malloc 函数才能够移植。

```
#define NALLOC  1024   /* minimum #units to request */

/* morecore:  ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)   /* no space at all */
        return NULL;
    up = (Header *) cp;
```

```
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

我们最后来看一下 free 函数。它从 freep 指向的地址开始，逐个扫描空闲块链表，寻找可以插入空闲块的地方。该位置可能在两个空闲块之间，也可能在链表的末尾。在任何一种情况下，如果被释放的块与另一空闲块相邻，则将这两个块合并起来。合并两个块的操作很简单，只需要设置指针指向正确的位置，并设置正确的块大小就可以了。

```
/* free:  put block ap in free list */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;    /* point to  block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;  /* freed block at start or end of arena */

    if (bp + bp->size == p->s.ptr) {    /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->size == bp) {           /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

虽然存储分配从本质上是与机器相关的，但是，以上的代码说明了如何控制与具体机器相关的部分，并将这部分程序控制到最少量。typedef 和 union 的使用解决了地址的对齐（假定 sbrk 返回的是合适的指针）问题。类型的强制转换使得指针的转换是显式进行的，这样做甚至可以处理设计不够好的系统接口问题。虽然这里所讲的内容只涉及到存储分配，但是，这种通用方法也适用于其它情况。

练习 8-6　标准库函数 calloc(n, size)返回一个指针，它指向 n 个长度为 size 的对象，且所有分配的存储空间都被初始化为 0。通过调用或修改 malloc 函数来实现 calloc 函数。

练习 8-7　malloc 接收对存储空间的请求时，并不检查请求长度的合理性；而 free 则认为被释放的块包含一个有效的长度字段。改进这些函数，使它们具有错误检查的功能。

练习 8-8　编写函数 bfree(p, n)，释放一个包含 n 个字符的任意块 p，并将它放入由 malloc 和 free 维护的空闲块链表中。通过使用 bfree，用户可以在任意时刻向空闲块链表中添加一个静态或外部数组。

附录A 参考手册

A.1 引言

本手册描述的 C 语言是 1988 年 10 月 31 日提交给 ANSI 的草案，批准号为“美国国家信息系统标准——C 程序设计语言，X3.159-1989”。尽管我们已尽最大努力，力求准确地将该手册作为 C 语言的指南介绍给读者，但它毕竟不是标准本身，而仅仅只是对标准的一个解释而已。

该手册的组织与标准基本类似，与本书的第 1 版也类似，但是对细节的组织有些不同。本手册给出的语法与标准是相同的，但是，其中少量元素的命名可能有些不同，词法记号和预处理器的定义也没有形式化。

本手册中，说明部分的文字指出了 ANSI 标准 C 语言与本书第 1 版定义的 C 语言或其它编译器支持的语言之间的差别。

A.2 词法规则

程序由存储在文件中的一个或多个翻译单元（translation unit）组成。程序的翻译分几个阶段完成，这部分内容将在 A.12 节中介绍。翻译的第一阶段完成低级的词法转换，执行以字符#开头的行中的指令，并进行宏定义和宏扩展。在预处理（将在 A.12 节中介绍）完成后，程序被归约成一个记号序列。

A.2.1 记号

C 语言中共有 6 类记号：标识符、关键字、常量、字符串字面值、运算符和其它分隔符。空格，横向制表符和纵向制表符、换行符，换页符和注释（统称空白符）在程序中仅用来分隔记号，因此将被忽略。相邻的标识符、关键字和常量之间需要用空白符来分隔。

如果到某一字符为止的输入流被分隔成若干记号，那么，下一个记号就是后续字符序列中可能构成记号的最长的字符串。

A.2.2 注释

注释以字符/*开始，以*/结束。注释不能够嵌套，也不能够出现在字符串字面值或字符字面值中。

A.2.3 标识符

标识符是由字母和数字构成的序列。第一个字符必须是字母，下划线“ _”也被看成是字母。大写字母和小写字母是不同的。标识符可以为任意长度。对于内部标识符来说，至少前 31 个字母是有效的，在某些实现中，有效的字符数可能更多。内部标识符包括预处理器的宏名和其它所有没有外部连接（参见 A.11.2 节）的名字。带有外部连接的标识符的限制更严格一些，实现可能只认为这些标识符的前 6 个字符是有效的，而且有可能忽略大小写的不同。

A.2.4 关键字

下列标识符被保留作为关键字，且不能用于其它用途：

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

某些实现还把 fortran 和且 asm 保留为关键字。

说明：关键字 const、signed 和 volatile 是 ANSI 标准中新增加的；enum 和 void 是第 1 版后新增加的，现已被广泛应用；entry 曾经被保留为关键字但从未被使用过，现在已经不是了。

A.2.5 常量

常量有多种类型。每种类型的常量都有一个数据类型。基本数据类型将在 A.4.2 节讨论。

常量：

整型常量
字符常量
浮点常量
枚举常量

1. 整型常量

整型常量由一串数字组成。如果它以数字 0 开头，则为八进制数，否则为十进制数。八进制常量不包括数字 8 和 9，以 0x 和 0X 开头的数字序列表示十六进制数，十六进制数包含从 a（或 A）到 f（或 F）的字母，它们分别表示数值 10 到 15。

整型常量若以字母 u 或 U 为后缀，则表示它是一个无符号数；若以字母 l 或 L 为后缀，则表示它是一个长整型数；若以字母 UL 为后缀，则表示它是一个无符号长整型数。

整型常量的类型同它的形式、值和后缀有关（有关类型的讨论，参见 A.4 节）。如果它没有后缀且是十进制表示，则其类型很可能是 int、long int 或 unsigned long int。如

果它没有后缀且是八进制或十六进制表示，则其类型很可能是 int、unsigned int、long int 或 unsigned long int。如果它的后缀为 u 或 U，则其类型很可能是 unsigned int 或 ungigned long int。如果它的后缀为 l 或 L，则其类型很可能是 long int 或 unsigned long int。

说明：ANSI 标准中，整型常量的类型比第 1 版要复杂得多。在第 1 版中，大的整型常量仅被看做是 long 类型。U 后缀是新增加的。

2. 字符常量

字符常量是用单引号引起来的一个或多个字符构成的序列，如'x'。单字符常量的值是执行时机器字符集中此字符对应的数值，多字符常量的值由具体实现定义。

字符常量不包括字符'和换行符，可以使用以下转义字符序列表示这些字符以及其它一些字符：

换行符	NL(LF)	\n	反斜杠	\	\\
横向制表符	HT	\t	问号	?	\?
纵向制表符	VT	\v	单引号	'	\'
回退符	BS	\b	双引号	"	\"
回车符	CR	\r	八进制数	ooo	\ooo
换页符	FF	\f	十六进制数	hh	\xhh
响铃符	BEL	\a			

转义序列\ooo 由反斜杠后跟 1 个、2 个或 3 个八进制数字组成，这些八进制数字用来指定所期望的字符的值。\0（其后没有数字）便是一个常见的例子，它表示字符 NUL。转义序列\xhh 中，反斜杠后面紧跟 x 以及十六进制数字，这些十六进制数用来指定所期望的字符的值。数字的个数没有限制，但如果字符值超过最大的字符值，该行为是未定义的。对于八进制或十六进制转义字符，如果实现中将类型 char 看做是带符号的，则将对字符值进行符号扩展，就好像它被强制转换为 char 类型一样。如果\后面紧跟的字符不在以上指定的字符中，则其行为是未定义的。

在 C 语言的某些实现中，还有一个扩展的字符集，它不能用 char 类型表示。扩展集中的常量要以一个前导符 L 开头（例如 L'x'。），称为宽字符常量。这种常量的类型为 wchar_t。这是一种整型类型，定义在标准头文件<stddef.h>中。与通常的字符常量一样，宽字符常量可以使用八进制或十六进制转义字符序列；但是，如果值超过 wchar_t 可以表示的范围，则结果是未定义的。

说明：某些转义序列是新增加的，特别是十六进制字符的表示。扩展字符也是新增加的。通常情况下，美国和西欧所用的字符集可以用 char 类型进行编码，增加 wchar_t 的主要目的是为了表示亚洲的语言。

3. 浮点常量

浮点常量由整数部分、小数点、小数部分、一个 e 或 E、一个可选的带符号整型类型的指数和一个可选的表示类型的后缀（即 f、F、l 或 L 之一）组成。整数和小数部分均由数字序列组成。可以没有整数部分或小数部分（但不能两者都没有），还可以没有小数点或者 e 和指数部分（但不能两者都没有）。浮点常量的类型由后缀确定，F 或 f 后缀表示它是 float 类型；l 或 L 后缀表明它是 long double 类型；没有后缀则表明是 double 类型。

说明：浮点常量的后缀是新增加的。

4. 枚举常量

声明为枚举符的标识符是 `int` 类型的常量（参见 A.8.4 节）。

A.2.6 字符串字面值

字符串字面值（string literal）也称为字符串常量，是用双引号引起来的一个字符序列，如“…”。字符串的类型为“字符数组”，存储类为 `static`（参见 A.4 节），它使用给定的字符进行初始化。对相同的字符串字面值是否进行区分取决于具体的实现。如果程序试图修改字符串字面值，则行为是未定义的。

我们可以把相邻的字符串字面值连接为一个单一的字符串。执行任何连接操作后，都将在字符串的后面增加一个空字节`\0`，这样，扫描字符串的程序便可以找到字符串的结束位置。字符串字面值不包含换行符和双引号字符，但可以用与字符常量相同的转义字符序列表示它们。

与字符常量一样，扩展字符集中的字符串字面值也以前导符 `L` 表示，如 `L"…"`。宽字符字符串字面值的类型为“ `wchar_t` 类型的数组”。将普通字符串字面值和宽字符字符串字面值进行连接的行为是未定义的。

说明：下列规定都是 ANSI 标准中新增加的：字符串字面值不必进行区分、禁止修改字符串字面值以及允许相邻字符串字面值进行连接。宽字符字符串字面值也是 ANSI 标准中新增加的。

A.3 语法符号

在本手册用到的语法符号中，语法类别用楷体及斜体字表示。文字和字符以打字型字体表不。多个候选类别通常列在不同的行中，但在一些情况下，一组字符长度较短的候选项可以放在一行中，并以短语“ one of”标识。可选的终结符或非终结符带有下标“ *opt*”。例如：

{ 表达式 *opt* }

表示一个括在花括号中的表达式，该表达式是可选的。A.13 节对语法进行了总结。

说明：与本书第 1 版给出的语法所不同的是，此处给出的语法将表达式运算符的优先级和结合性显式表达出来了。

A.4 标识符的含义

标识符也称为名字，可以指代多种实体：函数、结构标记、联合标记和枚举标记；结构成员或联合成员；枚举常量；类型定义名；标号以及对象等。对象有时也称为变量，它是一个存储位置。对它的解释依赖于两个主要属性：存储类和类型。存储类决定了与该标识对象相关联的存储区域的生存期，类型决定了标识对象中值的含义。名字还具有一个作用域和一个连接。作用域即程序中可以访问此名字的区域，连接决定另一作用域中的同一个名字是否

指向同一个对象或函数。作用域和连接将在 A.11 节中讨论。

A.4.1 存储类

存储类分为两类：自动存储类（automatic）和静态存储类（static）。声明对象时使用的一些关键字和声明的上下文共同决定了对象的存储类。自动存储类对象对于一个程序块（参见 A.9.3 节）来说是局部的，在退出程序块时该对象将消失。如果没有使用存储类说明符，或者如果使用了 `auto` 限定符，则程序块中的声明生成的都是自动存储类对象。声明为 `register` 的对象也是自动存储类对象，并且将被存储在机器的快速寄存器中（如果可能的话）。

静态对象可以是某个程序块的局部对象，也可以是所有程序块的外部对象。无论是哪一种情况，在退出和再进入函数或程序块时其值将保持不变。在一个程序块（包括提供函数代码的程序块）内，静态对象用关键字 `static` 声明。在所有程序块外部声明且与函数定义在同一级的对象总是静态的。可以通过 `static` 关键字将对象声明为某个特定翻译单元的局部对象，这种类型的对象将具有内部连接。当省略显式的存储类或通过关键字 `extern` 进行声明时，对象对整个程序来说是全局可访问的，并且具有外部连接。

A.4.2 基本类型

基本类型包括多种。附录 B 中描述的标准头文件`<limits.h>`中定义了本地实现中每种类型的最大值和最小值。附录 B 给出的数值表示最小的可接受限度。

声明为字符（`char`）的对象要大到足以存储执行字符集中的任何字符。如果字符集中的某个字符存储在一个 `char` 类型的对象中，则该对象的值等于字符的整型编码值，并且是非负值。其它类型的对象也可以存储在 `char` 类型的变量中，但其取值范围，特别是其值是否带符号，同具体的实现有关。

以 `unsigned char` 声明的无符号字符与普通字符占用同样大小的空间，但其值总是非负的。以 `signed char` 显式声明的带符号字符与普通字符也占用同样大小的空间。

说明：本书的第 1 版中没有 `unsigned char` 类型，但这种用法很常见。`signed char` 是新增加的。

除 `char` 类型外，还有 3 种不同大小的整型类型：`short int`、`int` 和 `long int`。普通 `int` 对象的长度与由宿主机器的体系结构决定的自然长度相同。其它类型的整型可以满足各种特殊的用途。较长的整数至少要占有与较短整数一样的存储空间；但是具体的实现可以使得一般整型（`int`）与短整型（`short int`）或长整型（`long int`）具有同样的大小。除非特别说明，`int` 类型都表示带符号数。

以关键字 `unsigned` 声明的无符号整数遵守算术模 2^n 的规则，其中，n 是表示相应整数的二进制位数，这样，对无符号数的算术运算永远不会溢出。可以存储在带符号对象中的非负值的集合是可以存储在相应的无符号对象中的值的子集，并且，这两个集合的重叠部分的表示是相同的。

单精度浮点数（`float`）、双精度浮点数（`double`）和多精度浮点数（`long double`）中的任何类型都可能是同义的，但精度从前到后是递增的。

说明：long double 是新增加的类型。在第 1 版中，long float 与 double 类型等价，但现在是不相同的。

枚举是一个具有整型值的特殊的类型。与每个枚举相关联的是一个命名常量的集合（参见 A.8.4 节）。枚举类型类似于整型。但是，如果某个特定枚举类型的对象的赋值不是其常量中的一个，或者赋值不是一个同类型的表达式，则编译器通常会产生警告信息。

因为以上这些类型的对象都可以被解释为数字，所以，可以将它们统称为算术类型。char 类型、各种大小的 int 类型（无论是否带符号）以及枚举类型都统称为整型类型（integral type）。类型 float、double 和 long double 统称为浮点类型（floating type）。

void 类型说明一个值的空集合，它常被用来说明不返回任何值的函数的类型。

A.4.3 派生类型

除基本类型外，我们还可以通过以下几种方法构造派生类型，从概念来讲，这些派生类型可以有无限多个：

- 给定类型对象的数组
- 返回给定类型对象的函数
- 指向给定类型对象的指针
- 包含一系列不同类型对象的结构
- 可以包含多个不同类型对象中任意一个对象的联合

一般情况下，这些构造对象的方法可以递归使用。

A.4.4 类型限定符

对象的类型可以通过附加的限定符进行限定。声明为 const 的对象表明此对象的值不可以修改；声明为 volatile 的对象表明它具有与优化相关的特殊属性。限定符既不影响对象取值的范围，也不影响其算术属性。限定符将在 A.8.2 节中讨论。

A.5 对象和左值

对象是一个命名的存储区域，左值（lvalue）是引用某个对象的表达式。具有合适类型与存储类的标识符便是左值表达式的一个明显的例子。某些运算符可以产生左值。例如，如果 E 是一个指针类型的表达式，*E 则是一个左值表达式，它引用由 E 指向的对象。名字“左值”来源于赋值表达式 E1=E2，其中，左操作数 E1 必须是一个左值表达式。对每个运算符的讨论需要说明此运算符是否需要一个左值操作数以及它是否产生一个左值。

A.6 转换

根据操作数的不同，某些运算符会引起操作数的值从某种类型转换为另一种类型。本节将说明这种转换产生的结果。A.6.5 节将讨论大多数普通运算符所要求的转换，我们在讲解每

个运算符时将做一些补充。

A.6.1 整型提升

在一个表达式中，凡是可以使用整型的地方都可以使用带符号或无符号的字符、短整型或整型位字段，还可以使用枚举类型的对象。如果原始类型的所有值都可用 `int` 类型表示，则其值将被转换为 `int` 类型；否则将被转换为 `unsigned int` 类型。这一过程称为整型提升（integral promotion）。

A.6.2 整型转换

将任何整数转换为某种指定的无符号类型数的方法是：以该无符号类型能够表示的最大值加 1 为模，找出与此整数同余的最小的非负值。在对二的补码表示中，如果该无符号类型的位模式较窄，这就相当于左截取；如果该无符号类型的位模式较宽，这就相当于对带符号值进行符号扩展和对无符号值进行 0 填充。

将任何整数转换为带符号类型时，如果它可以在新类型中表示出来，则其值保持不变，否则它的值同具体的实现有关。

A.6.3 整数和浮点数

当把浮点类型的值转换为整型时，小数部分将被丢弃。如果结果值不能用整型表示，则其行为是未定义的。特别是，将负的浮点数转换为无符号整型的结果是没有定义的。

当把整型值转换为浮点类型时，如果该值在该浮点类型可表示的范围内但不能精确表示，则结果可能是下一个较高或较低的可表示值。如果该值超出可表示的范围，则其行为是未定义的。

A.6.4 浮点类型

将一个精度较低的浮点值转换为相同或更高精度的浮点类型时，它的值保持不变。将一个较高精度的浮点类型值转换为较低精度的浮点类型时，如果它的值在可表示范围内，则结果可能是下一个较高或较低的可表示值。如果结果在可表示范围之外，则其行为是未定义的。

A.6.5 算术类型转换

许多运算符都会以类似的方式在运算过程中引起转换，并产生结果类型。其效果是将所有操作数转换为同一公共类型，并以此作为结果的类型。这种方式的转换称为普通算术类型转换。

首先，如果任何一个操作数为 `long double` 类型，则将另一个操作数转换为 `long double` 类型。

否则，如果任何一个操作数为 double 类型，则将另一个操作数转换为 double 类型。

否则，如果任何一个操作数为 float 类型，则将另一个操作数转换为 float 类型。

否则，同时对两个操作数进行整型提升；然后，如果任何一个操作数为 unsigned long int 类型，则将另一个操作数转换为 unsigned long int 类型。

否则，如果一个操作数为 long int 类型且另一个操作数为 unsigned int 类型，则结果依赖于 long int 类型是否可以表示所有的 unsigned int 类型的值。如果可以，则将 unsigned int 类型的操作数转换为 long int；如果不可以，则将两个操作数都转换为 unsigned long int 类型。

否则，如果一个操作数为 long int 类型，则将另一个操作数转换为 long int 类型。

否则，如果任何一个操作数为 unsigned int 类型，则将另一个操作数转换为 unsigned int 类型。

否则，将两个操作数都转换为 int 类型。

说明：这里有两个变化。第一，对 float 类型操作数的算术运算可以只用单精度而不是双精度；而在第 1 版中规定，所有的浮点运算都是双精度。第二，当较短的无符号类型与较长的带符号类型一起运算时，不将无符号类型的属性传递给结果类型；而在第 1 版中，无符号类型总是处于支配地位。新规则稍微复杂一些，但减少了无符号数与带符号数混合使用情况下的麻烦，当一个无符号表达式与一个具有同样长度的带符号表达式相比较时，结果仍然是无法预料的。

A.6.6 指针和整数

指针可以加上或减去一个整型表达式。在这种情况下，整型表达式的转换按照加法运算符的方式进行（参见 A.7.7 节）。

两个指向同一数组中同一类型的对象的指针可以进行减法运算，其结果将被转换为整型；转换方式按照减法运算符的方式进行（参见 A.7.7 节）。

值为 0 的整型常量表达式或强制转换为 void *类型的表达式可通过强制转换、赋值或比较操作转换为任意类型的指针。其结果将产生一个空指针，此空指针等于指向同一类型的另一空指针，但不等于任何指向函数或对象的指针。

还允许进行指针相关的其它某些转换，但其结果依赖于具体的实现。这些转换必须由一个显式的类型转换运算符或强制类型转换来指定（参见 A.7.5 节和 A.8.8 节）。

指针可以转换为整型，但此整型必须足够大；所要求的大小依赖于具体的实现。映射函数也依赖于具体的实现。

整型对象可以显式地转换为指针。这种映射总是将一个足够宽的从指针转换来的整数转换为同一个指针，其它情况依赖于具体的实现。

指向某一类型的指针可以转换为指向另一类型的指针，但是，如果该指针指向的对象不满足一定的存储对齐要求，则结果指针可能会导致地址异常。指向某对象的指针可以转换为一个指向具有更小或相同存储对齐限制的对象的指针，并可以保证原封不动地再转换回来。

“对齐”的概念依赖于具体的实现，但 char 类型的对象具有最小的对齐限制。我们将在 A.6.8 节的讨论中看到，指针也可以转换为 void *类型，并可原封不动地转换回来。

一个指针可以转换为同类型的另一个指针，但增加或删除了指针所指的对象类型的限定符（参见 A.4.4 节和 A.8.2 节）的情况除外。如果增加了限定符，则新指针与原指针等价，不同的是增加了限定符带来的限制。如果删除了限定符，则对底层对象的运算仍受实际声明中的限定符的限制。

最后，指向一个函数的指针可以转换为指向另一个函数的指针。调用转换后指针所指的函数的结果依赖于具体的实现。但是，如果转换后的指针被重新转换为原来的类型，则结果与原来的指针一致。

A.6.7 void

void 对象的（不存在的）值不能够以任何方式使用，也不能被显式或隐式地转换为任一非空类型。因为空（void）表达式表示一个不存在的值，这样的表达式只可以用在不需要值的地方，例如作为一个表达式语句（参见 A.9.2 节）或作为逗号运算符的左操作数（参见 A.7.18 节）。

可以通过强制类型转换将表达式转换为 void 类型。例如，在表达式语句中，一个空的强制类型转换将丢掉函数调用的返回值。

说明：void 没有在本书的第 1 版中出现，但是在本书第 1 版出版后，它一直被广泛使用着。

A.6.8 指向 void 的指针

指向任何对象的指针都可以转换为 void *类型，且不会丢失信息。如果将结果再转换为初始指针类型，则可以恢复初始指针。我们在 A.6.6 节中讨论过，执行指针到指针的转换时，一般需要显式的强制转换，这里所不同的是，指针可以被赋值为 void *类型的指针，也可以赋值给 void *类型的指针，并可与 void *类型的指针进行比较。

说明：对 void *指针的解释是新增加的。以前，char *指针扮演着通用指针的角色。ANSI 标准特别允许 void *类型的指针与其它对象指针在赋值表达式和关系表达式中混用，而对其它类型指针的混用则要求进行显式强制类型转换。

A.7 表达式

本节中各主要小节的顺序就代表了表达式运算符的优先级，我们将依次按照从高到低的优先级介绍。举个例子，按照这种关系，A.7.1 至 A.7.6 节中定义的表达式可以用作加法运算符+（参见 A.7.7 节）的操作数。在每一小节中，各个运算符的优先级相同。每个小节中还将讨论该节涉及到的运算符的左、右结合性。A.13 节中给出的语法综合了运算符的优先级和结合性。

运算符的优先级和结合性有明确的规定，但是，除少数例外情况外，表达式的求值次序

没有定义，甚至某些有副作用的子表达式也没有定义。也就是说，除非运算符的定义保证了其操作数按某一特定顺序求值，否则，具体的实现可以自由选择任一求值次序，甚至可以交换求值次序。但是，每个运算符将其操作数生成的值结合起来的方式与表达式的语法分析方式是兼容的。

说明：该规则废除了原先的一个规则，即：当表达式中的运算符在数学上满足交换律和结合律时，可以对表达式重新排序，但是，在计算时可能会不满足结合律。这个改变仅影响浮点数在接近其精度限制时的计算以及可能发生溢出的情况。

C 语言没有定义表达式求值过程中的溢出、除法检查和其它异常的处理。大多数现有 C 语言的实现在进行带符号整型表达式的求值以及赋值时忽略溢出异常，但并不是所有的实现都这么做。对除数为 0 和所有浮点异常的处理，不同的实现采用不同的方式，有时候可以用非标准库函数进行调整。

A.7.1 指针生成

对于某类型 *T*，如果某表达式或子表达式的类型为“*T* 类型的数组”，则此表达式的值是指向数组中第一个对象的指针，并且此表达式的类型将被转换为“指向 *T* 类型的指针”。如果此表达式是一元运算符&或 `sizeof`，则不会进行转换。类似地，除非表达式被用作&运算符的操作数，否则，类型为“返回 *T* 类型值的函数”的表达式将被转换为“指向返回 *T* 类型值的函数的指针”类型。

A.7.2 初等表达式

初等表达式包括标识符、常量、字符串或带括号的表达式。

```
初等表达式:
    标识符
    常量
    字符串
    （表达式）
```

如果按照下面的方式对标识符进行适当的声明，该标识符就是初等表达式。其类型由其声明指定。如果标识符引用一个对象（参见 A.5 节），并且其类型是算术类型、结构、联合或指针，那么它就是一个左值。

常量是初等表达式，其类型同其形式有关。更详细的信息，参见 A.2.5 节中的讨论。

字符串字面值是初等表达式。它的初始类型为“`char` 类型的数组”类型（对于宽字符字符串，则为“`wchar_t` 类型的数组”类型），但遵循 A.7.1 节中的规则。它通常被修改为“指向 `char` 类型（或 `wchar_t` 类型）的指针”类型，其结果是指向字符串中第一个字符的指针。某些初始化程序中不进行这样的转换，详细信息，参见 A.8.7 节。

用括号括起来的表达式是初等表达式，它的类型和值与无括号的表达式相同。此表达式是否是左值不受括号的影响。

A.7.3 后缀表达式

后缀表达式中的运算符遵循从左到右的结合规则。

后缀表达式:
    初等表达式
    后缀表达式[表达式]
    后缀表达式(参数表达式表 $_{opt}$)
    后缀表达式.标识符
    后缀表达式->标识符
    后缀表达式++
    后缀表达式--
参数表达式表:
    赋值表达式
    参数表达式表, 赋值表达式

1 数组引用

带下标的数组引用后缀表达式由一个后缀表达式后跟一个括在方括号中的表达式组成。方括号前的后缀表达式的类型必须为“指向 *T* 类型的指针”，其中 *T* 为某种类型；方括号中表达式的类型必须为整型。结果得到下标表达式的类型为 *T*。表达式 `E1[E2]`在定义上等同于`*((E1) + (E2))`。有关数组引用的更多讨论，参见 A.8.6 节。

2 函数调用

函数调用由一个后缀表达式（称为函数标志符，function designator）后跟由圆括号括起来的赋值表达式列表组成，其中的赋值表达式列表可能为空，并由逗号进行分隔，这些表达式就是函数的参数。如果后缀表达式包含一个当前作用域中不存在的标识符，则此标识符将被隐式地声明，等同于在执行此函数调用的最内层程序块中包含下列声明：

`extern int` 标识符`()`

该后缀表达式（在可能的隐式声明和指针生成之后，参见 A.7.1 节）的类型必须为“指向返回 *T* 类型的函数的指针”，其中 *T* 为某种类型，且函数调用的值的类型为 *T*。

说明：在第 1 版中，该类型被限制为“函数”类型，并且，通过指向函数的指针调用函数时必须有一个显式的*运算符。ANSI 标准允许现有的一些编译器用同样的语法进行函数调用和通过指向函数的指针进行函数调用。旧的语法仍然有效。

通常用术语“实际参数”表示传递给函数调用的表达式，而术语“形式参数”则用来表示函数定义或函数声明中的输入对象（或标识符）。

在调用函数之前，函数的每个实际参数将被复制，所有的实际参数严格地按值传递。函数可能会修改形式参数对象的值（即实际参数表达式的副本），但这个修改不会影响实际参数的值。但是，可以将指针作为实际参数传递，这样，函数便可以修改指针指向的对象的值。

可以通过两种方式声明函数。在新的声明方式中，形式参数的类型是显式声明的，并且是函数类型的一部分，这种声明称为函数原型。在旧的方式中，不指定形式参数类型。有关函数声明的讨论，参见 A.8.6 节和 A.10.1 节。

在函数调用的作用域中，如果函数是以旧式方式声明的，则按以下方式对每个实际参数进行默认参数提升：对每个整型参数进行整型提升（参见 A.6.l 节）；将每个 `float` 类型的参数转换为 `double` 类型。如果调用时实际参数的数目与函数定义中形式参数的数目不等，或者某个实际参数的类型提升后与相应的形式参数类型不一致，则函数调用的结果是未定义的。类型一致性依赖于函数是以新式方式定义的还是以旧式方式定义的。如果是旧式的定义，则比较经提升后函数调用中的实际参数类型和提升后的形式参数类型；如果是新式的定义，则提升后的实际参数类型必须与没有提升的形式参数自身的类型保持一致。

在函数调用的作用域中，如果函数是以新式方式声明的，则实际参数将被转换为函数原型中的相应形式参数类型，这个过程类似于赋值。实际参数数目必须与显式声明的形式参数数目相同，除非函数声明的形式参数表以省略号（`, ...`）结尾。在这种情况下，实际参数的数目必须等于或超过形式参数的数目；对于尾部没有显式指定类型的形式参数，相应的实际参数要进行默认的参数提升，提升方法同前面所述。如果函数是以旧式方式定义的，那么，函数原型中每个形式参数的类型必须与函数定义中相应的形式参数类型一致（函数定义中的形式参数类型经过参数提升后）。

说明：这些规则非常复杂，因为必须要考虑新旧式函数的混合使用。应尽可能避免新旧式函数声明混合使用。

实际参数的求值次序没有指定。不同编译器的实现方式各不相同。但是，在进入函数前，实际参数和函数标志符是完全求值的，包括所有的副作用。对任何函数都允许进行递归调用。

3 结构引用

后缀表达式后跟一个圆点和一个标识符仍是后缀表达式。第一个操作数表达式的类型必须是结构或联合，标识符必须是结构或联合的成员的名字。结果值是结构或联合中命名的成员，其类型是对应成员的类型。如果第一个表达式是一个左值，并且第二个表达式的类型不是数组类型，则整个表达式是一个左值。

后缀表达式后跟一个箭头（由`-`和`>`组成）和一个标识符仍是后缀表达式。第一个操作数表达式必须是一个指向结构或联合的指针，标识符必须是结构或联合的成员的名字。结果指向指针表达式指向的结构或联合中命名的成员，结果类型是对应成员的类型。如果该类型不是数组类型，则结果是一个左值。

因此，表达式 `E1->MOS` 与`(*E1).MOS` 等价。结构和联合将在 A.8.3 节中讨论。

说明：在本书的第 1 版中，规定了这种表达式中成员的名字必须属于后缀表达式指定的结构或联合，但是，该规则并没有强制执行。最新的编译器和 ANSI 标准强制执行了这一规则。

4 后缀自增运算符与后缀自减运算符

后缀表达式后跟一个`++`或`--`运算符仍是一个后缀表达式。表达式的值就是操作数的值。执行完该表达式后，操作数的值将加 1（`++`）或减 1（`--`）。操作数必须是一个左值。有关操作数的限制和运算细节的详细信息，参见加法类运算符（A.7.7 节）和赋值类运算符（A.7.17 节）中的讨论。其结果不是左值。

A.7.4 一元运算符

带一元运算符的表达式遵循从右到左的结合性。

一元表达式:

后缀表达式

++一元表达式

--一元表达式

一元运算符 强制类型转换表达式

sizeoff 一元表达式

sizeof(类型名)

一元运算符: one of

& * + - ~ !

1 前缀自增运算符与前缀自减运算符

在一元表达式的前面添加运算符++或--后得到的表达式是一个一元表达式。操作数将被加 1（++）或减 1（--），表达式的值是经过加 1、减 1 以后的值。操作数必须是一个左值。有关操作数的限制和运算细节的详细信息，参见加法类运算符（参见 A.7.7 节）和赋值类运算符（参见 A.7.17 节）。其结果不是左值。

2 地址运算符

一元运算符&用于取操作数的地址。该操作数必须是一个左值（不指向位字段、不指向声明为 register 类型的对象），或者为函数类型。结果值是一个指针，指向左值指向的对象或函数。如果操作数的类型为 *T*，则结果的类型为指向 *T* 类型的指针。

3 间接寻址运算符

一元运算符*表示间接寻址，它返回其操作数指向的对象或函数。如果它的操作数是一个指针且指向的对象是算术、结构、联合或指针类型，则它是一个左值。如果表达式的类型为“指向 *T* 类型的指针”，则结果类型为 *T*。

4 一元加运算符

一元运算符+的操作数必须是算术类型，其结果是操作数的值。如果操作数是整型，则将进行整型提升，结果类型是经过提升后的操作数的类型。

说明：一元运算符+是 ANSI 标准新增加的，增加该运算符是为了与一元运算符-对应。

5 一元减运算符

一元运算符-的操作数必须是算术类型，结果为操作数的负值。如果操作数是整型，则将进行整型提升。带符号数的负值的计算方法为：将提升后得到的类型能够表示的最大值减去提升后的操作数的值，然后加 1；但 0 的负值仍为 0。结果类型为提升后的操作数的类型。

6 二进制反码运算符

一元运算符~的操作数必须是整型，结果为操作数的二进制反码。在运算过程中需要对操作数进行整型提升。如果操作数为无符号类型，则结果为提升后的类型能够表示的最大值减去操作数的值而得到的结果值。如果操作数为带符号类型，则结果的计算方式为：将提升后的操作数转换为相应的无符号类型，使用运算符~计算反码，再将结果转换为带符号类型。结果的类型为提升后的操作数的类型。

7 逻辑非运算符

运算符!的操作数必须是算术类型或者指针。如果操作数等于 0，则结果为 1，否则结果为 0。结果类型为 int。

8 sizeof 运算符

sizeof 运算符计算存储与其操作数同类型的对象所需的字节数。操作数可以为一个未求值的表达式，也可以为一个用括号括起来的类型名。将 sizeof 应用于 char 类型时，其结果值为 1；将它应用于数组时，其值为数组中字节的总数。应用于结构或联合时，结果为对象的字节数，包括对象中包含的数组所需要的任何填充空间：有 n 个元素的数组的长度是一个数组元素长度的 n 倍。此运算符不能用于函数类型和不完整类型的操作数，也不能用于位字段。结果是一个无符号整型常量，具体的类型由实现定义。在标准头文件<stddef.h>（参见附录 B）中，这一类型被定义为 size_t 类型。

A.7.5 强制类型转换

以括号括起来的类型名开头的一元表达式将导致表达式的值被转换为指定的类型。

强制类型转换表达式:
    一元表达式
    (类型名)强制类型转换表达式

这种结构称为强制类型转换。类型名将在 A.8.8 节描述。转换的结果已在 A.6 节讨论过。包含强制类型转换的表达式不是左值。

A.7.6 乘法类运算符

乘法类运算符&、/和%遵循从左到右的结合性。

乘法类表达式:
    强制类型转换表达式
    乘法类表达式*强制类型转换表达式
    乘法类表达式/强制类型转换表达式
    乘法类表达式%强制类型转换表达式

运算符*和/的操作数必须为算术类型，运算符&的操作数必须为整型。这些操作数需要进行普通的算术类型转换，结果类型由执行的转换决定。

二元运算符*表示乘法。

二元运算符/用于计算第一个操作数同第二个操作数相除所得的商，而运算符%用于计算两个操作数相除后所得的余数。如果第二个操作数为 0，则结果没有定义。并且，(a/b)*b+a%b 等于 a 永远成立。如果两个操作数均为非负，则余数为非负值且小于除数，否则，仅保证余数的绝对值小于除数的绝对值。

A.7.7 加法类运算符

加法类运算符+和-遵循从左到右的结合性。如果操作数中有算术类型的操作数，则需要进行普通的算术类型转换。每个运算符可能为多种类型。

加法类表达式:
    乘法类表达式
    加法类表达式+乘法类表达式
    加法类表达式-乘法类表达式

运算符+用于计算两个操作数的和。指向数组中某个对象的指针可以和一个任何整型的值相加，后者将通过乘以所指对象的长度被转换为地址偏移量。相加得到的和是一个指针，它与初始指针具有相同的类型，并指向同一数组中的另一个对象，此对象与初始对象之间具有一定的偏移量。因此，如果 P 是一个指向数组中某个对象的指针，则表达式 P+1 是指向数组中下一个对象的指针。如果相加所得的和对应的指针不在数组的范围内，且不是数组末尾元素后的第一个位置，则结果没有定义。

说明：允许指针指向数组末尾元素的下一个元素是 ANSI 中新增加的特征，它使得我们可以按照通常的习惯循环地访问数组元素。

运算符-用于计算两个操作数的差值。可以从某个指针上减去一个任何整型的值，减法运算的转换规则和条件与加法的相同。

如果指向同一类型的两个指针相减，则结果是一个带符号整型数，表示它们指向的对象之间的偏移量。相邻对象间的偏移量为 1。结果的类型同具体的实现有关，但在标准头文件<stddef.h>中定义为 ptrdiff_t。只有当指针指向的对象属于同一数组时，差值才有意义。但是，如果 P 指向数组的最后一个元素，则(P+1)-P 的值为 1。

A.7.8 移位运算符

移位运算符<<和>>遵循从左到右的结合性。每个运算符的各操作数都必须为整型，并且遵循整型提升原则。结果的类型是提升后的左操作数的类型。如果右操作数为负值，或者大于或等于左操作数类型的位数，则结果没有定义。

移位表达式:
    加法类表达式
    移位表达式<<加法类表达式
    移位表达式>>加法类表达式

E1<<E2 的值为 E1（按位模式解释）向左移 E2 位得到的结果。如果不发生溢出，这个结果值等价于 E1 乘以 2^{E2}。E1>>E2 的值为 E1 向右移 E2 位得到的结果。如果 E1 为无符号数或为非负值，则右移等同于 E1 除以 2^{E2}。其它情况下的执行结果由具体实现定义。

A.7.9 关系运算符

关系运算符遵循从左到右的结合性，但这个规则没有什么作用。a<b<c 在语法分析时将

被解释为(a<b)<c，并且 a<b 的结果值只能为 0 或 1。

关系表达式:
    移位表达式
    关系表达式<移位表达式
    关系表达式>移位表达式
    关系表达式<=移位表达式
    关系表达式>=移位表达式

当关系表达式的结果为假时，运算符<（小于）、>（大于）、<=（小于等于）和>=（大于等于）的结果值都为 0；当关系表达式的结果为真时，它们的结果值都为 1。结果的类型为 int 类型。如果操作数为算术类型，则要进行普通的算术类型转换。可以对指向同一类型的对象的指针进行比较（忽略任何限定符），其结果依赖于所指对象在地址空间中的相对位置。指针比较只对相同对象才有意义：如果两个指针指向同一个简单对象，则相等；如果指针指向同一个结构的不同成员，则指向结构中后声明的成员的指针较大；如果指针指向同一个联合的不同成员，则相等；如果指针指向一个数组的不同成员，则它们之间的比较等价于对应下标之间的比较。如果指针 P 指向数组的最后一个成员，尽管 P+1 已指向数组的界外，但 P+1 仍比 P 大。其它情况下指针的比较没有定义。

说明：这些规则允许指向同一个结构或联合的不同成员的指针之间进行比较，与第 1 版比较起来放宽了一些限制。这些规则还使得与超出数组末尾的第一个指针进行比较合法化。

A.7.10 相等类运算符

相等类表达式:
    关系表达式
    相等类表达式==关系表达式
    相等类表达式!=关系表达式

运算符==（等于）和!=（不等于）与关系运算符相似，但它们的优先级更低。（只要 a<b 与 c<d 具有相同的真值，则 a<b==c<d 的值总为 1。）

相等类运算符与关系运算符具有相同的规则，但这类运算符还允许执行下列比较：指针可以与值为 0 的常量整型表达式或指向 void 的指针进行比较。参见 A.6.6 节。

A.7.11 按位与运算符

按位与表达式:
    相等类表达式
    按位与表达式&相等类表达式

执行按位与运算时要进行普通的算术类型转换。结果为操作数经按位与运算后得到的值。该运算符仅适用于整型操作数。

A.7.12 按位异或运算符

按位异或表达式:
    按位与表达式
    按位异或表达式^按位与表达式

执行按位异或运算时要进行普通的算术类型转换，结果为操作数经按位异或运算后得到的值。该运算符仅适用于整型操作数。

A.7.13 按位或运算符

按位或表达式:
    按位异或表达式
    按位或表达式|按位异或表达式

执行按位或运算时要进行常规的算术类型转换，结果为操作数经按位或运算后得到的值。该运算符仅适用于整型操作数。

A.7.14 逻辑与运算符

逻辑与表达式:
    按位或表达式
    逻辑与表达式&&按位或表达式

运算符&&遵循从左到右的结合性。如果两个操作数都不等于 0，则结果值为 1，否则结果值 0。与运算符&不同的是，&&确保从左到右的求值次序：首先计算第一个操作数，包括所有可能的副作用；如果为 0，则整个表达式的值为 0；否则，计算右操作数，如果为 0，则整个表达式的值为 0，否则为 1。

两个操作数不需要为同一类型，但是，每个操作数必须为算术类型或者指针。其结果为 `int` 类型。

A.7.15 逻辑或运算符

逻辑或表达式:
    逻辑与表达式
    逻辑或表达式||逻辑与表达式

运算符||遵循从左到右的结合性。如果该运算符的某个操作数不为 0，则结果值为 1；否则结果值为 0。与运算符|不同的是，||确保从左到右的求值次序：首先计算第一个操作数，包括所有可能的副作用；如果不为 0，则整个表达式的值为 1；否则，计算右操作数，如果不为 0，则整个表达式的值为 1；否则结果为 0。

两个操作数不需要为同一类型，但是每个操作数必须为算术类型或者指针。其结果为 `int`

类型。

A.7.16 条件运算符

条件表达式:
    逻辑或表达式
    逻辑或表达式?表达式:条件表达式

首先计算第一个表达式（包括所有可能的副作用），如果该表达式的值不等于 0，则结果为第二个表达式的值，否则结果为第三个表达式的值。第二个和第三个操作数中仅有一个操作数会被计算。如果第二个和第三个操作数为算术类型，则要进行普通的算术类型转换，以使它们的类型相同，该类型也是结果的类型。如果它们都是 `void` 类型，或者是同一类型的结构或联合，或者是指向同一类型的对象的指针，则结果的类型与这两个操作数的类型相同。如果其中一个操作数是指针，而另一个是常量 0，则 0 将被转换为指针类型，且结果为指针类型。如果一个操作数为指向 `void` 的指针，而另一个操作数为指向其它类型的指针，则指向其它类型的指针将被转换为指向 `void` 的指针，这也是结果的类型。

在比较指针的类型时，指针所指对象的类型的任何类型限定符（参见 A.8.2 节）都将被忽略，但结果类型会继承条件的各分支的限定符。

A.7.17 赋值表达式

赋值运算符有多个，它们都是从左到右结合。

赋值表达式:
    条件表达式
    一元表达式 赋值运算符 赋值表达式
赋值运算符: one of
`= *= /= %= += -= <<= >>= &= ^= !=`

所有这些运算符都要求左操作数为左值，且该左值是可以修改的：它不可以是数组、不完整类型或函数。同时其类型不能包括 `const` 限定符；如果它是结构或联合，则它的任意一个成员或递归子成员不能包括 `const` 限定符。赋值表达式的类型是其左操作数的类型，值是赋值操作执行后存储在左操作数中的值。

在使用运算符=的简单赋值中，表达式的值将替换左值所指向的对象的值。下面几个条件中必须有一个条件成立：两个操作数均为算术类型，在此情况下右操作数的类型通过赋值转换为左操作数的类型；两个操作数为同一类型的结构或联合；一个操作数是指针，另一个操作数是指向 `void` 的指针；左操作数是指针，右操作数是值为 0 的常量表达式；两个操作数都是指向同一类型的函数或对象的指针，但右操作数可以没有 `const` 或 `volatile` 限定符。

形式为 `E1 op= E2` 的表达式等价于 `E1 = E1 op (E2)`，惟一的区别是前者对 `E1` 仅求值一次。

A.7.18 逗号运算符

表达式:
    赋值表达式
    表达式，赋值表达式

由逗号分隔的两个表达式的求值次序为从左到右，并且左表达式的值被丢弃。右操作数的类型和值就是结果的类型和值。在开始计算右操作数以前，将完成左操作数涉及到的副作用的计算。在逗号有特别含义的上下文中，如在函数参数表（参见 A.7.3 节）和初值列表（A.8.7 节）中，需要使用赋值表达式作为语法单元，这样，逗号运算符仅出现在圆括号中。例如，下列函数调用：

```
f(a, (t=3, t+2), c)
```

包含 3 个参数，其中第二个参数的值为 5。

A.7.19 常量表达式

从语法上看，常量表达式是限定于运算符的某一个子集的表达式：

常量表达式:
    条件表达式

某些上下文要求表达式的值为常量，例如，`switch` 语句中 `case` 后面的数值、数组边界和位字段的长度、枚举常量的值、初值以及某些预处理器表达式。

除了作为 `sizeof` 的操作数之外，常量表达式中可以不包含赋值、自增或自减运算符、函数调用或逗号运算符。如果要求常量表达式为整型，则它的操作数必须由整型、枚举、字符和浮点常量组成；强制类型转换必须指定为整型，任何浮点常量都将被强制转换为整型。此规则对数组、间接访问、取地址运算符和结构成员操作不适用。（但是，`sizeof` 可以带任何类型的操作数。）

初值中的常量表达式允许更大的范围：操作数可以是任意类型的常量，一元运算符&可以作用于外部、静态对象以及以常量表达式为下标的外部或静态数组。对于无下标的数组或函数的情况，一元运算符&将被隐式地应用。初值计算的结果值必须为下列二者之一：一个常量；前面声明的外部或静态对象的地址加上或减去一个常量。

允许出现在`#if` 后面的整型常量表达式的范围较小，它不允许 `sizeof` 表达式、枚举常量和强制类型转换。详细信息参见 A.12.5 节。

A.8 声明

声明（declaration）用于说明每个标识符的含义，而并不需要为每个标识符预留存储空间。预留存储空间的声明称为定义（definition）。声明的形式如下：

声明

声明说明符 初始化声明符表 opt;

初始化声明符表中的声明符包含被声明的标识符；声明说明符由一系列的类型和存储类说明符组成。

声明说明符：

存储类说明符 声明说明符 opt

类型说明符 声明说明符 opt

类型限定符 声明说明符 opt

初姑化声明符表：

初始化声明符

初姑化声明符袁，初始化声明符

初始化声明符：

声明符

声明符 = 初值

声明符将在稍后部分讨论（参见 A.8.5 节）。声明符包含了被声明的名字。一个声明中必须至少包含一个声明符，或者其类型说明符必须声明一个结构标记、一个联合标记或枚举的成员。不允许空声明。

A.8.1 存储类说明符

存储类说明符如下所示：

存储类说明符：

auto

register

static

extern

typedef

有关存储类的意义，我们已在 A.4 节中讨论过。

说明符 auto 和 register 将声明的对象说明为自动存储类对象，这些对象仅可用在函数中。这种声明也具有定义的作用，并将预留存储空间。带有 register 说明符的声明等价于带有 auto 说明符的声明，所不同的是，前者暗示了声明的对象将被频繁地访问。只有很少的对象被真正存放在寄存器中，并且只有特定类型才可以。该限制同具体的实现有关。但是，如果一个对象被声明为 register，则将不能对它应用一元运算符&（显式应用或隐式应用都不允许）。

说明：对声明为 register 但实际按照 auto 类型处理的对象的地址进行计算是非法的。这是一个新增加的规则。

说明符 static 将声明的对象说明为静态存储类。这种对象可以用在函数内部或函数外部。在函数内部，该说明符将引起存储空间的分配，具有定义的作用。有关该说明符在函数外部的作用参见 A.11.2 节。

函数内部的 extern 声明表明，被声明的对象的存储空间定义在其它地方。有关该说明符在函数外部的作用参见 A.11.2 节。

typedef 说明符并不会为对象预留存储空间。之所以将它称为存储类说明符，是为了语法描述上的方便。我们将在 A.8.9 节中讨论它。

一个声明中最多只能有一个存储类说明符。如果没有指定存储类说明符，则将按照下列规则进行：在函数内部声明的对象被认为是 auto 类型；在函数内部声明的函数被认为是 extern 类型；在函数外部声明的对象与函数将被认为是 static 类型，且具有外部连接。详细信息参见 A.10 节和 A.11 节。

A.8.2 类型说明符

类型说明符的定义如下：

类型说明符：

void

char

short

int

long

float

double

signed

unsigned

结构或联合说明符

枚举说明符

类型定义名

其中，long 和 short 这两个类型说明符中最多有一个可同时与 int 一起使用，并且，在这种情况下省略关键字 int 的含义也是一样的。long 可与 double 一起使用。signed 和 unsigned 这两个类型说明符中最多有一个可同时与 int、int 的 short 或 long 形式、char 一起使用。signed 和 unsigned 可以单独使用，这种情况下默认为 int 类型。signed 说明符对于强制 char 对象带符号位是非常有用的；其它整型也允许带 signed 声明，但这是

多余的。

除了上面这些情况之外，在一个声明中最多只能使用一个类型说明符。如果声明中没有类型说明符，则默认为 int 类型。

类型也可以用限定符限定，以指定被声明对象的特殊属性。

类型限定符：

const

volatile

类型限定符可与任何类型说明符一起使用。可以对 const 对象进行初始化，但在初始化以后不能进行赋值。volatile 对象没有与实现无关的语义。

说明：const 和 volatile 属性是 ANSI 标准新增加的特性。const 用于声明可以存放在只读存储器中的对象，并可能提高优化的可能性。volatile 用于强制某个实现屏蔽可能的优化。倒如，对于具有内存映像输入 / 输出的机器，指向设备寄存器的指针可以声明为指向 volatile 的指针，目的是防止编译器通过指针删除明显多余的引用。除了诊断显式尝试修改 const 对象的情况外，编译器可能会忽略这些限定符。

A.8.3 结构和联合声明

结构是由不同类型的命名成员序列组成的对象。联合也是对象，在不同时刻，它包含多个不同类型成员中的任意一个成员。结构和联合说明符具有相同的形式。

结构或联合说明符：

结构或联合标识符 opt {结构声明表}

结构或联合 标识符

结构或联合：

struct

union

结构声明表是对结构或联合的成员进行声明的声明序列：

结构声明表

结构声明

结构声明表 结构声明

结构声明：

说明符限定符表 结构声明符表

说明符限定符表：

类型说明符 说明符限定符表 opt

类型限定符 说明符限定符表 opt

结构声明符表：

结构声明符

结构声明符表，结构声明符

通常，结构声明符就是结构或联合成员的声明符。结构成员也可能由指定数目的比特位组成，这种成员称为位字段，或仅称为字段，其长度由跟在声明符冒号之后的常量表达式指定。

结构声明符：

声明符

声明符 opt：常量表达式

下列形式的类型说明符将其中的标识符声明为结构声明表指定的结构或联合的标记：

结构或联合标识符 {结构声明表}

在同一作用域或内层作用域中的后续声明中，可以在说明符中使用标记（而不使用结构声明表）来引用同一类型，如下所示：

结构或联合 标识符

如果说明符中只有标记而无结构声明表，并且标记没有声明，则认为其为不完整类型。具有不完整结构或联合类型的对象可在不需要对象大小的上下文中引用，比如，在声明中（不是定义中），它可用于说明一个指针或创建一个 typedef 类型，其余情况则不允许。在引用之后，如果具有该标记的说明符再次出现并包含一个声明表，则该类型成为完整类型。即使是在包含结构声明表的说明符中，在该结构声明表内声明的结构或联合类型也是不完整的，一直到花括号“}”终止该说明符时，声明的类型才成为完整类型。

结构中不能包含不完整类型的成员。因此，不能声明包含自身实例的结构或联合。但是，除了可以命名结构或联合类型外，标记还可以用来定义自引用结构。由于可以声明指向不完整类型的指针，所以，结构或联合可包含指向自身实例的指针。

下列形式的声明适用一个非常特殊的规则：

结构或联合 标识符，

这种形式的声明声明了一个结构或联合，但它没有声明表和声明符。即使该标识符是外层作用域中已声明过的结构标记或联合的标记（参见 A.11.1 节），该声明仍将使该标识符成为当前作用域内一个新的不完整类型的结构标记或联合的标记。

说明：这是 ANSI 中一个新的比较难理解的规则。它旨在处理内层作用域中声明的相互递归调用的结构，但这些结构的标记可能已在外层作用域中声明。

具有结构声明表而无标记的结构说明符或联合说明符用于创建一个惟一的类型，它只能被它所在的声明直接引用。

成员和标记的名字不会相互冲突，也不会与普通变量冲突。一个成员名字不能在同一结

构或联合中出现两次，但相同的成员名字可用在不同的结构或联合中。

说明：在本书的第 1 版中，结构或联合的成员名与其父辈无关联。但是，在 ANSI 标准制定前，这种关联在编译器中普遍存在。

除字段类型的成员外，结构成员或联合成员可以为任意对象类型。字段成员（它不需要声明符，因此可以不命名）的类型为 int、unsigned int 或 signed int，并被解释为指定长度（用二进制位表示）的整型对象，int 类型的字段是否看作为有符号数同具体的实现有关。结构的相邻字段成员以某种力式（同具体的实现有关）存放在某些存储单元中（同具体的实现有关）。如果某字段之后的另一字段无法全部存入已被前面的字段部分占用的存储单元中，则它可能会被分割存放到多个存储单元中，或者是，存储单元中的剩余部分也可能被填充。我们可以用宽度为 0 的无名字段来强制进行这种填充，从而使得下一字段从下一分配单元的边界开始存储。

说明：在字段处理方面，ANSI 标准比第 1 版更依赖于具体的实现。如果要按照与实现相关的方式存储字段，建议阅读一下该语言规则。作为一种可移植的方法，带字段的结构可用来节省存储空间（代价是增加了指令空间和访问字段的时间），同时，它还可以用来在位层次上描述存储布局，但该方法不可移植，在这种情况下，必须了解本地实现的一些规则。

结构成员的地址值按它们声明的顺序递增。非字段类型的结构成员根据其类型在地址边界上对齐，因此，在结构中可能存在无名空穴。若指向某一结构的指针被强制转换为指向该结构第一个成员的指针类型，则结果将指向该结构的第一个成员。

联合可以被看作为结构，其所有成员起始偏移量都为 0，并且其大小足以容纳任何成员。任一时刻它最多只能存储其中的一个成员。如果指向某一联合的指针被强制转换为指向一个成员的指针类型，则结果将指向该成员。

如下所示是结构声明的一个简单例子：

```
struct tnode {
   char tword[20];
   int count;
   struct tnode *left;
   struct tnode *right;
}
```

该结构包含一个具有 20 个字符的数组、一个整数以及两个指向类似结构的指针。在给出这样的声明后，下列说明：

```
struct tnode s, *sp;
```

将把 s 声明为给定类型的结构，把 sp 声明为指向给定类型的结构的指针。在这些声明的基础上，表达式

```
sp->count
```

引用 sp 指向的结构的 count 字段，而

```
s.left
```

则引用结构的左子树指针，表达式

```
s.right->tword[0]
```

引用 s 右子树中 tword 成员的第一个字符。

通常情况下，我们无法检查联合的某一成员，除非已用该成员给联合赋值。但是，有一个特殊的情况可以简化联合的使用：如果一个联合包含共享一个公共初始序列的多个结构，并且该联合当前包含这些结构中的某一个，则允许引用这些结构中任一结构的公共初始部分。例如，下面这段程序是合法的：

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

A.8.4 枚举

枚举类型是一种特殊的类型，它的值包含在一个命名的常量集合中。这些常量称为枚举符。枚举说明符的形式借鉴了结构说明符和联合说明符的形式。

枚举说明符：

enum 标识符 opt {枚举符表}

enum 标识符

枚举符表：

枚举符

枚举符表, 枚举符

枚举符：

标识符

标识符 = 常量表达式

枚举符表中的标识符声明为 int 类型的常量，它们可以用在常量可以出现的任何地方。如果其中不包括带有=的枚举符，则相应常量值从 0 开始，且枚举常量值从左至右依次递增 1。如果其中包括带有=的枚举符，则该枚举符的值由该表达式指定，其后的标识符的值从该值开始依次递增。

同一作用域中的各枚举符的名字必须互不相同，也不能与普通变量名相同，但其值可以相同。

枚举说明符中标识符的作用与结构说明符中结构标记的作用类似，它命名了一个特定的枚举类型。除了不存在不完整枚举类型之外，枚举说明符在有无标记、有无枚举符表的情况下的规则与结构或联合中相应的规则相同。无枚举符表的枚举说明符的标记必须指向作用域中另一个具有枚举符表的说明符。

说明：相对于本书第 1 版，枚举类型是一个新概念，但它作为 C 语言的一部分已有好多年了。

A.8.5 声明符

声明符的语法如下所示

声明符：

指针 opt 直接声明符

直接声明符

标识符

(声明符)

直接声明符 [常量表达式 opt]

直接声明符(形式参数类型表)

直接声明符(标识表 opt)

指针：

* 类型限定符表 opt

* 类型限定符表 opt 指针

类型限定符表

类型限定符

类型限定符表 类型限定符

声明符的结构与间接指针、函数及数组表达式的结构类似，结合性也相同。

A.8.6 声明符的含义

声明符表出现在类型说明符和存储类说明符序列之后。每个声明符声明一个睢一的主标识符，该标识符是直接声明符产生式的第一个候选式。存储类说明符可直接作用于该标识符，但其类型由声明符的形式决定。当声明符的标识符出现在与该声明符形式相同的表达式中时，该声明符将被作为一个断言，其结果将产生一个指定类型的对象。

如果只考虑声明说明符（参见 A.8.2 节）的类型部分及特定的声明符，则声明可以表示为“T D”的形式，其中 T 代表类型，D 代表声明符。在不同形式的声明中，标识符的类型可用这种形式来表述。

在声明 T D 中，如果 D 是一个不加任何限定的标识符，则该标识符的类型为 T。

在声明 T D 中，如果 D 的形式为：

(D1)

则 D1 中标识符的类型与 D 的类型相同。圆括号不改变类型，但可改变复杂声明符之间的结合。

1，指针声明符

在声明 T D 中，如果 D 具有下列形式：

* 类型限定符表 D1

且声明 T D1 中的标识符的类型为“类型修饰符 T”，则 D 中标识符的类型为“类型修饰符 类型限定符表指向 T 的指针”。星号*后的限定符作用于指针本身，而不是作用于指针指向的对象。

例如。考虑下列声明：

```
int *ap[];
```

其中，ap[]的作用等价于 D1，声明“int ap[]”将把 ap 的类型声明为“int 类型的数组”，类型限定符表为空，且类型修饰符为“……的数组”。因此，该声明实际上将把 ap 声明为“指向 int 类型的指针数组”类型。

我们来看另外一个例子。下列声明：

```
int i, *pi, *const cpi = &i;
```

```
const int ci = 3, *pci;
```

声明了一个整型 i 和一个指向整型的指针 pi。不能修改常量指针 cpi 的值，该指针总是指向同一位置，但它所指之处的值可以改变。整型 ci 是常量，也不能修改（可以进行初始化，如本例中所示）。pci 的类型是“指向 const int 的指针”，pci 本身可以被修改以指向另一个地方，但它所指之处的值不能通过 pci 赋值来改变，

2 数组声明符

在声明 T D 中，如果 D 具有下列形式：

D1[常量表达式 opt]

且声明 T D1 中标识符的类型是“类型修饰符 T”，则 D 的标识符类型为“类型修饰符 T 类型的数组”。如果存在常量表达式，则该常量表达式必须为整型且值大于 0。若缺少指定数组上界的常量表达式，则该数组类型是不完整类型。

数组可以由算术类型、指针类型、结构类型或联合类型构造而成，也可以由另一个数组构造而成（生成多维数组）。构造数组的类型必须是完整类型，绝对不能是不完整类型的数组或结构。也就是说，对于多维数组来说，只有第一维可以缺省。对于不完整数组类型的对象来说，其类型可以通过对该对象进行另一个完整声明（参见 A.10.2 节）或初始化（参见 A.8.7 节）来使其完整。例如：

```
float fa[17], *afp[17];
```

声明了一个浮点数数组和一个指向浮点数的指针数组，而

```
static int x3d[3][5][7];
```

则声明了一个静态的三维整型数组，其大小为 3×5×7。具体来说，x3d 是一个由 3 个项组成的数组，每个项都是由 5 个数组组成的一个数组，5 个数组中的每个数组又都是由 7 个整型数组成的数组。x3d、x3d[i]、x3d[i][j]与 x3d[i][j][k]都可以合法地出现在一个表达式中。前三者是数组类型，最后一个是 int 类型。更准确地说，x3d[i][j]是一个有 7 个整型元素的数组；x3d[i]则是有 5 个元素的数组，而其中的每个元素又是一个具有 7 个整型元素的数组。

根据数组下标运算的定义，E1[E2]等价于*(E1+E2)。因此，尽管表达式的形式看上去不对称，但下标运算是可交换的运算。根据适用于运算符+和数组的转换规则（参见 A.6.6 节、A.7.1 节与 A.7.7 节），若 E1 是数组且 E2 是整数，则 E1[E2]代表 E1 的第 E2 个成员。

在本例中，x3d[i][j][k]等价于*(x3d[i][j]+k)。第一个子表达式 x3d[i][j]将按照 A.7.1 节中的规则转换为“指向整型数组的指针”类型，而根据 A.7.7 节中的规则，这里的加法运算需要乘以整型类型的长度。它遵循下列规则：数组按行存储（最后一维下标变动最快），且声明中的第一维下标决定数组所需的存储区大小，但第一维下标在下标计算时无其它作用。

3 函数声明符

在新式的函数声明 T D 中，如果 D 具有下列形式：

D1(形式参数类型表)

并且，声明 `T D1` 中标识符的类型为“类型修饰符 `T`”，则 `D` 的标识符类型是“返回 `T` 类型值且具有‘形式参数类型表’中的参数的‘类型修饰符’类型的函数”。

形式参数的语法定义为：

形式参数类型表：

形式参数表

形式参数表, ...

形式参数表：

形式参数声明

形式参数表, 形式参数声明

形式参数声明：

声明说明符 声明符

声明说明符 抽象声明符 `opt`

在这种新式的声明中，形式参数表指定了形式参数的类型。这里有一个特例，按照新式方式声明的无形式参数函数的声明符也有一个形式参数表，该表仅包含关键字 `void`。如果形式参数表以省略号“`, ...`”结尾，则该函数可接受的实际参数个数比显式说明的形式参数个数要多。详细信息参见 A.7.3 节。

如果形式参数类型是数组或函数，按照参数转换规则（参见 A.10.1 节），它们将被转换为指针。形式参数的声明中惟一允许的存储类说明符是 `register`，并且，除非函数定义的开头包括函数声明符，否则该存储类说明符将被忽略。类似地，如果形式参数声明中的声明符包含标识符，且函数定义的开头没有函数声明符，则该标识符超出了作用域。不涉及标识符的抽象声明符将在 A.8.8 节中讨论。

在旧式的函数声明 `T D` 中，如果 `D` 具有下列形式：

`D1(`标识符表 `opt)`

并且声明 `D1` 中的标识符的类型是“类型修饰符 `T`”，则 `D` 的标识符类型为“返回 `T` 类型值且未指定参数的‘类型修饰符’类型的函数”。形式参数（如果有的话）的形式如下：

标识符表：

标识符

标识符表, 标识符

在旧式的声明符中，除非在函数定义的前面使用了声明符，否则，标识符表必须空缺（参见 A.10.1 节）。声明不提供有关形式参数类型的信息。

例如，下列声明：

```
int f(), *fpi(), (*pfi)();
```

声明了一个返回整型值的函数 `f`、一个返回指向整型的指针的函数 `fpi` 以及一个指向返回整

型的函数的指针 pfi。它们都没有说明形式参数类型，因此都属于旧式的声明。

在下列新式的声明中：

```
int strcpy(char *dest, const char *source), rand(void);
```

strcpy 是一个返回 int 类型的函数，它有两个实际参数，第一个实际参数是一个字符指针，第一个实际参数星一个指向常量字符的指针。其中的形式参数名字可以起到注释说明的作用。第二个函数 rand 不带参数，且返回类型为 int。

说明：到目前为止，带形式参数原型的函数声明符是 ANSI 标准中引入的最重要的一个语言变化。它们优于第 1 版中的“旧式”声明符，因为它们提供了函数调用时的错误检查和参数强制转换，但引入的同时也带来了很多混乱和麻烦，而且还必须兼容这两种形式。为了保持兼容，就不得不在语法上进行一些处理，即采用 void 作为新式的无形式参数函数的显式标记。

采用省略号“, ...”表示函数变长参数表的做法也是 ANSI 标准中新引入的，并且，结合标准头文件<stdarg.h>中的一些宏，共同将这个机制正式化了。该机制在第 1 版中是官方上禁止的，但可非正式地使用。

这些表示法起源于 C++。

A.8.7 初始化

声明对象时，对象的初始化声明符可为其指定一个初始值。初值紧跟在运算符=之后，它可以是一个表达式，也可以是嵌套在花括号中的初值序列。初值序列可以以逗号结束，这样可以使格式简洁美观。

初值：

赋值表达式

{初值表}

{初值表,}

初值表：

初值

初值表, 初值

对静态对象或数组而言，初值中的所有表达式必须是 A.7.19 节中描述的常量表达式。如果初值是用花括号括起来的初值表，则对 auto 或 register 类型的对象或数组来说，初值中的表达式也同样必须是常量表达式。但是，如果自动对象的初值是一个单个的表达式，则它不必是常量表达式，但必须符合对象赋值的类型要求。

说明：第 1 版不支持自动结构、联合或数组的初始化。而 ANSI 标准是允许的，但只能通过常量结构进行初始化，除非初值可以通过简单表达式表示出来。

未显式初始化的静态对象将被隐式初始化，其效果等同于它（或它的成员）被赋以常量 0。未显式初始化的自动对象的初始值没有定义。

指针或算术类型对象的初值是一个单个的表达式，也可能括在花括号中。该表达式将赋值给对象。

结构的初值可以是类型相同的表达式，也可以是括在花括号中的按其成员次序排列的初值表。无名的位字段成员将被忽略，因此不被初始化。如果表中初值的数目比结构的成员数少，则后面余下的结构成员将被初始化为 0。初值的数目不能比成员数多。

数组的初值是一个括在花括号中的、由数组成员的初值构成的表。如果数组大小未知，则初值的数目将决定数组的大小，从而使数组类型成为完整类型。若数组大小固定，则初值的数目不能超过数组成员的数目。如果初值的数目比数组成员的数目少，则尾部余下的数组成员将被初始化为 0。

这里有一个特例：字符数组可用字符串字面值初始化。字符串中的各个字符依次初始化数组中的相应成员。类似地，宽字符字面值（参见 A.2.6 节）可以初始化 wchar_t 类型的数组。若数组大小未知，则数组大小将由字符串中字符的数目（包括尾部的空字符）决定。若数组大小固定，则字符串中的字符数（不计尾部的空字符）不能超过数组的大小。

联合的初值可以是类型相同的单个表达式，也可以是括在花括号中的联合的第一个成员的初值。

说明：第 1 版不允许对联合进行和始化。“第一个成员”规则并不很完美，但在没有新语法的情况下很难对它进行一般化。除了至少允许以一种简单方式对联合进行显式初始化外，ANSI 规则还给出了非显式初始化的静态联合的精确语义。

聚集是一个结构或数组。如果一个聚集包含聚集类型的成员，则初始化时将递归使用初始化规则。在下列情况的初始化中可以省略括号：如果聚集的成员也足一个聚集，且该成员的初始化符以左花括号开头，则后续部分中用逗号隔开的初值表将初始化子聚集的成员。初值的数目不允许超过成员的数目。但是，如果子聚集的初值不以左花括号开头，则只从初值表中取出足够数目的元素作为于聚集的成员，其它剩余的成员将用来初始化该子聚集所在的聚集的下一个成员。

例如：

```
int x[] = { 1, 3, 5 };
```

将 x 声明并初始化为一个具有 3 个成员的一维数组，这是因为，数组未指定大小且有 3 个初值。下面的例子：

```
float y[4][3] = {
   { 1, 3, 5 },
   { 2, 4, 6 },
   { 3, 5, 7 },
};
```

是一个完全用花括号分隔的初始化：1、3和5这3个数初始化数组 y[0]的第一行，即 y[0][0]、y[0][1]和 y[0][2]。类似地，另两行将初始化 y[1]和 y[2]。因为初值的数目不够，所以 y[3]中的元素将被初始化为 0，完全相同的效果还可以通过下列声明获得：

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

y 的初值以左花括号开始，但 y[0]的初值则没有以左花括号开始，因此 y[0]的初始化将使用表中的 3 个元素。同理，y[1]将使用后续的 3 个元素进行初始化，y[2]依此类推。另外，下列声明：

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

将初始化 y 的第一列（将 y 看成为一个二维数组），其余的元素将默认初始化为 0。

最后

```
char msg[] = "Syntax error on line %s\n";
```

声明了一个字符数组，并用一个字符串字面值初始化该字符数组的元素。该数组的大小包括尾部的空字符。

A.8.8 类型名

在某些上下文中（例如，需要显式进行强制类型转换、需要在函数声明符中声明形式参数类型、作为 sizeof 的实际参数等），我们需要提供数据类型的名字。使用类型名可以解决这个问题，从语法上讲，也就是对某种类型的对象进行声明，只是省略了对象的名字而已。

类型名：

说明符限定符表 抽象声明符 opt

抽象声明符：

指针

指针 opt 直接抽象声明符

直接抽象声明符

(抽象声明符)

直接抽象声明符 opt [常量表达式 opt]

直接抽象声明符 opt (形式参数类型表 opt)

如果该结构是声明中的一个声明符，就有可能惟一确定标识符在抽象声明符中的位置。命名的类型将与假设标识符的类型相同。例如：

```
int
int *
```

```
int *[3]

int (*)[]

int *()

int (*[])(void)
```

其中的 6 个声明分别命名了下列类型：“整型”、“指向整型的指针”、“包含 3 个指向整型的指针的数组”、“指向未指定元素个数的整型数组的指针”、“未指定参数、返回指向整型的指针的函数”、“一个数组，其长度未指定，数组的元素为指向函数的指针，该函数没有参数且返回一个整型值”。

A.8.9 typedef

存储类说明符为 typedef 的声明不用于声明对象，而是定义为类型命名的标识符。这些标识符称为类型定义名。

类型定义名：

标识符

typedef 声明按照普通的声明方式将个类型指派给其声明符中的每个名字（参见 A.8.6 节）。此后，类型定义名在语法上就等价于相关类型的类型说明符关键字。

例如，在定义

```
typedef long Blockno, *Blockptr;

typedef struct { double r, theta; } Complex;
```

之后，下述形式：

```
Blockno b;

extern Blockptr bp;

Complex z, *zp;
```

都是合法的声明。b 的类型为 long，bp 的类型为“指向 long 类型的指针”。z 的类型为指定的结构类型，zp 的类型为指向该结构的指针。

typedef 类型定义并没有引入新的类型，它只是定义了数据类型的同义词，这样，就可以通过另一种方式进行类型声明。在本侧中，b 与其它任何 long 类型对象的类型相同。

类型定义名可在内层作用域中重新声明，但必须给出一个非空的类型说明符集合。例如，下列声明：

```
extern Blockno;
```

并没有重新声明 Blockno，但下列声明：

```
extern int Blockno;
```

则重新声明了 Blockno。

A.8.10 类型等价

如果两个类型说明符表包含相同的类型说明符集合（需要考虑类型说明符之间的蕴涵关系，例如，单独的 long 蕴含了 long int），则这两个类型说明符表是等价的。具有不同标记的结构、不同标记的联合和不同标记的枚举是不等价的，无标记的联合、无标记的结构或无标记的枚举指定的类型也是不等价的。

在展开其中的任何 typedef 类型并删除所有函数形式参数标识符后，如果两个类型的抽象声明符（参见 A.8.8 节）相同，且它们的类型说明符表等价，则这两个类型是相同的。数组长度和函数形式参数类型是其中很重要的因素。

A.9 语句

如果不特别指明，语句都是顺序执行的。语句执行都有一定的结果，但没有值。语句可分为几种类型。

语句：

带标号语句

表达式语句

复合语句

选择语句

循环语句

跳转语句

A.9.1 带标号语句

语句可带有标号前缀。

带标号语句：

标识符：语句

case 常量表达式：语句

default：语句

由标识符构成的标号声明了该标识符。标识符标号的惟一用途就是作为 goto 语句的跳转目标。标识符的作用域是当前函数。因为标号有自己的名字空间，因此不会与其它标识符混淆，并且不能被重新声明。详细信息参见 A.11.1 节。

case 标号和 default 标号用在 switch 语句中（参见 A.9.4 节）。case 标号中的常量

表达式必须为整型。

标号本身不会改变程序的控制流。

A.9.2 表达式语句

大部分语句为表达式语句，其形式如下所示：

表达式语句：

表达式 opt;

大多数表达式语句为赋值语句或函数调用语句。表达式引起的所有副作用在下一条语句执行前结束。没有表达式的语句称为空语句。空语句常常用来为循环语句提供一个空的循环体或设置标号。

A.9.3 复合语句

当需要把若干条语句作为一条语句使用时，可以使用复合语句（也称为“程序块”）。函数定义中的函数体就是一个复合语句。

复合语句：

{声明表 opt 语句表 opt}

声明表：

声明

声明表 声明

语自表

语句

语句表 语句

如果声明表中的标识符位于程序块外的作用域中，则外部声明在程序块内将被挂起（参见 A.11.1 节），在程序块之后再恢复其作用。在同一程序块中，一个标识符只能声明一次。此规则也适用于同一名字空间的标识符（参见 A.11 节），不同名字空间的标识符被认为是不同的。

自动对象的初始化在每次进入程序块的顶端时执行，执行的顺序按照声明的顺序进行。如果执行跳转语句进入程序块，则不进行初始化。Static 类型的对象仅在程序开始执行前初始化一次。

A.9.4 选择语句

选择语句包括下列几种控制流形式：

选择语句：

if (表选式) 语句

if (表达式) 语句 else 语句

switch (表达式) 语句

在两种形式的 if 语句中，表达式（必须为算术类型或指针类型）首先被求值（包括所有的副作用），如果不等于 0，则执行第一个子语句。在第二种形式中，如果表达式为 0，则执行第二个子语句，通过将 else 与同一嵌套层中碰到的最近的未匹配 else 的 if 相连接，可以解决 else 的歧义性问题。

switch 语句根据表达式的不同取值将控制转向相应的分支。关键字 switch 之后用圆括号括起来的表达式必须为整型，此语句控制的子语句一般是复合语句。子语句中的任何语句可带一个或多个 case 标号（参见 A.9.1 节）。控制表达式需要进行整型提升（参见 A.6.1 节），case 常量将被转换为整型提升后的类型。同一 switch 语句中的任何两个 case 常量在转换后不能有相同的值。一个 switch 语句最多可以有一个 default 标号。swltch 语句可以嵌套，case 或 default 标号与包含它的最近的 switch 相关联。

switch 语句执行时，首先计算表达式的值及其副作用，并将其值与每个 case 常量比较，如果某个 case 常量与表达式的值相同，则将控制转向与该 case 标号匹配的语句。如果没有 case 常量与表达式匹配，并且有 default 标号，则将控制转向 default 标号的语句。如果没有 case 常量匹配，且没有 default 标号，则 switch 语句的所有子语句都不执行。

说明：在本书第 1 版中，switch 语句的控制表达式与 case 常量都必须为 int 类型。

A.9.5 循环语句

循环语句用于指定程序段的循环执行。

循环语句

while (表达式) 语句

do 语句 while (表达式);

for (表达式 opt; 表达式 opt; 表达式 opt) 语句

在 while 语句和 do 语句中，只要表达式的值不为 0，其中的子语句将一直重复执行。表达式必须为算术类型或指针类型。while 语句在语句执行前测试表达式，并计算其副作用，而 do 语句在每次循环后测试表达式。

在 for 语句中，第一个表达式只计算一次，用于对循环初始化。该表达式的类型没有限制。第二个表达式必须为算术类型或指针类型，在每次开始循环前计算其值。如果该表达式的值等于 0，则 for 语句终止执行。第三个表达式在每次循环后计算，以重新对循环进行初始化，其类型没有限制。所有表达式的副作用在计算其值后立即结束。如果子语句中没有 continue 语句，则语句

for (表达式 1; 表达式 2; 表达式 3) 语句

等价于

表达式 1;

```
while (表达式 2) {
    语句
    表达式 3;
}
```

for 语句中的 3 个表达式中都可以省略。第二个表达式省略时等价于测试一个非 0 常量。

A.9.6 跳转语句

跳转语句用于无条件地转移控制。

跳转语句：

```
goto 标识符;
continue;
break;
return 表达式 opt;
```

在 goto 语句中，标识符必须是位于当前函数中的标号(参见 A.9.1 节)。控制将转移到标号指定的语句。

continue 语句只能出现在循环语句内，它将控制转向包含此语句的最内层循环部分。更准确地说，在下列任一语句中：

```
while (...) {      do {              for (...) {
  ...               ...               ...
contin: ;          contin: ;         contin: ;
}                 } while (...);     }
```

如果 continuet 语句不包含在更小的循环语句中，则其作用与 goto contin 语句等价。

break 语句只能用在循环语句或 switch 语句中，它将终止包含该语句的最内层循环语句的执行，并将控制转向被终止语句的下一条语句。

return 语句用于将控制从函数返回给调用者。当 return 语句后跟一个表达式时，表达式的值将返回给函数调用者。像通过赋值操作转换类型那样，该表达式将被转换为它所在的函数的返回值类型。

控制到达函数的结尾等价于一个不带表达式的 return 语句。在这两种情况下，返回值都是没有定义的。

A.10 外部声明

提供给 C 编译器处理的输入单元称为翻译单元。它由一个外部声明序列组成，这些外部声明可以是声明，也可以是函数定义。

翻译单元：

外部声明

翻译单元 外部声明

外部声明：

函数定义

声明

与程序块中声明的作用域持续到整个程序块的末尾类似，外部声明的作用域一直持续到其所在的翻译单元的末尾。外部声明除了只能在这一级上给出函数的代码外，其语法规则与其它所有声明相同。

A.10.1 函数定义

函数定义具有下列形式：

函数定义：

声明说明符 opt 声明符 声明表 opt 复合语句

声明说明符中只能使用存储类说明符 extern 或 static。有关这两个存储类说明符之间的区别，参见 A.11.2 节。

函数可返回算术类型、结构、联合、指针或 vold 类型的值，但不能返回函数或数组类型。函数声明中的声明符必须显式指定所声明的标识符具有函数类型，也就是说，必须包含下列两种形式之一（参见 A.8.6 节）：

直接声明符(形式参数类型表)

直接声明符(标识符表 opt)

其中，直接声明符可以为标识符或用圆括号括起来的标识符。特别是，不能通过 typedef 定义函数类型。

第一种形式是一种新式的函数定义，其形式参数及类型都在形式参数类型表中声明，函数声明符后的声明表必须空缺。除了形式参数类型表中只包含 void 类型（表明该函数没有形式参数）的情况外，形式参数类型表中的每个声明符都必须包含一个标识符。如果形式参数类型表以“, ...”结束，则调用该函数时所用的实际参数数目就可以多于形式参数数目。va_arg 宏机制在标准头文件<stdarg.h>中定义，必须使用它来引用额外的参数，我们将在附录 B 中介绍。带有可变形式参数的函数必须至少有一个命名的形式参数。

第二种形式是一种旧式的函数定义：标识符表列出，形式参数的名字，这些形式参数的

类型由声明表指定。对于未声明的形式参数，其类型默认为 int 类型。声明表必须只声明标识符表中指定的形式参数，不允许进行初始化，并且仅可使用存储类说明符 register。

在这两种方式的函数定义中，可以这样理解形式参数：在构成函数体的复合语句的开始处进行声明，并且在该复合语句中不能重复声明相同的标识符（但可以像其它标识符一样在该复合语句的内层程序块中重新声明）。如果某一形式参数声明的类型为“ type 类型的数组”，则该声明将会被自动调整为“指向 type 类型的指针”。类似地，如果某一形式参数声明为“返回 type 类型值的函数”，则该声明将会被调整为“指向返回 type 类型值的函数的指针”。调用函数时，必要时要对实际参数进行类型转换，然后赋值给形式参数，详细信息参见 A.7.3 节。

说明：新式函数定义是 ANSI 标准新引入的一个特征。有关提升的一些细节也有细微的变化。第 1 版指定，float 类型的形式参数声明将被调整为 double 类型。当在函数内部生成一个指向形式参数的指针时，它们之间的区别就显而易见了。

下面是一个新式函数定义的完整例子：

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

其中，int 是声明说明符；max(int a, int b, int c)是函数的声明符；{...}是函数代码的程序块。相应的旧式定义如下所示：

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

其中，int max(a, b, c)是声明符，int a, b, c;是形式参数的声明表。

A.10.2 外部声明

外部声明用于指定对象、函数及其它标识符的特性。术语“外部”表明它们位于函数外部，并且不直接与关键字 extern 连接。外部声明的对象可以不指定存储类，也可指定为 extern 或 static。

同一个标识符的多个外部声明可以共存于同一个翻译单元中，但它们的类型和连接必须保持一致，并且标识符最多只能有一个定义。

如果一个对象或函数的两个声明遵循 A.8.10 节中所述的规则，则认为它们的类型是一致的。并且，如果两个声明之间的区别仅仅在于：其中一个的类型为不完整结构、联合或枚举类型（参见 A.8.3 节），而另一个是对应的带同一标记的完整类型，则认为这两个类型是一致的。此外，如果一个类型为不完整数组类型（参见 A.8.6 节），而另一个类型为完整数组类型，其它属性都相同，则认为这两个类型是一致的。最后，如果一个类型指定了一个旧式函数，而另一个类型指定了带形式参数声明的新式函数，二者之间其它方面都相同，则认为它们的类型也是一致的。

如果一个对象或函数的第一个外部声明包含 static 说明符，则该标识符具有内部连接，否则具有外部连接。有关连接的详细信息，参见 A.11.2 节中的讨论。

如果一个对象的外部声明带有初值，则该声明就是一个定义。如果一个外部对象声明不带有初值，并且不包含 extern 说明符，则它是一个临时定义。如果对象的定义出现在翻译单元中。则所有临时定义都将仅仅被认为是多余的声明；如果该翻译单元中不存在该对象的定义，则该临时定义将转变为一个初值为 0 的定义。

每个对象都必须有且仅有一个定义。对于具有内部连接的对象，该规则分别适用于每个翻译单元，这是因为，内部连接的对象对每个翻译单元是惟一的。对于具有外部连接的对象，该规则适用于整个程序。

说明：虽然单一定义规则（One-definition rule）在表达上与本书第 1 版有所不同，但在效果上是等价的。某些实现通过将临时定义的概念一般化而放宽了这个限制。在另一种形式中，一个程序中所有翻译单元的外部连接对象的所有临时定义将集中进行考虑，而不是在各翻译单元中分别考虑，UNIX 系统通常就采用这种方法，并且被认为是该标准的一般扩展。如果定义在程序中的某个地方出现，则临时定义仅被认为是声明，但如果没有定义出现，则所有临时定义将被转变为初值为 0 的定义。

A.11 作用域与连接

一个程序的所有单元不必同时进行编译。源文件文本可保存在若干个文件中，每个文件中可以包含多个翻译单元，预先编译过的例程可以从库中进行加载，程序中函数间的通信可以通过调用和操作外部数据来实现。

因此，我们需要考虑两种类型的作用域：第一种是标识符的词法作用域，它是体现标识符特性的程序文本区域；第二种是与具有外部连接的对象和函数相关的作用域，它决定各个单独编译的翻译单元中标识符之间的连接。

A.11.1 词法作用域

标识符可以在若干个名字空间中使用而互不影响。如果位于不同的名字空间中，即使是在同一作用域内，相同的标识符也可用于不同的目的。名字空间的类型包括：对象、函数、类型定义名和枚举常量；标号；结构标记、联合标记和枚举标记；各结构或联合自身的成员。

说明：这些规则与本手册第 1 版中所述的内容有几点不同。以前标号没有自己的名字空间；结构标记和联合标记分别有各自的名字空间，在某些实现中杖举标记也有自己的名字空间；把不同种类的标记放在同一名字空间中是新增加的限制。与第 1 版之间最大的不同在于：

每个结构和联合都为其成员建立不同的名字空间，因此同一名字可出现在多个不同的结构中。这一规则在最近几年使用得很多。

在外部声明中，对象或函数标识符的词法作用域从其声明结束的位置开始，到所在翻译单元结束为止。函数定义中形式参数的作用域从定义函数的程序块开始处开始，并贯穿整个函数；函数声明中形式参数的作用域到声明符的末尾处结束。程序块头部中声明的标识符的作用域是其所在的整个程序块。标号的作用域是其所在的函数。结构标记、联合标记、枚举标记或枚举常量的作用域从其出现在类型说明符中开始，到翻译单元结束为止（对外部声明而言）或到程序块结束为止（对函数内部声明而言）。

如果某一标识符显式地在程序块（包括构成函数的程序块）头部中声明，则该程序块外部中此标识符的任何声明都将被挂起，直到程序块结束再恢复其作用。

A.11.2 连接

在翻译单元中，具有内部连接的同一对象或函数标识符的所有声明都引用同一实体，并且，该对象或函数对这个翻译单元来说是惟一的。具有外部连接的同一对象或函数标识符的所有声明也引用同一实体，并且该对象或函数是被整个程序中共享的。

如 A.10.2 节所述，如果使用了 `static` 说明符，则标识符的第一个外部声明将使得该标识符具有内部连接，否则，该标识符将具有外部连接。如果程序块中对一个标识符的声明不包含 `extern` 说明符，则该标识符没有连接，并且在函数中是惟一的。如果这种声明中包含 `extern` 说明符，并且，在包含该程序块的作用域中有一个该标识符的外部声明，则该标识符与该外部声明具有相同的连接，并引用同一对象或函数。但是，如果没有可见的外部声明，则该连接是外部的。

A.12 预处理

预处理器执行宏替换、条件编译以及包含指定的文件，以#开头的命令行（“ #”前可以有空格）就是预处理器处理的对象。这些命令行的语法独立于语言的其它部分，它们可以出现在任何地方，其作用可延续到所在翻译单元的末尾（与作用域无关）。行边界是有实际意义的；每一行都将单独进行分析（有关如何将行连结起来的详细信息参见 A.12.4 节）。对预处理器而言，记号可以是任何语言记号，也可以是类似于#include 指令（参见 A.12.4 节）中表示文件名的字符序列，此外，所有未进行其它定义的字符都将被认为是记号。但是，在预处理器指令行中，除空格、横向制表符外的其它空白符的作用是没有定义的。

预处理过程在逻辑上可以划分为几个连续的阶段（在某些特殊的实现中可以缩减）。

1）首先，将 `A.12.1` 节所述的三字符序列替换为等价字符。如果操作系统环境需要，还要在源文件的各行之间插入换行符。

2）将指令行中位于换行符前的反斜杠符\删除掉，以把各指令行连接起来（参见 `A.12.2` 节）。

3）将程序分成用空白符分隔的记号。注释将被替换为一个空白符。接着执行预处理指令，并进行宏扩展（参见 `A.12.3` 节～`A.12.10` 节）。

4）将字符常量和字符串字面值中的转义字符序列（参见A.2.5节与A.2.6节）替换为等价字符，然后把相邻的字符串字面值连接起来。

5）收集必要的程序和数据，并将外部函数和对象的引用与其定义相连接，翻译经过以上处理得到的结果，然后与其它程序和库连接起来。

A.12.1 三字符序列

C语育源程序的字符集是7位ASCII码的子集，但它是ISO 646-1983不变代码集的超集。为了将程序通过这种缩减的字符集表示出来，下列所示的所有三字符序列都要用相应的单个字符替换，这种替换在进行所有它他处理之前进行。

```
??=  #          ??(  [          ??<  {
??/  \          ??)  ]          ??>  }
??'  ^          ??!  |          ??-  ~
```

除此之外不进行其它替换。

说明：三字符序列是ANSI标准新引入的特征。

A.12.2 行连接

通过将以反斜杠\结束的指令行末尾的反斜杠和其后的换行符删除掉。可以将若干指令行合并成一行。这种处理要在分隔记号之前进行。

A.12.3 宏定义和扩展

类似于下列形式的控制指令：

#define 标识符 记号序列

将使得预处理器把该标识符后续出现的各个实例用给定的记号序列替换。记号序列前后的空白符都将被丢弃掉。第二次用#define指令定义同一标识符是错误的，除非第二次定义中的标记序列与第一次相同（所有的空白分隔符被认为是相同的）。

类似于下列形式的指令行：

#define 标识符(标识符表opt) 记号序列

是一个带有形式参数（由标识符表指定）的宏定义，其中第一个标识符与圆括号（之间没有空格。同第一种形式一样，记号序列前后的空白符都将被丢弃掉。如果要对宏进行重定义，则必须保证其形式参数个数、拼写及记号序列都必须与前面的定义相同。

类似于下列形式的控制指令：

#undef 标识符

用于取消标识符的预处理器定义。将#undef 应用于未知标识符（即未用#define 指令定义的标识符）并不会导致错误。

按照第二种形式定义宏时，宏标识符（后面可以跟一个空白符，空白符是可选的）及其后用一对圆括号括起来的、由逗号分隔的记号序列就构成了一个宏调用。宏调用的实际参数是用逗号分隔的记号序列，用引号或嵌套的括号括起来的逗号不能用于分隔实际参数。在处理过程中，实际参数不进行宏扩展。宏调用时，实际参数的数目必须与定义中形式参数的数目匹配。实际参数被分离后，前导和尾部的空白符将被删除。随后，由各实际参数产生的记号序列将替换未用引号引起来的相应形式参数的标识符（位于宏的替换记号序列中）。除非替换序列中的形式参数的前面有一个#符号，或者其前面或后面有一个##符号，否则，在插入前要对宏调用的实际参数记号进行检查，并在必要时进行扩展。

两个特殊的运算符会影响替换过程。首先，如果替换记号序列中的某个形式参数前面直接是一个#符号（它们之间没有空白符），相应形式参数的两边将被加上双引号（"），随后，#和形式参数标识符将被用引号引起来的实际参数替换。实际参数中的字符串字面值、字符常量两边或内部的每个双引号（"）或反斜杠（\）前面都要插入一个反斜杠（\）。

其次，无论哪种宏的定义记号序列中包含一个##运算符，在形式参数替换后都要把##及其前后的空白符都删除掉，以便将相邻记号连接起来形成一个新记号。如果这样产生的记号无效，或者结果依赖于##运算符的处理顺序，则结果没有定义。同时，##也可以不出现在替换记号序列的开头或结尾。

对这两种类型的宏，都要重复扫描替换记号序列以查找更多的已定义标识符。但是。当某个标识符在某个扩展中被替换后，再次扫描并再次遇到此标识符时不再对其执行替换，而是保持不变。

即使执行宏扩展后得到的最终结果以#打头，也不认为它是预处理指令。

说明：有关宏扩展处理的细节信息，ANSI 标准比第 1 版描述得更详细。最重要的变化是加入了#和##运算符，这就使得引用和连接成为可能。某些新规则（特别是与连接有关的规则）比较独特（参见下面的例子）。

例如，这种功能可用来定义“表示常量”，如下例所示：

```
#define TABSIZE 100

int table[TABSIZE];
```

定义

```
#define ABSDIFF(a, b)  ((a)>(b) ? (a)-(b) : (b)-(a))
```

定义了一个宏，它返回两个参数之差的绝对值。与执行同样功能的函数所不同的是，参数与返回值可以是任意算术类型，甚至可以是指针。同时，参数可能有副作用，而且需要计算两次，一次进行测试，另一次则生成值。

假定有下列定义：

```
#define tempfile(dir)   #dir "%s"
```

宏调用 tempfile(/usr/tmp)将生成

```
"/usr/tmp" "%s"
```

随后，该结果将被连接为一个单个的字符串。给定下列定义：

```
#define cat(x, y)      x ## y
```

那么，宏调用 cat(var, 123)将生成 var123。但是，宏调用 cat(cat(1,2),3)没有定义：##阻止了外层调用的参数的扩展。因此，它将生成下列记号串：

```
cat  (  1  ,  2  )3
```

并且，)3 (不是一个合法的记号，它由第一个参数的最后一个记号与第二个参数的第一个记号连接而成。如果再引入第二层的宏定义，如下所示：

```
#define xcat(x, y)     cat(x,y)
```

我们就可以得到正确的结果。xcat(xcat(1, 2), 3)将生成 123，因为 xcat 自身的扩展不包含##运算符。

类似地，ABSDIFF(ABSDIFF(a,b),c)将生成所期望的经完全扩展后的结果。

A.12.4 文件包含

下列形式的控制指令：

```
#include <文件名>
```

将把该行替换为文件名指定的文件的内容。文件名不能包含>或换行符。如果文件名中包含字符"、'、\、或/*，则其行为没有定义。预处理器将在某些特定的位置查找指定的文件，查找的位置与具体的实现相关。

类似地，下列形式的控制指令：

```
#include "文件名"
```

首先从源文件的位置开始搜索指定文件（搜索过程与具体的实现相关），如果没有找到指定的文件，则按照第一种定义的方式处理。如果文件名中包含字符'、\、或/*，其结果仍然是没有定义的，但可以使用字符>。

最后，下列形式的指令行：

```
#include 记号序列
```

同上述两种情况都不同，它将按照扩展普通文本的方式扩展记号序列进行解释。记号序列必须被解释为<...>或"..."两种形式之一，然后再按照上述方式进行相应的处理。

#include 文件可以嵌套。

A.12.5 条件编译

对一个程序的某些部分可以进行条件编译，条件编译的语法形式如下：

预处理器条件：

if 行文本 elif 部分 opt else 部分 opt #endif

if 行：

#if 常量表达式

#ifdef 标识符

#ifndef 标识符

elif 部分：

elif 行 文本 elif 部分 opt

elif 行：

#elif 常量表达式

else 部分：

else 行 文本

else 行：

#else

其中，每个条件编译指令（if 行、elif 行、else 行以及#endif）在程序中均单独占一行。预处理器依次对#if 以及后续的#elif 行中的常量表达式进行计算，直到发现某个指令的常量表达式为非 0 值为止，这时将放弃值为 0 的指令行后面的文本。常量表达式不为 0 的#if 和#elif 指令之后的文本将按照其它普通程序代码一样进行编译。在这里，“文本”是指任何不属于条件编译指令结构的程序代码，它可以包含预处理指令，也可以为空。一旦预处理器发现某个#if 或#elif 条件编译指令中的常量表达式的值不为 0，并选择其后的文本供以后的编译阶段使用时，后续的#elif 和#else 条件编译指令及相应的文本将被放弃。如果所有常量表达式的值都为 0，并且该条件编译指令链中包含一条#else 指令。则将选择#else 指令之后的文本。除了对条件编译指令的嵌套进行检查之外，条件编译指令的无效分支（即条件值为假的分支）控制的文本都将被忽略。

#if 和#elif 中的常量表达式将执行通常的宏替换。并且，任何下列形式的表达式：

defined 标识符

或

Defined(标识符)

都将在执行宏扫描之前进行替换，如果该标识符在预处理器中已经定义，则用 1 替换它，否则，用 0 替换。预处理器进行宏扩展之后仍然存在的任何标识符都将用 0 来替换。最后，每个整型常量都被预处理器认为其后面跟有后缀 L，以便把所有的算术运算都当作是在长整型或无符号长整型的操作数之间进行的运算。

进行上述处理之后的常量表达式（参见 A.7.19 节）满足下列限制条件：它必须是整型，并且其中不包含 sizeof、强制类型转换运算符或枚举常量。

下列控制指令：

#ifdef 标识蒋

#ifndef 标识符

分别等价于：

#if defined 标识符

#if !defined 标识符

说明：#elif 是 ANSI 中新引入的条件编译指令，但此前它已经在某些预处理器中实现了。defined 预处理器运算符也是 ANSI 中新引入的特征。

A.12.6 行控制

为了便于其它预处理器生成 C 语言程序，下列形式的指令行：

#line 常量 "文件名"

#line 常量

将使编译器认为（出于错误诊断的目的）：下一行源代码的行号是以十进制整型常量的形式给出的，并且，当前的输入文件是由该标识符命名的。如果缺少带双引号的文件名部分，则将不改变当前编译的源文件的名字。行中的宏将先进行扩展，然后再进行解释。

A.12.7 错误信息生成

下列形式的预处理器控制指令：

#error 记号序列 opt

将使预处理器打印包含该记号序列的诊断信息。

A.12.8 pragma

下列形式的控制指令：

#pragma 记号序列 opt

将使预处理器执行一个与具体实现相关的操作。无法识别的 pragma（编译指示）将被忽略掉。

A.12.9 空指令

下列形式的预处理器行不执行任何操作：

#

A.12.10预定义名字

某些标识符是预定义的，扩展后将生成特定的信息。它们同预处理器表达式运算符 defined 一样，不能取消定义或重新进行定义。

__LINE__	包含当前源文件行数的十进制常量。
__FILE__	包含正在被编译的源文件名字的字符串字面值。
__DATE__	包含编译日期的字符串字面值，其形式为“Mmm dd yyyy”。
__TIME__	包含编译时间的字符串字面值，其形式为“hh:mm:ss”。
__STDC__	整型常量 1。只有在遵循标准的实现中该标识符才被定义为 1。

说明：#error 与#pragma 是 ANSI 标准中新引入的特征。这些预定叉的预处理器宏也是新引入的，其中的一些宏先前已经在某些编译器中实现。

A.13 语法

这一部分的内容将简要概述本附录前面部分中讲述的语法。它们的内容完全相同，但顺序有一些调整。

本语法没有定义下列终结符：整型常量、字符常量、浮点常量、标识符、字符串和枚举常量。以打字字体形式表示的单词和符号是终结符。本语法可以直接转换为自动语法分析程序生成器可以接受的输入。除了增加语法记号说明产生式中的候选项外，还需要扩展其中的“one of”结构，并（根据语法分析程序生成器的规则）复制每个带有 opt 符号的产生式：一个带有 opt 符号，一个没有 opt 符号。这里还有一个变化，即删除了产生式“类型定义名:标识符”，这样就使得其中的类型定义名成为个终结符。该语法可被 YACC 语法分析程序生成器接受，但由于 if-else 的歧义性问题，还存在一处冲突。

翻译单元

外部声明

翻译单元 外部声明

外部声明：

函数定义

声明

函数定义：

声明说明符 opt 声明符声明表 opt 复合语句

声明：

声明说明符 初始化声明符表 opt;

声明表：

声明

声明表 声明

声明说明符：

存储类说明符 声明说明符 opt

类型说明符 声明说明符 opt

类型限定符 声明说明符 opt

存储类说明符：one of

auto register static extern tyedef

类型说明符：one of

void char short int long float double signed

unsigned 结构或联合说明符 枚举说明符 类型定义名

类型限定符：one of

const volatile

结构或联合说明符：

结构或联合 标识符 opt {结构声明表}

结构或联合 标识符

结构或联合：one of

struct union

结构声明表：

结构声明

结构声明表 结构声明

初始化声明符表

初始化声明符

初始化声明符表, 初始化声明符

初始化声明符：

声明符

声明符=初始化符

结构声明：

说明符限定符表 结构声明符表;

说明符限定符表：

类型说明符 说明符限定符表 opt

类型限定符 说明符限定符表 opt

结构声明符表：

结构声明符

结构声明符表, 结构声明符

结构声明符：

声明符

声明符 opt: 常量表达式

敉举说明符

enum 标识符 opt {枚举符表}

enum 标识符

枚举符表：

枚举符

枚举符表, 枚举符

枚举符

标识符

标识符=常量表达式

声明符

指针 opt 直接声明符

直接声明符：

标识符

(声明符)

直接声明符[常量表达式]

直接声明符(形式参数类型表)

直接声明符(标识符表 opt)

指针：

* 类型限定符表 opt

* 类型限定符表 opt 指针

类型限定符表：

类型限定符

类型限定符表 类型限定符

形式参数类型表：

    形式参数表

    形式参数表, ...

形式参数表：

    形式参数表声明

    形式参数表, 形式参数声明

形式参数声明：

    声明说明符 声明符

    声明说明符 抽象声明符 opt

标识符表：

    标识符

    标识符表, 标识符

初值：

    赋值表达式

    {初值表}

    {初值表, }

初值表：

    初值

    初值表, 初值

类型名：

    说明符限定符表 抽象声明符 opt

抽象声明符：

    指针

    指针 opt 直接抽象声明符

直接抽象声明符：

    (抽象声明符)

    直接抽象声明符 opt [常量表达式]

    直接抽象声明符 opt (形式参数类型表 opt)

类型定义名：

    标识符

语句：

带标号语句

表达式语句

复合语句

选择语句

循环语句

跳转语句

带标号语句：

标识符: 语句

case 常量表达式语句

dafault: 语句

表达式语句；

表达式 opt;

复合语句：

{声明表 opt 语句表 opt}

语句表：

语句

语句表 语句

选择语句：

if (表达式) 语句

if (表达式) 语句 else 语句

switch (表达式) 语句

循环语句

while (表达式) 语句

do 语句 while (表达式);

for (表达式 opt; 表达式 opt; 表达式 opt) 语句

跳转语句：

goto 标识符;

continue;

break;

`return` 表达式 `opt;`

表达式：

赋值表达式

表达式, 赋值表选式

赋值表达式：

条件表达式

一元表达式 赋值运算符 赋值表达式

赋值运算符：one of

`= *= /= %= += -= <<= >>= &= ^= |=`

条件表达式：

逻辑或表达式

或表达式?表达式:条件表达式

常量表达式：

条件表达式

逻辑或表达式：

逻辑与表达式

逻辑或表达式||逻辑与表达式

逻辑与表达式

接位或表达式

逻辑与表达式&&按位或表达式

按位或表达式：

按位异或表达式

按位或表达式|按位异或表达式

接位异或表达式：

按位与表达式

接位异或表达式^按位与表达式

按位与表达式：

相等类表达式

按位与表达式&相等类表达式

相等类表达式：

关系表达式

相等类表达式==关系表达式

相等类表达式!=关系表达式

关系表达式：

移位表达式

关系表达式<移位表达式

关系表达式>移位表达式

关系表达式<=移位表达式

关系表达式>=移位表达式

移位表达式

加法类表达式

移位表达式<<加法类表达式

移位表达式>>加法类表达式

加法类表达式：

乘法类表达式

加法类表达式+乘法类表达式

加法类表达式-乘法类表达式

乘法类表达式：

强制类型转换表达式

类表达式*强制类型转换表达式

乘法类表达点/强制类型转换表达式

乘法类表达式%强制类型转换表达式

强制类型转换表达式：

一元表达式

(类型名)强制类型转换表达式

一元表达式：

后缀一元式

++一元表达式

--一元表达式

一元运算符强制类型转换表达式

sizeof 一元表达式

sizeof(类型名)

一元运算符：one of

& * + - ~ !

后缀表达式：

初等表达式

后缀表达式[表达式]

后缀表达式(参数表达式表 opt)

后缀表达式.标识符

后缀表达式->标识符

后缀表达式++

后缀表达式--

初等表达式：

标识符

常量

字符串

(表达式)

参数表达式表：

赋值表达式

参数表达式表, 赋值表达式

常量：

整型常量

字符常量

浮点常量

枚举常量

下列预处理器语法总结了控制指令的结构，但不适合于机械化的语法分析。其中包含符号“文本”（即通常的程序文本）、非条件预处理器控制指令或完整的预处理器条件结构。

控制指令：

=define 标识符 记号序列

=define 标识符(标识符表 opt) 记号序列

#undef 标识符

#include <文件名>

#include "文件名"

#include 记号序列

#line 常量 "文件名"

#line 常量

#error 记号序列。

#pragma 记号序列。

#

预处理器条件指令

预处理器条件指令：

if 行 文本 elif 部分 opt else 部分 opt #endif

if 行：

#if 常量表达式

#ifdef 标识符

#ifndef 标识符

elif 部分：

elif 行 文本 elif 部分 opt

elif 行：

#elif 常量表达式

else 部分：

else 行 文本

else 行：

#else

附录B 标准库

本附录总结了 ANSI 标准定义的函数库。标准库不是 C 语言本身的构成部分，但是支持标准 C 的实现会提供该函数库中的函数声明、类型以及宏定义。在这部分内容中，我们省略了一些使用比较受限的函数以及一些可以通过其它函数简单合成的函数，也省略了多字节字符的内容，同时，也不准备讨论与区域相关的一些属性，也就星与本地语言、国籍或文化相关的属性。

标准库中的函数，类型以及宏分别在下面的标准头文件中定义：

```
<assert.h>  <float.h>   <math.h>    <stdarg.h>  <stdlib.h>
<ctype.h>   <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>   <locale.h>  <signal.h>  <stdio.h>   <time.h>
```

可以通过下列方式访问头文件：

```
#include <头文件>
```

头文件的包含顺序是任意的，并可包含任意多次。头文件必须被包含在任何外部声明或定义之外，并且，必须在使用头文件中的任何声明之前包含头文件。头文件不一定是一个源文件。

以下划线开头的外部标识符保留给标准库使用，同时，其它所有以一个下划线和一个大写字母开头的标识符以及以两个下划线开头的标识符也都保留给标准库使用。

B.1 输入与输出：<stdio.h>

头文件`<stdio.h>`中定义的输入和输出函数、类型以及宏的数目几乎占整个标准库的三分之一。

流（stream）是与磁盘或其它外围设备关联的数据的源或目的地。尽管在某些系统中（如在著名的 UNIX 系统中），文本流和二进制流是相同的，但标准库仍然提供了这两种类型的流。文本流是由文本行组成的序列，每一行包含 0 个或多个字符，并以`'\n'` 结尾。在某些环境中，可能需要将文本流转换为其它表示形式（例如把`'\n'`映射成回车符和换行符），或从其它表示形式转换为文本流。二进制流是由未经处理的字节构成的序列，这些字节记录着内部数据，并具有下列性质：如果在同一系统中写入二进制流，然后再读取该二进制流，则读出和写入的内容完全相同。

打开一个流，将把该流与一个文件或设备连接起来，关闭流将断开这种连接，打开一个文件将返回一个指向 `FILE` 类型对象的指针，该指针记录了控制该流的所有必要信息，在不引起歧义的情况下，我们在下文中将不再区分“文件指针”和“流”。

程序开始执行时，`stdin`、`stdout` 和 `stderr` 这 3 个流已经处于打开状态。

B.1.1 文件操作

下列函数用于处理与文件有关的操作。其中，类型 size_t 是由运算符 sizeof 生成的无符号整型。

```
FILE *fopen(const char *filename, const char *mode)
```

fopen 函数打开 filename 指定的文件，并返回一个与之相关联的流。如果打开操作失败，则返回 NULL。

访问模式 mode 可以为下列合法值之一：

"r"	打开文本文件用于读
"w"	创建文本文件用于写，并删除已存在的内容（如果有的话）
"a"	追加；打开或创建文本文件，并向文件末尾追加内容
"r+"	打开文本文件用于更新（即读和写）
"w+"	创建文本文件用于更新，并删除已存在的内容（如果有的话）
"a+"	追加；打开或创建文本文件用于更新，写文件时追加到文件末尾

后 3 种方式（更新方式）允许对同一文件进行读和写。在读和写的交叉过程中，必须调用 fflush 函数或文件定位函数。如果在上述访问模式之后再加上 b，如“ rb”或“ w+b”等，则表示对二进制文件进行操作。文件名 filename 限定最多为 FILENAME_MAX 个字符。一次最多可打开 FOPEN_MAX 个文件。

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

freopen 函数以 mode 指定的模式打开 filename 指定的文件，并将该文件关联到 stream 指定的流。它返回 stream；若出错则返回 NULL。Freopen 函数一般用于改变与 stdin、stdout 和 stderr 相关联的文件。

```
int fflush(FILE *stream)
```

对输出流来说，fflush 函数将已写到缓冲区但尚未写入文件的所有数据写到文件中。对输入流来说，其结果是未定义的。如果在写的过程中发生错误，则返回 EOF，否则返回 0。fflush(NULL)将清洗所有的输出流。

```
int fclose(FILE *stream)
```

fclose 函数将所有未写入的数据写入 stream 中，丢弃缓冲区中的所有未读输入数据，并释放自动分配的全部缓冲区，最后关闭流。若出错则返回 EOF，否则返回 0。

```
int remove(const char *filename)
```

remove 函数删除 filename 指定的文件，这样，后续试图打开该文件的操作将失败。如果删除操作失败，则返回一个非 0 值。

```
int rename(const char *oldname, const char *newname)
```

rename 函数修改文件的名字。如果操作失败，则返回一个非 0 值。

```
FILE *tmpfile(void)
```

tmpfile 函数以模式"wb+"创建一个临时文件，该文件在被关闭或程序正常结束时将被自动删除。如果创建操作成功，该函数返回一个流；如果创建文件失败，则返回 NULL。

```
char *tmpnam(char s[L_tmpnam])
```

tmpnam(NULL)函数创建一个与现有文件名不同的字符串，并返回一个指向一内部静态数组的指针。tmpnam(S)函数把创建的字符串保存到数组 s 中，并将它作为函数值返回。s 中至少要有 L_tmpnam 个字符的空间。Tmpnam 函数在每次被调用时均生成不同的名字。在程序执行的过程中，最多只能确保生成 TMP_MAX 个不同的名字。注意，tmpnam 函数只是用于创建一个名字，而不是创建一个文件。

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

setvbuf 函数控制流 stream 的缓冲。在执行读、写以及其它任何操作之前必须调用此函数。当 mode 的值为_IOFBF 时，将进行完全缓冲。当 mode 的值为_IOLBF 时，将对文本文件进行行缓冲，当 mode 的值为_IONBF 时，表示不设置缓冲。如果 buf 的值不是 NULL，则 setvbuf 函数将 buf 指向的区域作为流的缓冲区，否则将分配一个缓冲区。size 决定缓冲区的长度。如果 setvbuf 函数出错，则返回个一非 0 值。

```
void setbuf(FILE *stream, char *buf)
```

如果 buf 的值为 NULL，则关闭流 stream 的缓冲；否则 setbuf 函数等价于(void)setvbuf(stream, buf, _IOFBF, BUFSIZ)。

B.1.2 格式化输出

printf 函数提供格式化输出转换。

```
int fprintf(FILE *stream, const char *format, ...)
```

fprintf 函数按照 format 说明的格式对输出进行转换，并写到 stream 流中。返回值是实际写入的字符数。若出错则返回一个负值。

格式串由两种类型的对象组成：普通字符（将被复制到输出流中）与转换说明（分别决定下一后续参数的转换和打印）。每个转换说明均以字符%开头，以转换字符结束。在%与转换字符之间可以依次包括下列内容：

- 标志（可以以任意顺序出现），用于修改转换说明
  - 指定被转换的参数在其字段内左对齐
  + 指定在输出的数前面加上正负号
  空格 如果第一个字符不是正负号，则在其前面加上一个空格
  0 对于数值转换，当输出长度小于字段宽度时，添加前导 0 进行填充
  # 指定另一种输出形式。如果为 o 转换，则第一个数字为零；如果为 x 或 X 转换，则指定在输出的非 0 值前加 0x 或 0X；对于 e、E、f、g 或 G 转换，指定输出总包括一个小数点；对于 g 或 G 转换，指定输出值尾部无意义的 0 将被保留
- 一个数值，用于指定最小字段宽度。转换后的参数输出宽度至少要达到这个数值。如果参数的字符数小于此数值，则在参数宽边（如果要求左对齐的话则为右边）填充一些字符。填充字符通常为空格，但是，如果设置了 0 填充标志，则填充字符为 0。
- 点号，用于分隔字段宽度和精度。
- 表示精度的数。对于字符串，它指定打印的字符的最大个数；对于 e、E 或 f 转换，它指定打印的小数点后的数字位数；对于 g 或 G 转换，它指定打印的有效数字位数；对于整型数，它指定打印的数字位数（必要时可加填充位 0 以达到要求的宽度）。
- 长度修饰符 h、l 或 L。h 表示将相应的参数按 short 或 unsigned short 类型输

出。l 表示将相应的参数按 long 或 unsigned long 类型输出；L 表示将相应的参数按 long double 类型输出。

宽度和精度中的任何一个或两者都可以用*指定，这种情况下，该值将通过转换下一个参数计算得到（下一个参数必须为 int 类型）。

表 B-1 中列出了这些转换字符及其意义。如果%后面的字符不是转换字符，则其行为没有定义。

表 B-1 printf 函数的转换字符

转换字符	参数类型；转换结果
d, i	int：有符号十进制表示
o	unsigned int：无符号八进制表示
x, X	unsigned int：无符号十六进制表示（没有前导 0x 或 0X），如果是 0x，则使用 abcdef，如果是 0X，则使用 ABCDEF
u	int：无符号十进制表示
c	int：转换为 unsigned char 类型后为一个字符
s	char *：打印字符串中的字符，直到遇到'\0'或已打印了由精度指定的字符数
f	double：形式为[-]mmm.ddd 的十进制表示，其中，d 的数目由精度确确定，默认精度为 6。精度为 0 时不输出小数点
e, E	double：形式为[-]m.dddddd e ±xx 或[-]m.dddddd E ±xx。d 的数目由精度确定，默认精度为 6。精度为 0 时不输出小数点
g, G	double：当指数小于-4 或大于等于精度时，采用%e 或%E 的格式，否则采用%f 的格式。尾部的 0 和小数点不打印
p	void *：打印指针值（具体表示方式与实现有关）
n	int *：到目前为止，此 printf 调用输出的字符的数目将被写入到相应参数中。不进行参数转换
%	不进行参数转换；打印一个符号%

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

printf(...)函数等价于 fprintf(stdout, …)。

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

sprintf 函数与 printf 函数基本相同，但其输出将被写入到字符串 s 中，并以'\0'结束。s 必须足够大，以足够容纳下输出结果。该函数返回实际输出的字符数，不包括'\0'。

```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)
```

vprintf、vfprintf、vsprintf 这 3 个函数分别与对应的 printf 函数等价，但它们用 arg 代替了可变参数表。arg 由宏 va_start 初始化，也可能由 va_arg 调用初始化。详细信息参见 B.7 节中对<stdarg.h>头文件的讨论。

B.1.3 格式化输入

scanf 函数处理格式化输入转换。

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf 函数根据格式串 format 从流 stream 中读取输入，并把转换后的值赋值给后续各个参数，其中的每个参数都必须是一个指针。当格式串 format 用完时，函数返回。如果到达文件的末尾或在转换输入前出错，该函数返回 EOF；否则，返回实际被转换并赋值的输入项的数目。

格式串 format 通常包括转换说明，它用于指导对输入进行解释。格式字符串中可以包含下列项目：

- 空格或制表符
- 普通字符（%除外），它将与输入流中下一个非空白字符进行匹配
- 转换说明，由一个%、一个赋值屏蔽字符*（可选）、一个指定最大字段宽度的数（可选）、一个指定目标字段宽度的字符（h、l 或 L）（可选）以及一个转换字符组成。

转换说明决定了下一个输入字段的转换方式。通常结果将被保存在由对应参数指向的变量中。但是，如果转换说明中包含赋值屏蔽字符*，例如%*s，则将跳过对应的输入字段，并不进行赋值。输入字段时一个由非空白符字符组成的字符串，当遇到下一个空白符或达到最大字段宽度（如果有的话）时，对当前输入字段的读取结束。这意味着，scanf 函数可以跨越行的边界读取输入，因为换行符也是空白符（空白符包括空格、横向制表符、纵向制表符、换行符、回车符和换页符）。

转换字符说明了对输入字段的解释方式。对应的参数必须是指针。合法的转换字符如表 B-2 所示。

如果参数是指向 short 类型而非 int 类型的指针，则在转换字符 d、i、n、o、u 和 x 之前可以加上前缀 h。如果参数是指向 long 类型的指针，则在这几个转换字符前可以加上字母 l。如果参数是指向 double 类型而非 float 类型的指针，则在转换字符 e、f 和 g 前可以加上字母 l。如果参数是指向 long double 类型的指针，则在转换字符 e、f 和 g 前可以加上字母 L。

表 B-2 scanf 函数的转换字符

转换字符	输入数据；参数类型
d	十进制整数；int *
i	整型数；int *。该整型数可以是八进制（以 0 开头）或十六进制（以 0x 或 0X 开头）
o	八进制整型数（可以带或不带前导 0）；int *
u	无符号十进制整型数；unsigned int *
x	十六进制整型数（可以带或不带前导 0x 或 0X）；int *
c	字符；char *，按照字段宽度的大小把读取的字符保存到制定的数组中，不增加'\0'字段宽度的默认值为 1。在这种情况下，读取输入时将不跳过空白符。如果需要读入下一个非空白符，可以使用%1s
s	由非空白符组成的字符串（不包含引号）；char *，它指向一个字符数组，该字符数组必须有足够空间，以保存该字符串以及在尾部添加的'\0'字符
e, f, g	浮点数，float *。Float 类型浮点数的输入格式为：一个可选的正负号、一个可能包含小数点的数字串，

	一个可选的指数字段（字母 e 或 E 后跟一个可能带正负号的整型数）
p	printf("%p")函数调用打印的指针值；void *
n	将到目前为止该函数调用读取的字符数写入对应的参数中；int *。部读取输入字符。不增加已转换的项目计数
[...]	与方括号中的字符集合匹配的输入字符中最长的非空字符串；char *。末尾将添加'\0'。[]...]表示集合中包含字符“]”
[^...]	与方括号中的字符集合不匹配的输入字符中最长的非空字符串；char *。末尾将添加'\0'。[^]...]表示集合中不包含字符“]”
%	表示“%”，不进行赋值

```
int scanf(const char *format, ...)
```

scanf(...)函数与 fscanf(stdin, ...)相同。

```
int sscanf(const char *s, const char *format, ...)
```

sscanf(s, ...)函数与 scanf(...)等价，所不同的是，前者的输入字符来源于字符串 s。

B.1.4 字符输入 / 输出函数

```
int fgetc(FILE *stream)
```

fqetc 函数返回 stream 流的下一个字符，返回类型为 unsigned char（被转换为 int 类型）。如果到达文件末尾或发生错误，则返回 EOF。

```
char *fgets(char *s, int n, FILE *stream)
```

fgets 函数最多将下 n-1 个字符读入到数组 s 中。当遇到换行符时，把换行符读入到数组 s 中，读取过程终止。数组 s 以'\0'结尾。fgets 函数返回数组 s。如果到达文件的末尾或发生错误，则返回 NULL。

```
int fputc(int c, FILE *stream)
```

fputc 函数把字符 c（转换为 unsigned char 类型）输出到流 stream 中。它返回写入的字符，若出错则返回 EOF。

```
int fputs(const char *s, FILE *stream)
```

fputs 函数把字符串 s（不包含字符'\n'）输出到流 Btream 中；它返回一个非负值，若出错则返回 EOF。

```
int getc(FILE *stream)
```

getc 函数等价于 fgetc，所不同的是，当 getc 函数定义为宏时，它可能多次计算 stream 的值。

```
int getchar(void)
```

getchar 函数等价于 getc(stdin)。

```
char *gets(char *s)
```

gets 函数把下一个输入行读入到数组 s 中，并把末尾的换行符替换为字符'\0'。它返回数组 s，如果到达文件的末尾或发生错误，则返回 NULL。

```
int putc(int c, FILE *stream)
```

putc 函数等价于 fputc，所不同的是，当 putc 函数定义为宏时，它可能多次计算 stream 的值。

```
int putchar(int c)
```

putchar(c)函数等价于 putc(c, stdout)。

```
int puts(const char *s)
```

puts 函数把字符串 s 和一个换行符输出到 stdout 中。如果发生错误，则返回 EOF；否则返回一个非负值。

```
int ungetc(int c, FILE *stream)
```

ungetc 函数把 c（转换为 unsigned char 类型）写回到流 stream 中，下次对该流进行读操作时，将返回该字符。对每个流只能写回一个字符，且此字符不能是 EOF。ungetc 函数返回被写回的字符，如果发生错误，则返回 EOF。

B.1.5 直接输入 / 输出函数

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
```

fread 函数从流 stream 中读取最多 nobj 个长度为 size 的对象，并保存到 ptr 指向的数组中。它返回读取的对象数目，此返回值可能小于 nobj。必须通过函数 feof 和 ferror 获得结果执行状态。

```
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)
```

fwrite 函数从 ptr 指向的数组中读取 nobj 个长度为 size 的对象，并输出到流 stream 中。它返回输出的对象数目。如果发生错误，返回值会小于 nobj 的值。

B.1.6 文件定位函数

```
int fseek(FILE *stream, long offset, int origin)
```

fseek 函数设置流 stream 的文件位置，后续的读写操作将从新位置开始。对于二进制文件，此位置被设置为从 origin 开始的第 offset 个字符处。Origin 的值可以为 SEEK_SET（文件开始处）、SEEK_CUR（当前位置）或 SEEK_END（文件结束处）。对于文本流，offset 必须设置为 0，或者是由函数 ftell 返回的值（此时 origin 的值必须是 SEEK_SET）。fseek 函数在出错时返回一个非 0 值。

```
long ftell(FILE *stream)
```

ftell 函数返回 stream 流的当前文件位置，出错时该函数返回-lL。

```
void rewind(FILE *stream)
```

rewind(fp)函数等价于语句 fseek(fp, 0L, SEEK_SET)；clearerr(fp)的执行结果。

```
int fgetpos(FILE *stream, fpos_t *ptr)
```

fgetpos 函数把 stream 流的当前位置记录在*ptr 中，供随后的 fsetpos 函数调用使用。若出错则返回一个非 0 值。

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

fsetpos 函数将流 stream 的当前位置设置为 fgetpos 记录在*ptr 中的位置。若出错则返回一个非 0 值。

B.1.7 错误处理函数

当发生错误或到达文件末尾时，标准库中的许多函数都会设置状态指示符。这些状态指示符可被显式地设置和测试。另外，整型表达式 errno（在<errno.h>中声明）可以包含一个错误编号，据此可以进一步了解最近一次出错的信息。

```
void clearerr(FILE *stream)
```

clearerr 函数清除与流 stream 相关的文件结束符和错误指示符。

```
int feof(FILE *stream)
```

如果设置了与 stream 流相关的文件结束指示符，feof 函数将返回一个非 0 值，

```
int ferror(FILE *stream)
```

如果设置了与 stream 流相关的错误指示符，ferror 函数将返回一个非 0 值。

```
void perror(const char *s)
```

perror(s)函数打印字符串 s 以及与 errno 中整型值相应的错误信息，错误信息的具体内容与具体的实现有关。该函数的功能类似于执行下列语句：

```
fprintf(stderr, "%s: %s\n", s, "error message");
```

有关函数 strerror 的信息，参见 B.3 节中的介绍。

B.2 字符类别测试：<ctype.h>

头文件<ctype.h>中声明了一些测试字符的函数。每个函数的参数均为 int 类型，参数的值必须是 EOF 或可用 unsigned char 类型表示的字符，函数的返回值为 int 类型。如果参数 c 满足指定的条件，则函数返回非 0 值（表示真），否则返回 0（表示假）。这些函数包括：

```
isalnum(c)
```
函数 isalpha(c)或 isdigit(c)为真

isalpha(c)　函数 isupper(c)或 islower(c)为真
iscntrl(c)　c 为控制字符
isdigit(c)　c 为十进制数字
isgraph(c)　c 是除空格外的可打印字符
islower(c)　c 是小写字母
isprint(c)　c 是包括空格的可打印字符
ispunct(c)　c 是除空格、字母和数字外的可打印字符
isspace(c)　c 是空格、换页符、换行符、回车符、横向制表符或纵向制表符
isupper(c)　c 是大写字母
isxdigit(c)　c 是十六进制数字

在 7 位 ASCII 字符集中，可打印字符是从 0x20（' '）到 0x7E（'~'）之间的字符；控制字符是从 0（NUL）到 0xlF（US）之间的字符以及字符 0x7F（DEL）。

另外，下面两个函数可用于字母的大小写转换：

int tolower(int c)　将 c 转换为小写字母

int toupper(int c)　将 c 转换为大写字母

如果 c 是大写字母，则 tolower(c)返回相应的小写字母，否则返回 c。如果 c 是小写字母，则 toupper(c)返回相应的大写字母，否则返回 c。

B.3　字符串函数：<string.h>

头文件<string.h>中定义了两组字符串函数。第一组函数的名字以 str 开头；第二组函数的名字以 mem 开头。除函数 memmove 外，其它函数都没有定义重叠对象间的复制行为。比较函数将把参数作为 unsigned char 类型的数组看待。

在下表中，变量 s 和 t 的类型为 char *；cs 和 ct 的类型为 const char *；n 的类型为 size_t；c 的类型为 int（将被转换为 char 类型）。

char *strcpy(s,ct)	将字符串 ct（包括'\0'）复制到字符串 s 中，并返回 s
char *strncpy(s,ct,n)	将字符串 ct 中最多 n 个字符复制到字符串 s 中，并返回 s。如果 ct 中少于 n 个字符，则用'\0'填充
char *strcat(s,ct)	将字符串 ct 连接到 s 的尾部，并返回 s
char *strncat(s,ct,n)	将字符串 ct 中最多前 n 个字符连接到字符串 s 的尾部，并以'\0'结束；该函数返回 s
int strcmp(cs,ct)	比较字符串 cs 和 ct；当 cs<ct 时，返回一个负数；当 cs==ct 时，返回 0；当 cs>ct 时，返回一个正数
int strncmp(cs,ct,n)	将字符串 cs 中至多前 n 个字符与字符串 ct 相比较。当 cs<ct 时，返回一个负数：当 cs==ct 时，返回 0；当 cs>ct 时，返回一个正数
char *strchr(cs,c)	返回指向字符 c 在字符串 cs 中第一次出现的位置的指针；如果 cs 中不包含 c，则该函数返回 NULL

char *strrchr(cs,c)	返回指向字符 c 在字符串 cs 中最后一次出现的位置的指针；如果 cs 中不包含 c，则该函数返回 NULL
size_t strspn(cs,ct)	返回字符串 cs 中包含 ct 中的字符的前缀的长度
size_t strcspn(cs,ct)	返回字符串 cs 中不包含 ct 中的字符的前缀的长度
char *strpbrk(cs,ct)	返回一个指针，它指向字符串 ct 中的任意字符第一次出现在字符串 cs 中的位置；如果 cs 中没有与 ct 相同的字符，则返回 NULL
char *strstr(cs,ct)	返回一个指针，它指向字符串 ct 第一次出现在字符串 cs 中的位置；如果 cs 中不包含字符串 ct，则返回 NULL
size_t strlen(cs)	返回字符串 cs 的长度
char *strerror(n)	退回一个指针，它指向与错误编号 n 对应的错误信息字符串（错误信息的具体内容与具体实现相关）
char *strtok(s,ct)	strtok 函数在 s 中搜索由 ct 中的字符界定的记号。详细信息参见下面的讨论

对 strtok(s, ct)进行一系列调用，可以把字符串 s 分成许多记号，这些记号以 ct 中的字符为分界符。第一次调用时，s 为非空。它搜索 s，找到不包含 ct 中字符的第一个记号，将 s 中的下一个字符替换为'\0'，并返回指向记号的指针。随后，每次调用 strtok 函数时（由 s 的值是否为 NULL 指示），均返回下一个不包含 ct 中字符的记号。当 s 中没有这样的记号时，返回 NULL。每次调用时字符串 ct 可以不同。

以 mem 开头的函数按照字符数组的方式操作对象，其主要目的是提供一个高效的函数接口。在下表列出的函数中，s 和 t 的类型均为 void *，cs 和 ct 的类型均为 const void *，n 的类型为 size_t，c 的类型为 int（将被转换为 unsigned char 类型）。

void *memcpy(s,ct,n)	将字符串 ct 中的 n 个字符拷贝到 s 中，并返回 s
void *memmove(s,ct,n)	该函数的功能与 memcpy 相似，所不同的是，当对象重叠时，该函数仍能正确执行
int memcmp(cs,ct,n)	将 cs 的前 n 个字符与 ct 进行比较，其返回值与 strcmp 的返回值相同
void *memchr(cs,c,n)	返回一个指针，它指向 c 在 cs 中第一次出现的位置。如果 cs 的前 n 个字符中找不到匹配，则返回 NULL
void *memset(s,c,n)	将 s 中的前 n 个字符替换为 c，并返回 s

B.4 数学函数：<math.h>

头文件<math.h>中声明了一些数学函数和宏。

宏 EDOM 和 ERANGE（在头文件<error.h>中声明）是两个非 0 整型常量，用于指示函数的定义域错误和值域错误；HUGE_VAL 是一个 double 类型的正数。当参数位于函数定义的作用域之外时，就会出现定义域错误。在发生定义域错误时，全局变量 errno 的值将被设置为 EDOM，函数的返回值与具体的实现相关。如果函数的结果不能用 double 类型表示，则会发生值域错误。当结果上溢时，函数返回 HUGE_VAL，并带有正确的正负号，errpo 的值将被设置为 ERANGE。当结果下溢时，函数返回 0，而 errno 是否设置为 ERANGE 要视具体

的实现而定。

在下表中，x 和 y 的类型为 double，n 的类型为 int，所有函数的返回值的类型均为 double。三角函数的角度用弧度表示。

sin(x)	x 的正弦值
cos(x)	x 的余弦值
tan(x)	x 的正切值
asin(x)	$\sin^{-1}(x)$，值域为$[-\pi/2, \pi/2]$，其中 $x\in[-1, 1]$
acos(x)	$\cos^{-1}(x)$，值域为$[0, \pi]$，其中 $x\in[-1, 1]$
atan(x)	$\tan^{-1}(x)$，值域为$[-\pi/2, \pi/2]$
atan2(y,x)	$\tan^{-1}(y/x)$，值域为$[-\pi, \pi]$
sinh(x)	x 的双曲正弦值
cosh(x)	x 的双曲余弦值
tanh(x)	x 的双曲正切值
exp(x)	幂函数 e^x
log(x)	自然对数 ln(x)，其中 x>0
log10(x)	以 10 为底的对数 $\log_{10}(x)$，其中 x>0
pow(x,y)	x^y。如果 x=0 且 y≤0，或者 x<0 且 y 不是整型数，将产生定义域错误
sqrt(x)	x 的平方根，其中 x≥0
ceil(x)	不小于 x 的最小整型数，其中 x 的类型为 double
floor(x)	不大于 x 的最大整型数，其中 x 的类型为 double
fabs(x)	x 的绝对值|x|
ldexp(x,n)	计算 $x.2^n$的值
frexp(x, int *ip)	把 x 分成一个在[1/2, 1]区间内的真分数和一个 2 的幂数。结果将返回真分数部分，并将幂数保存在*exp 中。如果 x 为 0，则这两部分均为 0
modf(x, double *ip)	把 J 分成整数和小数两部分，两部分的正负号均与 x 相同。该函数返回小数部分，整数部分保存在‘ ip 中
fmod(x,y)	求 JJy 的浮点余数，符号与 j 相同。如果 y 为 0，则结果与具体的实现相关

B.5 实用函数：<stdlib.h>

头文件<stdlib.h>中声明了一些执行数值转换、内存分配以及其它类似工作的函数。

double atof(const char *s)

atof 函数将字符串 s 转换为 double 类型。该函数等价于 strtod（s, (char**)NULL）。

int atoi(const char *s)

atoi 函数将字符串 s 转换为 int 类型。该函数等价于(int)strtol(s, (char**)NULL, 10)。

```
long atol(const char *s)
```

atol 函数将字符串 s 转换为 long 类型。该函数等价于 strtol(s, (char**)NULL, 10)。

```
double strtod(const char *s, char **endp)
```

strtod 函数将字符串 s 的前缀转换为 double 类型，并在转换时跳过 s 的前导空白符。除了 endp 为 NULL，否则该函数将把指向 s 中未转换部分（s 的后缀部分）的指针保存在*endp 中。如果结果上溢，则函数返回带有适当符号 HUGE_VAL；如果结果下溢，则返回 0。在这两种情况下，errno 都将被设置为 ERANGE。

```
long strtol(const char *s, char **endp, int base)
```

strtol 函数将字符串 s 的前缀转换为 long 类型，并在转换时跳过 s 的前导空白符。除非 endp 为 NULL，否则该函数将把指向 s 中未转换部分（s 的后辍部分）的指针保存在*endp 中。如果 base 的取值在 2～36 之间，则假定输入是以该数为基底的；如果 base 的取值为 0，则基底为八进制、十进制或十六进制。以 0 为前缀的是八进制，以 0x 或 0X 为前缀的是十六进制。无论在哪种情况下。字母均表示 10～base-1 之间的数字。如果 base 值是 16，则可以加上前导 0x 或 0X。如果结果上溢，则函数根据结果的符号返回 LONG_MAX 或 LONG_MIN，同时将 errno 的值设置为 ERANGE。

```
unsigned long strtoul(const char *s, char **endp, int base)
```

strtoul 函数的功能与 strtol 函数相同，但其结果为 unsigned long 类型，错误值为 ULONG_MAX。

```
int rand(void)
```

rand 函数产生一个 0～RAND_MAX 之间的伪随机整数。RAND_MAX 的取值至少为 32767。

```
void srand(unsigned int seed)
```

srand 函数将 seed 作为生成新的伪随机数序列的种子数。种子数 seed 的初值为 1。

```
void *calloc(size_t nobj, size_t size)
```

calloc 函数为由 nobj 个长度为 size 的对象组成的数组分配内存，并返回指向分配区域的指针；若无法满足要求，则返回 NULL。该空间的初始长度为 0 字节。

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

malloc 函数为长度为 size 的对象分配内存，并返回指向分配区域的指针；若无法满足要求，则返回 NULL。该函数不对分配的内存区域进行初始化。

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

realloc 函数将 p 指向的对象的长度修改为 size 个字节。如果新分配的内存比原内存大，则原内存的内容保持不变，增加的空间不进行初始化。如果新分配的内存比原内存小，则新分配内存单元不被初始化：realloc 函数返回指向新分配空间的指针；若无法满足要求，则返回 NULL。在这种情况下，原指针 p 指向的单元内容保持不变。

```
void free(void *p)
```

free 函数释放 p 指向的内存空间。当 p 的值为 NULL 时，该函数不执行任何操作。P 必须指向先前使用动态分配函数 malloe、realloc 或 calloc 分配的空间。

```
void abort(void)
```

abort 函数使程序非正常终止。其功能与 raise(SIGABRT)类似。

```
void exit(int status)
```

exit 函数使程序正常终止。atexit 函数的调用顺序与登记的顺序相反，这种情况下，所有已打开的文件缓冲区将被清洗，所有已打开的流将被关闭，控制也将返回给环境。status 的值如何返回给环境要视具体的实现而定，但 0 值表示终止成功。也可使用值 EXIT_SUCCESS 和 EXIT_FAILURE 作为返回值。

```
int atexit(void (*fcn)(void))
```

atexit 函数登记函数 fcn，该函数将在程序正常终止时被调用。如果登记失败，则返回非 0 值。

```
int system(const char *s)
```

system 函数将字符串 s 传递给执行环境。如果 s 的值为 NULL，并且有命令处理程序，则该函数返回非 0 值。如果 s 的值不是 NULL，则返回值与具体的实现有关。

```
char *getenv(const char *name)
```

getenv 函数返回与 name 有关的环境字符串。如果该字符串不存在，则返回 NULL。其细节与具体的实现有关。

```
void *bsearch(const void *key, const void *base,
    size_t n, size_t size,
    int (*cmp)(const void *keyval, const void *datum))
```

bsearch 函数在 base[0]...base[n-1]之间查找与*key 匹配的项。在函数 cmp 中，如果第一个参数（查找关键字）小于第二个参数（表项），它必须返回一个负值；如果第一个参数等于第二个参数，它必须返回零；如果第一个参数大于第二个参数，它必须返回一个正值。数组 base 中的项必须按升序排列。bsearch 函数返回一个指针，它指向一个匹配项，如果不存在匹配项，则返回 NULL。

```
void qsort(void *base, size_t n, size_t size,
    int (*cmp)(const void *, const void *))
```

qsort 函数对 base[0]...base[n-1]数组中的对象进行升序排序，数组中每个对象的长度为 size。比较函数 cmp 与 bsearch 函数中的描述相同。

```
int abs(int n)
```

abs 函数返回 int 类型参数 n 的绝对值。

```
long labs(long n)
```

labs 函数返回 long 类型参数 n 的绝对值。

```
div_t div(int num, int denom)
```

div 函数计算 num/denom 的商和余数，并把结果分别保存在结构类型 div_t 的两个 int 类型的成员 quot 和 rem 中。

```
ldiv_t ldiv(long num, long denom)
```

idiv 函数计算 num/denom 的商和余数，并把结果分别保存在结构类型 idiv_t 的两个

long 类型的成员 quot 和 rem 中。

B.6 诊断：<assert.h>

assert 宏用于为程序增加诊断功能。其形式如下：

```
void assert(int expression)
```

如果执行语句

```
assert(expression)
```

时，表达式的值为 0，则 assert 宏将在 stderr 中打印一条消息，比如：

```
Assertion failed: 表连式, file 源文件名, line 行号
```

打印消息后，该宏将调用 abort 终止程序的执行。其中的源文件名和行号来自于预处理器宏__FILE__及__LINE__。

如果定义了宏 NDEBUG，同时又包含了头文件<assert.h>，则 assert 宏将被忽略。

B.7 可变参数表：<stdarg.h>

头文件<stdarg.h>提供了遍历未知数目和类型的函数参数表的功能。

假定函数 f 带有可变数目的实际参数，lastarg 是它的最后一个命名的形式参数。那么，在函数 f 内声明一个类型为 va_list 的变量 ap，它将依次指向每个实际参数：

```
va_list ap;
```

在访问任何未命名的参数前，必须用 va—start 宏初始化 ap 一次：

```
va_start(va_list ap, lastarg);
```

此后，每次执行宏 va_arg 都将产生一个与下一个未命名的参数具有相同类型和数值的值，它同时还修改 ap，以使得下一次执行 va_arg 时返回下一个参数：

```
type va_arg(va_list ap, type);
```

在所有的参数处理完毕之后，且在退出函数 f 之前，必须调用宏 va_end 一次，如下所示：

```
void va_end(va_list ap);
```

B.8 非局部跳转：<setjmp.h>

头文件<setjmp.h>中的声明提供了一种不同于通常的函数调用和返回顺序的方式，特别是，它允许立即从一个深层嵌套的函数调用中返回。

```
int setjmp(jmp_buf env)
```

setjmp 宏将状态信息保存到 env 中，供 longjmp 使用。如果直接调用 setjmp，则返

回值为 0；如果是在 longjmp 中调用 setjmp，则返回值为非 0。Setjmp 只能用于某些上下文中，如用于 if 语句、switch 语句、循环语句的条件测试中以及一些简单的关系表达式中。例如：

```
     if (setjmp(env) == 0)
        /* get here on direct call */
     else
        /* get here by calling longjmp */
void longjmp(jmp_buf env, int val)
```

longjmp 通过最近一次调用 setjmp 时保存到 env 中的信息恢复状态，同时，程序重新恢复执行，其状态等同于 setjmp 宏调用刚刚执行完并返回非 0 值 val。包含 setjmp 宏调用的函数的执行必须还没有终止。除下列情况外，可访问对象的值同调用 longjmp 时的值相同：在调用 setjmp 宏后，如果调用 setjmp 宏的函数中的非 volatile 自动变量改变了，则它们将变成未定义状态。

B.9 信号：<signal.h>

头文件<signal.h>提供了一些处理程序运行期间引发的各种异常条件的功能，比如来源于外部的中断信号或程序执行错误引起的中断信号。

```
void (*signal(int sig, void (*handler)(int)))(int)
```

signal 决定了如何处理后续的信号。如果 handler 的值是 SIG_DFL，则采用由实现定义的默认行为；如果 handler 的值是 SIG_IGN，则忽略该信号；否则，调用 handler 指向的函数(以信号作为参数)。有效的信号包括：

SIGABRT	异常终止，例如由 abort 引起的终止
SIGFPE	算术运算出错，如除数为 0 或溢出
SIGILL	非法函数映像，如非法指令
SIGINT	用于交互式目的信号，如中断
SIGSEGV	非法存储器访问，如访问不存在的内存单元
SIGTERM	发送给程序的终止请求

对于特定的信号，signal 将返回 handler 的前一个值；如果出现错误，则返回值 SIG_ERR。

当随后碰到信号 sig 时，该信号将恢复为默认行为，随后调用信号处理程序，就好像由(*handler)(sig)调用的一样。信号处理程序返回后，程序将从信号发生的位置重新开始执行。

信号的初始状态由具体的实现定义。

```
int raise(int sig)
```

raise 向程序发送信号 sig。如果发送不成功，则返回一个非 0 值。

B.10 日期与时间函数：<time.h>

头文件<time.h>中声明了一些处理日期与时间的类型和函数。其中的一些函数用于处理当地时间，因为时区等原同，当地时间与日历时间可能不相同。clock_t 和 time_t 是两个

表示时间的算术类型，struct tm 用于保存日历时间的各个构成部分。结构 tm 中各成员的用途及取值范围如下所示：

int tm_sec;	从当前分钟开始经过的秒数(0, 61)
int tm_min;	从当前小时开始经过的分钟数(0, 59)
int tm_hour;	从午夜开始经过的小时数(0, 23)
int tm_mday;	当月的天数(1, 31)
int tm_mon;	从 1 月起经过的月数(0, 11)
int tm_year;	从 1900 年起经过的年数
int tm_wday;	从星期天起经过的天数(0, 6)
int tm_yday;	从 1 月 1 日起经过的天数(0, 365)
int tm_isdst;	夏令时标记

使用令时，tm_isdst 的值为正，否则为 0。如果该信息无效，则其值为负。

clock_t clock(void)

clock 函数返回程序开始执行后占用的处理器时间。如果无法获取处理器时间，则返回值为-1。clock()/CLOCKS_PER_SEC 是以秒为单位表示的时间。

time_t time(time_t *tp)

time 函数返回当前日历时间。如果无法获取日历时间，则返回值为-1。如果 tp 不是 NULL，则同时将返回值赋给*tp。

double difftime(time_t time2, time_t time1)

difftime 函数返回 time2-time1 的值（以秒为单位）。

time_t mktime(struct tm *tp)

mktime 函数将结构*tp 中的当地时间转换为与 time 表示方式相同的日历时间，结构中各成员的值位于上面所示范围之内。mktime 函数返回转换后得到的日历时间；如果该时间不能表示，则返回-1。

下面 4 个函数返回指向可被其它调用覆盖的静态对象的指针。

char *asctime(const struct tm *tp)

asctime 函数将结构*tp 中的时间转换为下列所示的字符串形式：

Sun Jan 3 15:14:13 1988\n\0

char *ctime(const time_t *tp)

ctime 函数将结构*tp 中的日历时间转换为当地时间。它等价于下列函数调用：

asctime(localtime(tp))

struct tm *gmtime(const time_t *tp)

gmtime 函数将*tp 中的日历时间转换为协调世界时（UTC）。如果无法获取 UTC，则该函数返回 NULL。函数名字 gmtime 有一定的历史意义。

struct tm *localtime(const time_t *tp)

localtime 函数将结构*tp 中的日历时间转换为当地时间。

```
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm
*tp)
```

strftime 函数根据 fmt 中的格式把结构*tp 中的日期与时间信息转换为指定的格式，并存储到 s 中，其中 fmt 类似于 printf 函数中的格式说明。普通字符（包括终结符'\0'）将复制到 s 中。每个%c 将按照下面描述的格式替换为与本地环境相适应的值。最多 smax 个字符写到 s 中。strftime 函数返回实际写到 s 中的字符数（不包括字符'\0'）；如果字符数多于 smax，该函数将返回值 0。

fmt 的转换说明及其含义如下所示：

%a	一星期中各天的缩写名
%A	一星期中各天的全名
%b	缩写的月份名
%B	月份全名
%c	当地时间和日期表示
%d	一个月中的某一天（01-31）
%H	小时（24 小时表示）（00-23）
%I	小时（12 小时表示）（01-12）
%j	一年中的各天（001—366）
%m	月份（01-12）
%M	分钟（00-59）
%p	与 AM 与 PM 相应的当地时间等价表示方法
%S	秒（00-61）
%U	一年中的星期序号（00-53，将星期日看作是每周的第一天）
%w	一周中的各天（0-6，星期日为 0）
%W	一年中的星期序号（00-53，将星期一看作是每周的第一天）
%x	当地日期表示
%X	当地时间表示
%y	不带世纪数目的年份（00-99）
%Y	带世纪数目的年份
%Z	时区名（如果有的话）
%%	%本身

B.11 与具体实现相关的限制：<limits.h>和<float.h>

头文件<limits.h>定义了一些表示整型大小的常量。以下所列的值是可接受的最小值，在实际系统中可以使用更大的值。

CHAR_BIT	8	char 类型的位数
CHAR_MAX	UCHAR_MAX 或 SCHAR_MAX	char 类型的最大值
CHAR_MIN	0 或 SCHAR_MIN	char 类型的最小值
INT_MAX	32767	int 类型的最大值
INT_MIN	-32767	int 类型的最小值
LONG_MAX	2147483647	long 类型的最大值
LONG_MIN	-2147483647	long 类型的最小值

SCHAR_MAX	+127	signed char 类型的最大值
SCHAR_MIN	-127	signed char 类型的最小值
SHRT_MAX	+32767	short 类型的最大值
SHRT_MIN	-32767	short 类型的最小值
UCHAR_MAX	255	unsigned char 类型的最大值
UINT_MAX	65535	unsigend int 类型的最大值
ULONG_MAX	4294967295	unsigned long 类型的最大值
USHRT_MAX	65535	unsigned short 类型的最大值

下表列出的名字是<float.h>的一个子集，它们是与浮点算术运算相关的一些常量。给出的每个值代表相应量的最小取值。各个实现可以定义适当的值。

FLT_RADIX	2	指数表示的基数，例如 2、16
FLT_ROUNDS		加法的浮点舍入模式
FLT_DIG	6	表示精度的十进制数字
FLT_EPSILON	1E-5	最小的数 x，x 满足：$1.0 + x \neq 1.0$
FLT_MANT_DIG		尾数中的数（以 FLT_RADIX 为基数）
FLT_MAX	1E+37	最大的浮点数
FLT_MAX_EXP		最大的数 n，n 满足 $FLT\_RADIX^{n}-1$ 仍是可表示的
FLT_MIN	1E-37	最小的规格化浮点数
FLT_MIN_EXP		最小的数 n，n 满足：10^{n} 是一个规格化数
DBL_DIG	10	表示精度的十进制数字
DBL_EPSILON	1E-9	最小的数 x，x 满足：$1.0 + x \neq 1.0$
DBL_MANT_DIG		尾数中的数（以 FLT_RADIX 为基数）
DBL_MAX	1E+37	最大的双精度浮点数
DBL_MAX_EXP		最大的数 n，n 满足 $FLT\_RADIX^{n}-1$ 仍是可表示的
DBL_MIN	1E-37	最小的规格化双精度浮点数
DBL_MIN_EXP		最小的数 n，n 满足：10^{n} 是一个规格化数

附录C 变更小结

自本书第 1 版出版以来，C 语言的定义已经发生了一些变化。几乎每次变化都是对原语言的一次扩充，同时每次扩充都是经过精心设计的，并保持了与现有版本的兼容性；其中的一些修改修正了原版本中的歧义性描述；某些修改是对已有版本的变更。许多新增功能都是随 AT&T 提供的编译器的文档一同发布的，并被此后的其它 C 编译器供应商采纳。前不久，ANSI 标准化协会在对 C 语言进行标准化时采纳了其中绝大部分的修改，并进行了其它一些重要修正。甚至在正式的 C 标准发布之前，ANSI 的报告就已经被一些编译器提供商部分地先期采用了。

本附录总结了本书第 1 版定义的 C 语言与 ANSI 新标准之间的差别。我们在这里仅讨论语言本身，不涉及环境和库。尽管环境和库也是标准的重要组成部分，但它们与第 1 版几乎无可比之处，因为第 1 版并没有试图规定一个环境或库。

- 与第 1 版相比，标准 C 中关于预处理的定义更加细致，并进行了扩充：明确以记号为基础；增加了连接记号的运算符（##）和生成字符串的运算符（#）；增加了新的控制指令（如#elif 和#pragma）；明确允许使用相同记号序列重新声明宏；字符串中的形式参数不再被替换。允许在任何地方使用反斜杠字“\”进行行的连接，而不仅仅限于在字符串和宏定义中。详细信息参见 A.12 节。
- 所有内部标识符的最小有效长度增加为 31 个字符；具有外部连接的标识符的最小有效长度仍然为 6 个字符（很多实现中允许更长的标识符）。
- 通过双问号”??”引入的三字符序列可以表示某些字符集中缺少的字符。定义了#、\、^、[、]、{、}、|、~等转义字符，参见 A.12.1 节。注意，三字符序列的引入可能会改变包含“??”的字符串的含义。
- 引入了一些新关键字（void、const、volatile、signed 和 enum）。关键字 entry 将不再使用。
- 定义了字符常量和字符串字面值中使用的新转义字符序列。如果\及其后字符构成的不是转义序列，则其结果是未定义的。参见 A.2.5 节。
- 所有人都喜欢的一个小变化：8 和 9 不用作八进制数字。
- 新标准引入了更大的后缀集合，使得常量的类型更加明确：U 或 L 用于整型，F 或 L 用于浮点数。它同时也细化了无后缀常量类型的相关规则（参见 A.2.5 节）。
- 相邻的字符串将被连接在一起。
- 提供了宽字符字符串字面值和字符常量的表示方法，参见 A.2.6 节。
- 与其它类型一样，对字符类型也可以使用关键字 signed 或 unsigned 显式声明为带符号类型或无符号类型。放弃了将 long float 作为 double 的同义词这种独特的用法，但可以用 long double 声明更高精度的浮点数。
- 有段时间，C 语言中可以使用 unsigned char 类型。新标准引入了关键字 signed，用来显式表示字符和其它整型对象的符号。
- 很多编译器在几年前就实现了 void 类型。新标准引入了 void *类型，并作为一种通用指针类型；在此之前 char *扮演着这一角色。同时，明确地规定了在不进行强制类型转换的情况下，指针与整型之间以及不同类型的指针之间运算的规则。

- 新标准明确指定了算术类型取值范围的最小值，并在两个头文件（<limits.h>和<float.h>）中给出了各种特定实现的特性。
- 新增加的枚举类型是第 1 版中所没有的。
- 标准采用了 C++中的类型限定符的概念，如 const（参见 A.8.2 节）。
- 字符串不再是可以修改的，因此可以放在只读内存区中。
- 修改了“普通算术类型转换”，特别地，“整型总是转换为 unsigned 类型，浮点数总是转换为 double 类型”已更改为“提升到最小的足够大的类型”。参见 A.6.5 节。
- 旧的赋值类运算符（如=+）已不再使用。同时，赋值类运算符现在是单个记号；而在第 1 版中，它们是两个记号，中间可以用空白符分开。
- 在编译器中，不再将数学上可结合的运算符当做计算上也是可结合的。
- 为了保持与一元运算符-的对称，引入了一元运算符+。
- 指向函数的指针可以作为函数的标志符，而不需要显式的*运算符。参见 A.7.3 节。
- 结构可以被赋值、传递给函数以及被函数返回。
- 允许对数组应用地址运算符，其结果为指向数组的指针。
- 在第 1 版中，sizeof 运算符的结果类型为 int，但随后很多编译器的实现将此结果作为 unsigned 类型。标准明确了该运算符的结果类型与具体的实现有关，但要求将其类型 size_t 在标准头文件<stddef.h>中定义。关于两个指针的差的结果类型（ptrdiff_t）也有类似的变化。参见 A.7.4 节与 A.7.7 节。
- 地址运算符&不可应用于声明为 register 的对象，即使具体的实现未将这种对象存放在寄存器中也不允许使用地址运算符。
- 移位表达式的类型是其左操作数的类型，右操作数不能提升结果类型。参见 A.7.8 节。
- 标准允许创建一个指向数组最后一个元素的下一个位置的指针，并允许对其进行算术和关系运算。参见 A.7.7 节。
- 标准（借鉴于 C++）引入了函数原型声明的表示法，函数原型中可以声明变元的类型。同时，标准中还规定了显式声明带可变变元表的函数的方法，并提供了一种被认可的处理可变形式参数表的方法。参见 A.7.3 节、A.8.6 节和 B.7 节。旧式声明的函数仍然可以使用，但有一定限制。
- 标准禁止空声明，即没有声明符，且没有至少声明一个结构，联合或枚举的声明。另一方面，仅仅只带结构标记或联合标记的声明是对该标记的重新声明，即使该标记声明在外层作用域中也是这样。
- 禁止没有任何说明符或限定符（只是一个空的声明符）的外部数据说明。
- 在某些实现中，如果内层程序块中包含一个 extern 声明，则该声明对该文件的其它部分可见。ANSI 标准明确规定，这种声明的作用域仅为该程序块。
- 形式参数的作用域扩展到函数的复合语句中，因此，函数中最顶层的变量声明不能与形式参数冲突。
- 标识符的名字空间有一些变化。ANSI 标准将所有的标号放在个单独的名字空间中，同时也为标号引入了一个单独的名字空间，参见 A.1.1 节。结构或联合的成员名将与其所属的结构或联合相关联（这已经是许多实现的共同做法了）。
- 联合可以进行初始化，初值引用其第一个成员。
- 自动结构、联合和数组可以进行初始化，但有一些限制。
- 显式指定长度的字符数组可以用与此长度相同的字符串字面值初始化（不包括字符\0）。
- switch 语句的控制表达式和 case 标号可以是任意整型。