C程序设计语言

汪帆

兰州大学　数学与统计学院

May 16, 2008

导言

C语言是一种通用的程序设计语言，它同UNIX系统之间有非常密切的联系. 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”.

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语言的影响间接的来自于B语言，它是Ken Thompson为第一个UNIX系统与1970年开发的.

C语言最初是由Dennis Ritchie为UNIX操作系统设计的，并且在DEC PDP-11计算机上实现.

入门

学习一门新程序设计语言的唯一途径就是使用它编写程序.
打印出如下内容:
hello, world

在C语言中，使用如下程序:

```
#include<stdio.h>

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

变量与算术表达式

我们来看下一个程序，使用公式$C = (5/9)(F - 32)$打印温度对照表：

0	-17
20	-6
40	4
60	15
80	26
100	37
120	48
140	60
160	71
180	82
200	93

```
#include<stdio.h>

 /* print the Fahrenheit scale
 and the Centigrade scale */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;     /*temperature's floor level */
    upper = 200;   /*temperature's upper limit */
    step = 20;     /*step */

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

一. 包含在/*与*/之间的部分称为注释.
二. 在C语言中, 所有的变量都必须先声明后使用. 声明通常放在函数起始处, 在任何可执行语句之前. 声明用于说明变量的属性, 它由一个类型名变量表组成.
三. 温度表中的各行计算方式相同, 因此可以用循环语句重复输出各行.
四. 在C语言和许多其他语言中, 整数除法操作将执行舍位.
五. printf 是一个通用输出格式化函数. 必须包含 stdio.h 库文件.

上面的程序中存在两个问题:
一. 输出的数不是右对齐的, 所以输出结果不美观.
二. 我们使用的是整型算术运算, 所以得到的结果精度不高.
第一个问题容易解决, 如果在 printf 函数中指明打印宽度, 则打印的数字会在打印区域内右对齐. 例如:

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

这样, fahr 的占3个字符宽, celsius 的值占6个字符宽.

为了得到更精确的结果，应该用浮点算术运算代替上面的整型算术运算.

```
#include<stdio.h>

 /* print the Fahrenheit scale
  and the Centigrade scale */
main()
{
    float fahr, celsius;
    int lower, upper, step;
    lower = 0;     /*temperature's floor level */
    upper = 200;  /*temperature's upper limit */
    step = 20;     /*step */

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

printf 中的转换说明%3.0f表示待打印的浮点数至少有3个字符宽，且不带小数点和小数部分；%6.1f表明另一个待打印的数至少占6个字符宽，且小数点后面有1位数字.

%d　按照十进制整型数打印

%6d 按照十进制整型数打印，至少6个字符宽

%f　按照浮点数打印

%6f　按照浮点数打印，至少6个字符宽

%.2f 按照浮点数打印，小数点后面有两位

%6.2f 按照浮点数打印，至少6个字符宽，小数点后面有两位

此外，%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；%%表示百分号本身.

for语句

下面这段代码也可以实现上面的温度转换功能:

```
#include<stdio.h>

 /* print the Fahrenheit scale
 and the Centigrade scale */
main()
{
     int fahr;
     for (fahr = 0; fahr <= 200; fahr = fahr +20)
          printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

练习：修改温度转换程序，要求以逆序（即按照从200度到0度的顺序）打印温度转换表.

符号常量

在程序中使用200, 20等类似的"幻数"并不是一个好习惯，它们无法向阅读程序的人提供什么信息，而且使程序的修改变得很困难. 处理的一种方法是赋予它们有意义的名字.
*#define*指令可以把符合常量定义为一个特定的字符串:
#define 名字 替换文本
例如下面的程序:

```
#include<stdio.h>

#define LOWER 0       /*temperature's floor level */
#define UPPER 300     /*temperature's upper limit */
#define STEP  20      /*step */

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

字符的输入输出

标准库提供的输入和输出模型非常简单. 无论文本从何处输入, 输出到何处, 其输入和输出都是按照字符流的方式处理. 文本流是由多行字符构成的字符序列, 而每行字符则由 0 个或多个字符组成, 行末是一个换行符.

标准库提供了一次读或写一个字符的函数, 其中最简单的是 getchar 和 putchar 两个函数. 每次调用时, getchar 函数从文本流中读入下一个输入字符, 并将其作为结果值返回. 即在执行语句

```
c = getchar()
```

之后, 变量 c 中包含输入流中的下一个字符. 通常是键盘输入的.

每次调用 putchar 函数将打印一个字符. 例如, 语句

```
putchar(c)
```

将把整型变量 c 的内容以字符的形式打印出来, 通常在屏幕上.

1. 文件复制

借助与 getchar 和 putchar 函数，可以在不了解其他输入和输出知识的情况下编写出很有用的代码. 最简单的例子就是把输入一次一个字符的复制到输出，其基本思想如下:

读一个字符
while(该字符不是文件结束的标志)
    输出刚读入的字符
    读下一个字符

将上述基本思想转换为 C 语言程序为:

```
#include<stdio.h>

main()
{
    int c;

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

其中，关系运算符!=表示”不等于”.

在 C 语言中，类似与 $c = getchar()$ 之类的赋值操作是一个表达式，并且具有一个值，即赋值后左边变量保存的值. 所以赋值可以作为更大的表达式的一部分出现. 例如:

```
#include<stdio.h>

main()
{
    int c;

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

练习 1: 验证表达式 $getchar() != EOF$ 的值是 0 还是 1 .
练习 2: 编写一个打印 EOF 值的程序.

2. 字符计数

下面的程序对字符进行计数，它与上面的复制程序类似.

```
#include<stdio.h>

main()
{
    long nc;

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

使用 double (双精度浮点数)类型可以处理更大的数字. 下面不使用 while 循环, 而用 for 循环编写上面的程序:

```
#include<stdio.h>

main()
{
    double nc;

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

3. 行计数

下面的程序用来统计输入的行数，前面说到标准库保证输入文本流以行序列的形式出现，每一行均以换行符结尾. 所以，统计行数就是统计换行符的个数.

```
#include<stdio.h>
/* 统计输入中的行数*/
main()
{
    int c, nl;

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

练习: 编写一个统计空格，制表符与换行符个数的程序.

数组

下面一个程序，以统计各个数字，空白符(包括空格符，制表符及换行符)以及其他字符出现的次数.
所以的输入字符可以分为 12 类，因此可以用一个数组存放各个数字出现的次数.

```
#include<stdio.h>
/* 统计各个数字，空白符及其他字符出现的次数*/
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);
}
```

该程序中的声明语句

int ndigit [10];

将变量 ndigit 声明为由 10 个整型数组成的数组. 在 C 语言中, 数组下标总是从 0 开始, 因此该数组的 10 个元素分别为$ndigit[0], ndigit[1], \cdots, ndigit[9]$, 这可以通过初始化和打印数组的两个 for 循环语句反映出来.
数组下标可以是任何整型表达式, 包括整型变量, 例如i 以及整型常量.

函数

函数为计算的封装提供了一种简便的方法，此后使用函数时不需要考虑它是如何实现的. 使用设计正确的函数，程序员无需考虑功能是如何实现的，而只需要知道它具有哪些功能就足够了.

到目前为止，我们所使用的函数都是函数库中的函数，如 printf, getchar, putchar 等. 下面我们编写一个求幂的函数 power(m,n) . power(m,n) 用来计算整数 m 的 n 次幂，其中 n 是正整数.

```
#include<stdio.h>
int power(int m, int n);

/* 测试power函数*/
int main()
{
    int i;
    for(i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power函数: 求底数的n次幂, 其中n >= 0 */
int power(int base, int n)
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

函数定义的一般形式为:

```
返回值类型函数名(0个或多个参数声明)
{
        声明部分
        语句序列
}
```

练习: 重新编写温度转换程序, 使用函数实现温度转换计算.

参数—传值调用

- 在 C 语言中，所有函数参数都是”通过值”传递的. 即传递给被调用函数的参数值存放在临时变量中，而不是存放在原来的变量中.
- 最主要的问题在于，在 C 语言中，被调用函数不能直接修改主调函数中变量的值，而只能修改其私有的临时副本的值.
- 传值调用的利大与弊. 在被调用函数中，参数可以看作是便于初始化的局部变量，因此额外使用的变量更少，这样程序可以更紧凑简结.

下面的这个 power 函数利用了这一性质:

```
/* 另一版本的power函数*/
int power(int base, int n)
{
    int p;

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

在需要的时候，也可以让函数能够修改主调函数的变量．这种情况下，调用者需要向被调用者提供待设置值的变量的内存地址(技术上说，地址就是指向变量的指针)，而被调用函数则需要将对应的参数声明为指针类型，并通过它间接访问变量．我们将在后面再讨论指针．

如果是数组参数，情况就有所不同了．当把数组名作为参数时，传递给函数的值是数组的起始元素的位置或地址—它并不复制数组元素本身．在被调用函数中，可以通过数组下标访问或修改数组元素的值．

字符数组

字符数组是 C 语言中最常用的数组类型. 下面这个程序用来说明字符数组以及操作字符数组的函数的用法. 该程序读入一组文本行, 并且把最长的文本行打印出来. 这个算法的基本框架非常简单:

```
while (还有未处理的行)
        if (该行比已处理的最长行还要长)
            保存该行
            保存该行的长度
打印最长的行
```

从上面可以看出, 程序很自然的分成了若干片断, 分别用于读入新行, 测试读入的行, 保存该行, 其余部分则控制这一过程.

- 首先，我们编写一个独立的函数 getline，它读取输入的下一行. 在函数设计上，至少 getline 函数应该在读到文件末尾时返回一个信号；更为有用的设计是它能够在读入文本行时返回该行的长度，而在遇到文件结束符时返回 0. 由于 0 不是有效的行长度,因此可以作为文件结束的返回值. 每一行至少包括一个字符，只包含换行符的行，其长度为 1.
- 当发现某个新读入的行比以前读入的行还要长时，就需要把该行保存起来. 也就是说，我们需要用另一个函数 copy 把新行复制到一个安全的位置.
- 最后，需要在主函数 main 中控制 getline 和 copy 这两个函数.

```
#include<stdio.h>
#define MAXLINE 1000    /* 允许的输入行的最大长度*/

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

int main()
{
    int len;        /* 当前行长度*/
    int max;        /* 目前为止发现的最长行的长度*/
    char line[MAXLINE];        /* 当前的输入行*/
    char longest[MAXLINE];     /* 用来保存最长的行*/
    max = 0;
    while((len = getline(line, MAXLINE)) > 0)
        if(len > max){
        max = len;
        copy(longest, line);}
    if(max > 0)   /* 存在最长的行*/
        printf("%s", longest);
    return 0;
}
```

```
/* getline 函数: 将一行读入到 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 函数: 将 from 复制到 to ; 这里假定 to 足够大*/
void copy(char to[], char from[])
{
    int i;

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

程序的开始对 getline 和 copy 这两个函数进行了声明，这里假定它们都存放在同一个文件中.

类型，运算符与表达式

- 变量和常量是程序处理的两种基本数据对象.
- 声明语句说明变量的名字以及类型，也可以指定变量的初值.
- 运算符指定将要进行的操作.
- 表达式则把变量与常量组合起来生成新的值.
- 对象的类型决定该对象可取值的集合以及可以对该对象执行的操作.

在这一章中我们要详细讲述这些内容.

变量名

对变量的命名和符号常量的命名存在一些限制条件. 名字是由字母和数字组成的序列, 但其第一个字符必须为字母. 下划线"_"被看作是字母, 通常用于命名较长的变量名, 以提高其可读性.

因为库例程的名字通常以下划线开头, 因此变量名不要以下划线开头.

大写字母与小写字母是有区别的. 在传统的C语言用法中, 变量名使用小写字母, 符号常量名全部使用大写字母.

类似与if, else, int, float等关键字是保留给语言本身使用的，不能把它用作变量名. 所有关键字中的字符都必须小写.

选择的变量名要能够尽量从字面上表达变量的用途，这样做不容易引起混淆. 局部变量一般使用较短的变量名（尤其是循环控制变量），外部变量使用较长的名字.

数据类型及长度

C语言只提供了以下几种基本数据类型:
char 字符型，占用一个字节，可以存放本地字符集中的一个字符
int 整型，通常反映了所用机器中整数的最自然长度
float 单精度浮点型
double 双精度浮点型
此外，还可以在这些基本数据类型的前面加上一些限定符.
short 与 long 两个限定符用于限定整型:

short int sh;
long int counter;

在上述这种类型的声明中，关键字 int 可以省略.

int通常代表特定机器中整数的自然长度. short类型通常为16位, long类型通常位32位, int类型可以为16位或32位.

各种编译器可以根据硬件特性自主选择合适的类型长度, 但是要遵循以下的限制: short与int类型至少为16位, 而long类型至少为32位, 并且short类型不得长于int类型, 而int类型不得长于long类型.

类型限定符signed与unsigned可用于限定char类型或任何整数. unsigned类型的数总是正值或0, 并遵守算术模2^n定律, 其中n是该类型占用的位数.

例如, 如果char对象占用8位, 那么unsigned char类型变量的取值范围位0 ~ 255, 而signed char类型变量的取值范围为$-128 \sim 127$.

long double 类型表示高精度的浮点数.

有关这些类型长度的定义的符号常量以及其他与机器和编译器有关的属性可以在标准头文件$< limits.h >$与$< float.h >$中找到.

常量

- 类似于 1234 的整数常量属于 int 类型. long 类型的常量以字母 l 或 L 结尾, 如 12345678L. 如果一个整数太大以至于无法用 int 类型表示时, 也将用 long 类型处理. 无符号常量以字母 u 或 U 结尾. 后缀 ul 或 UL 表明是 unsigned long 类型.
- 浮点数常量中包含一个小数点（如 123.4）或一个指数（如 1e-2）. 也可以两者都有. 没有后缀的浮点数常量为 double 类型. 后缀 f 或 F 表示 float 类型, 而后缀 l 或 L 表示 long double 类型.

- 整型数除了用十进制表示外, 还可以用八进制或十六进制表示. 带前缀 0 的整型常量表示它为八进制形式; 前缀为 0x 或 0X ,则表示它为十六进制形式.
- 八进制和十六进制的常量也可以用后缀 L 表示 long 类型, 使用后缀 U 表示 unsigned 类型. 例如, 0XFUL 是一个 unsigned long 类型（无符号长类型）的常量, 其值等于十进制数 15.

一个字符常量就是一个整数, 书写时将一个字符括在单引号中, 如 'x'. 字符在机器字符集中的数值就是字符常量的值. 例如, 在 ASCII 字符集中, 字符 '0' 的值为 48, 它与数值 0 没有关系.

字符常量一般用来与其他字符进行比较, 但也可以像其他整数一样参与整数运算.

常量表达式是仅仅只包含常量的表达式. 这种表达式在编译时求值, 而不在运行时求值, 它可以出现在常量可以出现的任何位置.

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面值, 是用双引号引起来的 0 个或者多个字符组成的字符序列. 例如:

"I␣am␣a␣string!"

或

"" /* 空字符串*/

都是字符串. 双引号不是字符串的一部分, 它只用于限定字符串.

- 实际上, 字符串常量就是一个字符数组 (专门存放字符的数组). 在字符串内部表示使用一个空字符 $'\backslash 0'$ 作为串的结尾, 因此, 存储字符串的物理存储单元比括在双引号中的字符数多一个.
- C语言对字符串的长度没有限制, 但是程序必须扫描完整个字符串后才能确定字符串的长度.
- 标准头文件$< string.h >$中声明了标准库函数 strlen(s) 可以返回字符串参数 s 的长度, 但长度不包括末尾的 $'\backslash 0'$.

我们可以自己设计实现 strlen 函数的一个版本, 例如:

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

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

要注意字符常量与仅包含一个字符的字符串之间的区别: 'x' 与 "x" 是不同的. 前者是一个整数, 其值是字母 x 在机器字符集中对应的数值; 后者是一个包含一个字符 (即字母 x) 以及一个结束符 '\0' 的字符数组.

枚举常量是另外一种类型的常量. 枚举是一个常量整型值的列表, 例如:

enum boolean {NO, YES};

在没有显式的说明下, enum 类型中的第一个枚举名的值为 0 , 第二个为 1 , 依此类推.
如果只指定了部分枚举名的值, 那么未指定的枚举名的值将依照最后一个指定值向后递增, 如:

```
enum escapes { Jan = 1, Feb, Mar, Apr, May, Jun,
               Jul,  Aug, Sep, Oct, Nov, Dec };
```

不同枚举中的名字必须互不相同. 同一枚举中不同名字可以具有相同的值.

声明

- 所有的变量都必须先声明后使用，一个声明指定一种变量类型，后面所带的变量表可以包含一个或多个变量.
- 也可以在声明的同时对变量进行初始化. 在声明中，如果变量名的后面紧跟一个表达式，该表达式就充当对变量进行初始化的初始化表达式. 例如:

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

任何变量的声明都可以使用 const 限定符限定. 该限定符指定变量的值不能被修改. 对数组而言, const 限定符指定数组所有的值都不能被修改:

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

const 限定符也可以配合数组参数使用, 它表明函数不能修改数组元素的值:

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

算术运算符

- 二元算术运算符包括: $+$, $-$, $*$, $/$, $\%$(取模运算符).
- 整数除法会截断结果中的小数部分.
- 表达式 $x \,\%\, y$ 的结果是 x 除以 y 的余数, 当 x 能被 y 整除时, 其值为 0 . 例如, 如果某一年的年份能被 4 整除但不能被 100 整除, 那么这一年就是闰年, 另外, 能够被 400 整除的年份也是闰年. 所以我们可以用下面的语句判断闰年:

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n",year);
else
    printf("%d is not a leap year\n",year);
```

- 取模运算符 % 不能应用于 float 或 double 类型.
- 在有负操作数的情况下，整数除法截取的方向以及取模运算结果的符号取决于具体机器的实现.
- 二元运算符 + 和 − 具有相同的运算优先级，它们的优先级比运算符 *，/ 和 % 的优先级低，而运算符 *，/ 和 % 又比一元运算符 + 和 − 的优先级低. 算术运算符采用从左到右的结合规则.

关系运算符与逻辑运算符

关系运算符包括下列几个运算符:

$$>,\quad >=,\quad <,\quad <=$$

它们具有相同的优先级. 优先级仅次于它们的是相等性运算符:

$$==,\quad !=$$

关系运算符的优先级比算术运算符低.
因此, 表达式 $i < lim - 1$ 等价于 $i < (lim - 1)$.

逻辑运算符 &&(逻辑与运算) 与 ||(逻辑或运算) 有一些较为特殊的属性. 由 && 和 || 连接的表达式按从左到右的顺序进行求值, 并且, 在知道结果值为真或假后立即停止计算.
例如, 下面的语句在功能上与导言中的输入函数 getline 中的循环语句等价的循环语句:

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

运算符 && 的优先级比 || 的优先级高, 但两者都比关系运算符和相等性运算符的优先级低.
根据定义, 在关系表达式或逻辑表达式中, 如果关系为真, 则表达式的结果为数值 1 ; 如果为假, 则结果值为数值 0 .

逻辑非运算符！的作用是将非 0 操作数转换为 0，将操作数 0 转换为 1．该运算符通常用于如下的类似的结构中:

$$if(!valid)$$

一般不采用下列形式:

$$if(valid == 0)$$

练习: 在不使用运算符 && 或 || 的条件下编写一个与上面的 for 循环语句等价的循环语句.

类型转换

- 当一个运算符的几个操作数类型不同时，就需要通过一些规则把它们转换为某种共同的类型．一般来说，自动转换是指把”比较窄的”操作数转换为”比较宽的”操作数，并且不丢失信息的转换．
- 不允许使用无意义的表达式，例如，不能把一个 float 类型的表达式作为数组的下标．
- 对于可能导致信息丢失的表达式，编译器可能会给出警告信息，比如把较长的的整型值赋值给较短的整型变量．

因为 char 类型是最小的整型，所以在算术表达式中可以自由使用 char 类型变量. 下面的函数 atoi，将一串数字（存放在字符数组中）转换为相应的数值:

```
/* atoi 函数: 将字符串 s 转换为相应的整型数*/
int atoi(char s[])
{
    int i, n;

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

函数 lower 是将 char 类型转换成 int 类型的另一个例子, 它将 ASCII 字符集中的字符映射到小写字母. 如果待转换的字符不是大写字母, lower 函数将返回字符本身.

```
/* lower 函数: 将字符 c 转换为小写形式;只对 ASCII 字符集有效*/
int lower(int c)
{
    if(c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

当关系表达式 (如 $i > j$) 以及由 && 或 || 连接的逻辑表达式的判定结果为真时, 表达式的值为 1 ; 当判定结果为假时, 表达式的值为 0 . 因此对于赋值语句

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

来说, 当 c 为数字时, d 的值为 1 , 否则 d 的值为 0 . 在 if, while, for 等循环语句的测试条件部分, "真"就意味着"非 0 ", 这两者之间没有区别.

在 C 语言中, 很多情况下会进行隐式的算术类型转换. 一般来说, 如果二元运算符的两个操作数具有不同的类型, 那么在进行运算之前要把"较低"的类型提升为"较高"的类型. 运算结果为较高的类型.

在没有 unsigned 类型的操作数, 则只要使用下面的非正式规则就可以了:

- 如果其中一个操作数的类型为 long double，则将另一个操作数转换为 long double 类型.
- 如果其中一个操作数的类型为 double，则将另一个操作数转换为 double 类型.
- 如果其中一个操作数的类型为 float，则将另一个操作数转换为 float 类型.
- 将 char 与 short 类型的操作数转换为 int 类型.
- 如果其中一个操作数的类型为 long，则将另一个操作数也转换为 long 类型.

赋值时也要进行类型转换. 赋值运算符右边的值需要转换为左边变量的类型, 左边变量的类型即赋值表达式结果的类型. 当把较长的整数转换为较短的整数或 char 类型时, 超出的高位部分将被丢弃.

```
int i;
char c;

i = c;
c = i;
```

上面的语句执行后, c 的值将不变. 但是将上面的两个赋值语句次序颠倒一下, 则执行后可能会丢失信息.

在任何表达式中都可以使用一个称为**强制类型转换**的一元运算符强制进行显式类型转换. 在下面的语句中, **表达式**将按照上述的转换规则转换为**类型名**指定的类型:

(类型名) 表达式

在上述语句中, 表达式首先被赋值给类型名指定的类型的某个变量, 然后再用该变量替换上述的整条语句.
例如, 库函数 sqrt 的参数为 double 类型 (sqrt 在$< math.h >$中声明) . 因此,如果 n 是整数, 可以使用

sqrt((**double**) n)

在把 n 传递给函数 sqrt 之前先将其转换为 double 类型.

标准库中包含一个可移值的实现伪随机数发生器函数 rand 以及一个初始化种子数的函数 srand . 前一个函数 rand 使用了强制类型转换.

```
unsigned long int next = 1;

/* rand 函数: 返回取值在 0 到 32767 之间的伪随机数*/
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}

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

自增运算符与自减运算符

C语言中有两个用于变量递增与递减的特殊运算符. 自增运算符++使其操作数递增 1 , 自减运算符−−使其操作数递减 1 . 例如:

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

++与−−这两个运算符特殊的地方主要表现在: 它们既可以用作前缀运算符 (用在变量前面, 如 ++n) , 也可以用作后缀运算符 (用在变量后面, 如 n++) .
但是, 这两者之间是有区别的.

表达式 ++n 先将 n 的值递增 1，然后再使用变量 n 的值，而表达式 n++ 先使用变量 n 的值，然后再将 n 的值递增 1．即对于使用变量 n 的上下文来说，++n 和 n++ 的效果是不同的．如果 n 的值是 5，那么

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

执行后的结果是将 x 的值置为 5，而

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

将 x 的值置为 6．这两条语句执行完后，变量 n 的值都是 6．自增与自减运算符只能用于变量类似与表达式$(i+j)++$是非法的．

在不需要使用任何具体值且仅需要递增变量的情况下，前缀方式和后缀方式效果相同. 但某些情况下需要酌情考虑.
例如: 考虑下面的函数 squeeze(s, c) ，它将删除字符串 s 中出现的所有字符 c :

```
/* squeeze 函数: 从字符串 s 中删除字符 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[j];
    s[j] = '\0';
}
```

下面再看一个例子, 考虑标准函数 strcat(s, t) , 它将字符串 t 连接到字符串 s 的尾部. 函数 strcat 假定字符串 s 有足够的空间保存结果. 下面这个函数没有任何返回值 (在标准库中, 该函数返回一个指向新字符串的指针) :

```
/* strcat 函数: 将字符串 t 连接到字符串 s 的尾部; s 有足够的空间*/
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0')                    /* 判断是否为字符串 s 的尾部*/
        i++;
    while ((s[i++] = t[j++]) != '\0')       /* 拷贝 t */
            ;
}
```

在将 t 中的字符逐个拷贝到 s 的尾部时, 变量 i 和 j 使用的都是后缀运算符++, 从而保证在循环过程中 i 与 j 均指向下一个位置.

按位运算符

C 语言提供了 6 个位操作运算符. 这些运算符只能作用于整型操作数, 即只能作用于带符号或无符号的 char, short, int 与 long 类型:

- & 按位与(AND).
- | 按位或(OR).
- ^ 按位异或(XOR)
- << 左移
- >> 右移
- ~ 按位求反(一元运算符)

- 按位与运算符 & 经常用于屏蔽某些二进制位，例如

  n = n & 0177;

  该语句将 n 中除 7 个低二进制位外的其他各位均置为 0 .
- 按位或运算符 | 常用于将某些二进制位置为1，例如

  x = x | SET_ON;

  该语句中将 x 中对应与 SET_ON 中为 1 的那些二进制位置为 1 .
- 按位异或运算符 ^ 当两个操作数的对应位不相同时将该位设置为 1 ，否则，将该位设置为 0 .

- 要注意位运算符 $\&$, $|$ 同逻辑运算符 $\&\&$, $\|$ 区分开来, 后者用于从左至右求表达式的真值. 例如, 如果 x 的值为 1 , y 的值为 2 , 那么, $x\&y$的结果为 0 , 而$x\&\&y$的结果为 1 .
- 移位运算符 $<<$ 与 $>>$ 分别用于将运算符的左操作数左移与右移, 移动的位数则由右操作数指定 (右操作数的值必须是负值) . 因此, 表达式 $x << 2$ 表示将 x 的值左位移 2 位, 右边空出的 2 位用 0 填补, 该表达式等价于对左操作数乘以 4 .
- 在对 unsigned 类型的值进行右位移时, 左边空出的部分将用 0 填补; 当对 signed 类型的值进行右位移时, 某些机器对左边空出的位用符号位填补 (算术移位) , 而有些机器对左边空出的部分用 0 填补 (逻辑移位) .

- 一元运算符 ~ 用于求整数的二进制反码，即分别将操作数各二进制位上的 1 变为 0，0 变为 1．例如：

  x = x & ~077

  将把 x 的最后六位设置为 0．

下面我们来看一个函数 getbits(x, p, n)，它返回 x 中从右边数第 p 位开始向右数 n 位的字段．这里假定最右边一位是第 0 位，n 与 p 都是合理的正值．

```
/* getbits 函数: 返回 x 中从第 p 位开始的 n 位*/
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1−n)) & ~(~0 << n);
}
```

赋值运算符与表达式

在赋值表达式中，如果表达式左边的变量重复出现在表达式的右边，如：

$$\mathrm{i} = \mathrm{i} + 2$$

则可以将这种表达方式缩写为下列形式：

$$\mathrm{i} \mathrel{+}= 2$$

其中的运算符 $+=$ 称为**赋值运算符**.

大多数二元运算符（即由左右两个操作数的运算符，比如 +）都有一个相应的赋值运算符 $op=$，其中，op 可以是下面这些运算符之一:

$+ \quad - \quad * \quad / \quad \% \quad << \quad >> \quad \& \quad \hat{} \quad |$

如果 $expr1$ 与 $expr2$ 是两个表达式，那么

expr1 op= expr2

就等价与:

expr1 = (expr1) op (expr2)

下面我们看一个例子：这个函数 bitcount 统计其整型参数的值为 1 的二进制位的个数.

```
/* bitcount 函数: 统计 x 中值为 1 的二进制位数*/
int bitcount(unsigned x)
{
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

注意，我们将 x 声明为无符号类型是为了保证将 x 右移时，无论该程序在什么机器上运行，左边空出的位都用 0（而不是符号位）填补.

条件表达式

下面这段语句:

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

用于求 a 与 b 中的最大值，并将结果保存到 z 中．我们可以用条件表达式（使用三元运算符"? :"）来编写上面这段代码.

在表达式

```
expr1 ? expr2 : expr3
```

中，首先计算 expr1，如果其值不等于 0（为真），则计算 expr2 的值，并以它作为条件表达式的值，否则计算 expr3 的值，并以该值作为条件表达式的值. 因此，以上语句可以改写为:

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

条件表达式中的第一个表达式两边的括号并不是必须的，这是因为条件运算符 "? :" 的优先级非常低，仅高于赋值运算符.

采用条件表达式可以编写出很简洁的代码. 例如, 下面的这个循环语句打印一个数组的 n 个元素, 每行打印 10 个元素, 每列之间用一个空格隔开, 每行用一个换行符结束 (包括最后一行) :

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

在每 10 个元素之后以及在第 n 个元素之后都要打印一个换行符, 所有其他元素后都要打印一个空格.

控制流

程序设计中的控制流语句用于控制各计算操作执行的次序. 我们在前面已经使用过一些最常用的流控制结构, 例如 if-else 结构, for 循环, while 循环等等.
下面我们详细的学习控制流语句.

语句与程序块

在 x = 0, i ++ 或着 printf(...) 这样的表达式之后加上一个分号（;），它们就变成了语句. 例如:

```
x = 0;
i ++;
printf (...) ;
```

在 C 语言中，分号是语句结束符.
用一对花括号"{"与"}"把一组声明和语句括在一起就构成了一个复合语句（也叫程序块），复合语句在语法上等价与单条语句. 右花括号用于结束程序块，其后不需要分号.

if-else语句

if-else 语句用于条件判定，其语法如下：

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

其中 else 部分是可选的.

因为 if-else 语句的 else 部分是可选的, 所以在嵌套的 if 语句中省略它的 else 部分将导致歧义. 例如:

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

else 部分与内层的 if 匹配, 我们可以通过程序的缩进结构看出来.

如果这不符合我们的意图，则必须使用花括号强制实现正确的匹配关系：

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

歧义性在下面这种情况最为有害:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0){
            printf(" ..... ");
            return i;
        }
else        /*     错     */
    printf("error -- n is negative\n");
```

程序的缩进结构明确的表明了设计意图，但是编译器无法获得这一信息，它会将 else 部分与内层的 if 配对. 这种错误非常难发现，所以建议在有 if 语句嵌套的情况下使用花括号.

else-if 语句

在 C 语言中我们会经常用到下列结构:

```
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;
```

这种 if 语句序列是编写多路判定最常用的方法. 其中的各表达式将被依次求值, 一旦某个表达式结果为真, 则执行与之相关的语句, 并终止整个语句序列的执行.
最后一个 else 部分用于处理"上述条件均不成立"的情况或默认情况.

下面通过一个折半查找函数说明三路判定程序的用法. 该函数用于判断已排序的数组 v 中是否存在某个特定的值 x . 数组 v 的元素必须以升序排列. 如果 v 中包含 x , 则该函数返回 x 在 v 中的位置 (介于 $0-(n-1)$ 之间的一个整数) ; 否则, 该函数返回 -1 .

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

switch 语句

switch 语句是一种多路判定语句，它测试表达式是否与一些常量数值中的某一个值匹配，并执行相应的分支动作.

```
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
}
```

每一个分支都由一个或多个整数值常量或常量表达式标记. 各分支及 default 分支的排列次序是任意的.

下面我们来看一个前面用 if-else 结构写过的程序，用来统计各个数字，空白符以及其他字符所出现的次数. 我们用 switch 结构改写如下:

```
#include<stdio.h>
/* 统计各个数字，空白符及其他字符出现的次数*/
main()
{
    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;
            defaule:
                nothre++;
                break;
            }
        }
    printf(" digits ␣=");
    for (i = 0; i < 10; ++i)
        printf("%d", ndigit[i]);
    printf(",␣white␣space␣=␣%d,␣other␣=␣%d\n", nwhite,nother);
}
```

while 循环与 for 循环

我们已经使用过 while 与 for 循环语句. 在 while 循环语句

while (表达式)
    语句

中，首先求表达式的值. 如果其值为真非 0，则执行语句，并再次求该表达式的值. 这一循环过程一直进行下去，直到该表达式的值为假（0）为止，随后继续执行语句后面的部分.

for 循环语句：

```
for (表达式1; 表达式2; 表达式3)
        语句
```

它等价与下列的 while 语句：

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

但是当 while 或 for 循环语句中包含 continue 语句时，上述二者就不一定等价了，我们将在后面来讨论 continue 语句.

for 循环语句的3个组成部分都是表达式. 最常见的情况是, 表达式1与表达式3是赋值表达式或函数调用, 表达式2是关系表达式. 这3个组成部分都可以省略, 但分号必须保留. 如果在 for 语句中省略表达式1与表达式3, 它就退化成了 while 循环语句. 如果省略测试条件, 即表达式2, 则认为其值永远是真值, 因此, 下列 for 循环语句:

```
for ( ; ; ) {
        ......
}
```

是一个“无限”循环语句, 这种语句只有借助其他手段 (如 break 语句或 return 语句) 才能终止执行.

至于设计程序时使用 while 循环还是 for 循环, 主要取决与设计人员的个人偏好. 但如果在语句中没有初始化或重新初始化的操作, 使用 while 语句更加自然一些. 例如:

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
        ;
```

如果语句中需要执行简单的初始化和变量递增, 使用 for 语句更合适一些, 它将循环控制集中在循环的开头, 结构更紧凑, 清晰. 例如:

```
for(i = 0; i < n; i++)
    ......
```

下面我们来看一个例子，重新编写将字符串转换为对应数值的函数 atoi. 要求它可以处理可选的前导空白符以及一个可选的加或减号.

```
#include<ctype.h>
/* atoi 函数: 将字符串 s 转换为相应的整型数, 另一版本*/
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (n= 0; isdigit (s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

把循环控制部分集中在一起, 对于多重嵌套循环, 优势更为明显. 下面的函数是对整型数组进行排序的 shell 排序算法.

```
/* shellsort 函数: 按递增顺序对 v[0],...,v[n-1] 进行排序*/
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;
            }
}
```

逗号运算符 “,” 也是 C 语言中优先级最低的运算符, 在 for 语句中经常会遇到它. 被逗号分隔的一对表达式将按照从左到右的顺序进行求值, 各表达式右边的操作数的类型和值即为其结果的类型和值. 这样, 在 for 语句中, 可以将多个表达式放在各个语句成分中, 比如同时处理两个循环控制变量.
下面我们通过函数 reverse(s) 来举例. 该函数用来倒置字符串 s 中各个字符的位置.

```
#include<string.h>
/* reverse 函数: 倒置字符串 s 中各个字符的位置*/
void reverse(char s[])
{
    int temp, i, j;
    for (i = 0, j = strlen(s) - 1; i < j; i++, j--){
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }
}
```

do-while 循环

while 与 for 这两种循环在循环体执行前对终止条件进行测试. 与此相反, C 语言中的第三种循环–do-while 循环则在循环体执行后测试循环条件, 这样循环体至少执行一次.
do-while 循环的语法形式如下:

```
do
        语句
while (表达式);
```

先执行循环体中的语句部分, 然后再求表达式的值. 如果表达式的值为真, 则在执行语句, 依次类推. 当表达式为假时, 循环终止.

下面我们通过函数 itoa 来说明 do-while 循环. 这个函数是 atoi 函数的逆函数, 它把数字转换为字符串.

```
/* itoa 函数: 将数字 n 转换为字符串并保存到 s 中*/
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0)                    /* 记录符号*/
        n = -n;                            /* 使 n 成为正数*/
    i = 0;
    do {                                   /* 以反序生成数字*/
        s[i++] = n % 10 + '0';             /* 取下一个数字*/
    } while ((n /= 10) > 0);               /* 删除该数字*/
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

这里有必要使用 do-while 语句，因为即使 n 的值为 0，也至少要把一个字符放到数组 s 中. 其中的 do-while 语句体中只有一条语句，尽管没有必要，但我们仍然用花括号将该语句括起来了，这样做可以避免草率的将 while 部分误认为是另一个 while 循环的开始.

练习

编写函数 itob(n, s, b)，将整数 n 转换为以 b 为底的数，并将转换结果以字符的形式保存到字符串 s 中. 例如，itob(n, s, 16) 把整数 n 格式化成十六进制整数保存在 s 中.

break 语句与 continue 语句

不通过循环头部或尾部的条件测试而跳出循环, 有时是很方便的. break 语句可用于从 for, while 与 do-while 等循环中提前退出. break 语句能使程序从 switch 语句或最内层循环中立即跳出.

下面的函数 trim 用于删除字符串尾部的空格符, 制表符与换行符. 当发现最右边的字符为非空格符, 非制表符, 非换行符时, 就用 break 从循环中跳出.

```
/* trim 函数: 删除字符串尾部的空格符, 制表符与换行符*/
int trim(char s[])
{
    int n;

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

continue 语句与 break 语句是相关联的. continue 语句用于使 for, while 或 do-while 语句开始下一次循环的执行. 在 while 与 do-while 语句中, continue 语句意味着立即执行测试部分; 在 for 循环语句中, 则意味着使控制转移到递增循环变量部分. continue 语句只能用于循环语句, 不能用于 switch 语句. 例如, 下面这段程序用于处理数组 a 中的非负元素. 如果某个元素的值为负, 则跳过不处理.

```
for (i = 0; i < n; i ++) {
    if (a[i] < 0)
        continue;
    ...
}
```

Some Exercises

练习

1. 以下程序片断的输出结果是什么：

```
int x=1;

while (x <= 5) {
    x++;
    printf ("The value of x is %d. \n", x);
}
printf ("The final value of x is %d. \n", x);
```

练习

2. 下面的程序段是否存在错误？

这段代码是要打印出 0-100 之间的所有偶数：

```
for (int x = 0; x % 2 != 1; x += 2) {
    printf("%d%c", x, " ");

    if (x >= 100)
        break;
}
```

练习

3. 编写一段程序，当用户输入一个 5 位数值时，程序可以将其分离成单独的数字，分离结果之间用 3 个空格分开.

提示：应用整形除和模数运算符.

练习

4. 表示直角三角形的三条边的长度的 3 个整数值，称为勾股弦. 这三条边必须符合一条规则：两条直角边的平方和等于斜边的平方. 编写一段程序，找出小于 500 的所有符合勾股定理的三角形的斜边长度值和直角边的长度值.

强化练习:

- 程序执行了多少次内部 for 循环?
- 在最里面的一层 for 循环中增加一条 break 语句. 该语句在找到第 20 组勾股弦时调用. 那么在找到第 20 组勾股弦之后, 程序会发生什么情况? 是所有三条 for 循环都退出, 还是只退出最里面的一层 for 循环.
- 在程序中增加一条 continue 语句, 使得勾股弦查找过程中直角边的长度不得为 8 . 重新计算程序需要执行多少次内部 for 循环. 并解释 continue 语句如何影响结果.

练习

5. 非负整数 n 的阶乘记为$n!$，定义如下：

$$n! = \begin{cases} n(n-1)(n-2)\cdots 1 & (n >= 1), \\ 1 & (n = 0). \end{cases}$$

a). 编写一段程序，读入一个非负整数，计算并打印其阶乘.
b). 编写一段程序，用以下公式估计自然底数e的值：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

c). 编写一段程序，用以下公式估计自然底数e^x的值：

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

函数与程序结构

函数可以把大的计算任务分解成若干个较小的任务，程序设计人员可以基于函数进一步构造程序，而不需要重新编写一些代码.

C语言在设计中考虑了函数的高效性与易用性这两个原则.

C语言程序一般都由许多小的函数组成，而不是由少量较大的函数组成. 一个程序可以保存在一个或多个源文件中. 各个文件可以单独编译，并且可以与库中已编译过的函数一起加载.

函数的定义形式如下:

```
返回值类型函数名(参数声明表)
{
        声明和语句
}
```

函数的基本知识

首先我们来设计并编写一个程序，它将输入的文本中包含特定的“模式”或字符串的各行打印出来.
这个任务可以划分为下面 3 个部分:

```
while (还有未运行的行)
        if (该行包含指定模式)
            打印该行
```

我们可以考虑将所有的代码都放在 main 主函数中，但是最好的方法是，利用其结构将各个部分设计成一个独立的函数. 分开处理三个小部分比处理一个大的整体更容易.

算法的第一部分我们在前面已经完成了，打印可以使用 printf 函数. 所以我只要编写一个函数判断“该行包含指定的模式”的函数.

我们编写函数 strindex(s, t) 来实现该目标. 这个函数返回字符串 t 在字符串 s 中出现的起始位置或索引. 当 s 中不包含 t 时，返回值为 -1

```
#include<stdio.h>
#define MAXLINE 1000

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

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

int main()
{
    char line[MAXLINE];
    int found = 0;

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

```
/* getline 函数: 将行保存到 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 函数: 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;
}
```

程序可以看成是变量定义和函数定义的集合. 函数之间的通信可以通过参数, 函数返回值以及外部变量进行. 函数在源文件中出现的次序可以是任意的. 只要保证每一个函数不被分离到多个文件中, 源程序可以分成多个文件.

在上面的模式查找程序中, 主程序 main 返回了一个状态, 即匹配的数目.

在不同的操作系统中, 保存多个源文件中的 C 语言程序的编译与加载机制是不同的.
例如, 在 UNIX 系统中, 假定上面的三个函数分别保存在名为 main.c, getline.c 与 strindex.c 的 3 个文件中, 那么可以使用命令:

```
gcc main.c getline.c strindex.c -o mode.o
```

来编译. 上面的命令将逐个编译三个源文件, 最后把目标文件连接成 mode.o 文件.

一般来说, 在编译多个源文件时, 编译器将为每一个源文件生成临时目标文件. 对于模块 main.c 来说, 编译器生成的临时目标文件的文件名是 main.o (绝大多数的 window 操作系统的编译器也是如此, 只不过它们生成的目标文件的后缀名是 obj 而不是 o).

利用这个特性, 我们可以只修改少数几个模块, 加快编译过程. 比如, 在前面的例子中, 如果说 main.c 和 getline.c 中没有错误, 那么在编译完成后, 它们对应的目标文件 main.o 和 getline.o 都将保存下来. 我们可以在下次编译时使用这些目标文件而不是源文件, 而不需要再从源文件开始编译这些模块.

```
gcc main.o getline.o strindex.c -o mode.o
```

有些编译器在编译完成后会自动删除临时目标文件，我们也可以进行增量编译，这需要我们在编译文件的时候给编译器传递命令选项 -c . 这个选项告诉编译器不要执行连接步骤，而保留其产生的临时目标文件，如：

```
gcc −c main.c          从 main.c 生成 main.o .
gcc −c getline.c       从 getline.c 生成 getline.o .
gcc −c strindex.c          从 strindex.c 生成 strindex.o .

gcc main.o getline.o strindex.o −o mode.o    最终生成可执行文件
```

外部变量

C 语言程序可以看成由一系列的外部对象构成，这些外部对象可能是变量或函数. 形容词 external 与 internal 是相对的，internal 用于描述定义在函数内部的函数参数及变量. 外部变量定义在函数之外，因此可以在许多函数中使用.

所谓外部变量，就是在某个源文件中定义，而在另一个源文件中被访问的变量. 如果我们需要在模块中访问某个外部变量，我们需要在该模块中按照普通方式声明该变量，同时在声明语句前面加上 extern 关键字.

因为外部变量可以在全局范围内访问，这就为函数之间的数据交换提供了一种可以代替函数参数与返回值的方式.

假定我们需要定义一个名为 moveNumber 的 int 类型变量，而且我们需要从另外一个文件定义的函数中访问这个这个变量的数值. 那么，如果我们把 moveNumber 变量定义在文件的最开始，所有函数的外面，那么该文件的所有函数都可以使用该变量，也就是说，moveNumber 被定义为一个全局变量. 实际上，其他源文件中的函数也可以访问上面定义的全局变量 moveNumber. 更确切的说，上面定义的 moveNumber 变量，不仅仅是一个全局变量，而且是一个外部全部变量. 为了从另一个模块中访问该变量，我们可以在需要访问该变量的模块中加入外部全局变量的声明语句，如下所示:

```
extern int moveNumber;
```

在使用外部变量的时候，我们必须遵循一个重要的原则，那就是我们必须在某个源文件中确切的定义该变量.（声明只是告诉编译器某个变量的类型，定义则明确的为该变量分配存储空间，类似的例子有函数定义和函数原型声明，我们可以在多个地方出现函数的原型声明，但是必须有一个且只有一个函数的确切定义）.

有两种方法可以定义全局变量，一种是在某个源文件中，在任何函数的外面，不使用 extren 关键字而声明该变量，如:

```
int moveNumber;
```

使用这种方法，还可以同时初始化该变量.

第二种定义外部全局变量的方式是在某个源文件所有函数的外面，使用 extern 关键字声明该变量，同时明确的初始化该变量，如下所示：

```
extern int moveNumber = 0;
```

要注意的是，这两种方法是互相排斥的，我们不能同时使用它们声明同一个变量.

下面的例子演示了外部变量的使用方法. 假定我们把下面的代码输入到源文件 main.c 中:

```
#include<stdio.h>

int i = 5;

int main()
{
    printf("%d", i);
    foo ();

    printf("%d\n", i);
    return 0;
}
```

main.c 中的程序定义了一个全局变量 i, 所有的其他模块都可以使用 extern 声明访问该全局变量. 假定我们在另外一个源文件 foo.c 中有如下语句:

```
extern int i;
void foo (void);
{
    i = 100;
}
```

随后我们使用如下的命令将两个源文件编译在一起:

```
gcc main.c foo.c
```

作用域规则

构成 C 语言程序的函数与外部变量可以分开进行编译. 一个程序可以存放在几个文件中, 原先已经编译过的函数可以从库中进行加载. 那么要注意这样几个问题:

- 如何进行声明才能确保程序在加载时各部分能正确连接?
- 如何安排声明的位置才能确保程序在加载时各部分能正确的连接?
- 如何组织程序中的声明才能确保只有一份副本?
- 如何初始化外部变量

名字的作用域指的是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来说, 其作用域是声明该变量名的函数. 不同的函数中声明的具有相同名字的各个局部变量之间没有任何关系. 函数的参数也是这样的, 实际上可以将它看作局部变量.

外部变量和函数的作用域从声明它的地方开始, 到其所在的文件的末尾结束.

例如, 如果 main, sp, val, push 和 pop 是依次定义在某个文件中的 5 个函数或外部变量, 如:

```
main ()    {.....}

int sp = 0;
double val[MAX];

void push(double f) {.....}

double pop(void) {.....}
```

那么在 push 和 pop 函数中不需要任何声明就可以通过名字访问变量 sp 与 val , 但是, 这两个变量名不能用在 main 函数中, push 和 pop 函数也不能用在 main 函数中.

另一方面，如果要在外部变量的定义之前使用该变量，或者外部变量的定义与变量的使用不再同一个文件中，则必须在相应的变量声明中强制性的使用关键字 extern .
将外部变量的声明与定义严格区分开很重要. 变量的声明主要用来说明变量的属性（主要是变量的类型），而变量的定义除此之外还要引起存储器的分配. 如果将下面语句放在所有函数的外部:

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

那么这两句将定义外部变量 sp 与 val，并为它们分配存储空间，同时还可以作为该源文件中其余部分的声明.

而下面的两句：

extern int sp;
extern double val[];

为源文件的其余部分声明了一个 int 类型的外部变量 sp 以及一个 double 数组类型的外部变量 val（数组的长度在其他地方确定），但是这两个声明并没有建立变量或为它们分配存储单元.

在一个源程序的所有源文件中，一个外部变量只能在某个文件中定义一次，而其他文件可以通过 extern 声明来访问它.

外部变量的定义中必须指定数组的长度，但是 extern 声明则不一定要指定数组的长度.

外部变量的初始化只能出现在其定义中.

静态变量

某些变量，仅供其所在的源文件中的函数使用，其他函数不能访问. 用 static 声明限定外部变量与函数，可以将其后声明的对象的作用域限定为被编译源文件的剩余部分.

通过 static 限定外部对象，可以达到隐藏外部对象的目的，比如，有两个函数 getch 和 ungetch ，需要共享 buf 和 bufp 两个变量，这样 buf 和 bufp 必须是外部变量，但是这两个对象不应该被 getch 和 ungetch 的调用函数所访问. 这样，这两个变量就需要指定为 static .

如果把上面两个函数和两个变量放在一个文件中编译, 如下所示:

```
static char buf[BUFSIZE];
static int bufp = 0;

int getch(void) {.....}

void ungetch(int c) {.....}
```

那么其他函数就不能访问变量 buf 和 bufp , 这样就会避免和同一程序中的其他文件中的相同的名字相冲突.
外部的 static 声明通常多用于变量, 当然, 也可以用于声明函数. 通常情况下, 函数名字是全局可访问的, 对于整个程序的各个部分都是可见的. 但是, 如果把一个函数声明为 static 类型, 则该函数名除了对该函数的声明所在的文件可见外, 其他文件都无法访问.

寄存器变量

register 声明告诉编译器，它所声明的变量放在机器的寄存器中，这样可以使程序更小，执行速度更快. 但编译器可以忽略此选项.
register 声明的形式如下:

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

register 声明只适用与自动变量以及函数的形式参数.
另外，无论寄存器变量实际上是不是存放在寄存器中，它的地址是不能访问的. 在不同的机器中，对寄存器变量的数目和类型的具体限制也是不同的.

初始化

这一节我们对前面讲过的各种存储类的初始化规则做一个总结.

- 在没有进行显式的初始化的情况下，外部变量和静态变量都将被初始化为 0，而自动变量和寄存器变量的初值则没有定义，即初值为无用的信息.
- 定义变量时，可以在变量名后紧跟一个等号和一个表达式来初始化变量:

```
int x = 1;
char squote = '\'';

long day = 1000*L+60*L+60*L+24*L;
```

- 对于外部变量与静态变量来说，初始化表达式必须是常量表达式，而且只能初始化一次（从概念上讲是在程序开始执行前执行初始化）.对于自动变量与寄存器变量，则在每次进入函数或程序块时都要被初始化.
- 对于自动变量与寄存器变量来说，初始化表达式可以不是常量表达式：表达式中可以包含任意在此表达式之前已经定义的值，包括函数调用.

- 数组的初始化可以在声明的后面紧跟一个初始化表达式列表, 初始化表达式列表用花括号括起来, 各初始化表达式之间用逗号分开. 例如:

  $$\textbf{int}\ \text{days}[]\ = \{31,\ 28,\ 31,\ 30,\ 31,30,\ 31,\ 31,\ 30,\ 31,\ 30,\ 31\};$$

- 如果初始化表达式的个数比数组元素数少, 则对外部变量, 静态变量和自动变量来说, 没有初始化表达式的元素将被初始化为 0 .
- 如果初始化表达式的个数比数组元素数多, 则是错误的.
- 不能一次将一个初始化表达式指定给多个数组元素, 也不能跳过前面的数组元素而直接初始化后面的数组元素.

- 字符数组的初始化比较特殊: 可以用一个字符串来代替用花括号括起来并用逗号分隔的初始化表达式序列. 例如:

  char pattern[] = "oule";

  与下面的声明是等价的:

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

递归

C 语言中，函数可以递归调用，即函数可以直接或者间接调用自身. 递归函数可以非常简洁和有效的解决某些计算问题.
这类问题通常的求解步骤能够归结为: 将同样的步骤应用于越来越小的问题子集，最终得到答案.
下面我们来看一个例子: 计算正整数的阶乘. n 的阶乘就是 1 到 n 这 n 个数的乘积，用符合表示 $n!$. 0 的阶乘是 1 .
所以，5的阶乘可以如下计算:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

而

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$$

我们可以看到 6! 是 5! 的 6 倍，也就是说：$6! = 6 * 5!$. 一般来说，任何正整数 n 的阶乘等于 n 乘于 $n-1$ 的阶乘：

$$n! = n * (n-1)!$$

我们可以得到如下程序：

```
#include<stdio.h>

int main ()
{
    unsigned int j;
    unsigned long int factorial (unsigned int n);

    for ( j = 0; j < 11; ++j )
        printf("%2u! = %lu \n", j, factorial (j));
    return 0;
}
```

```
/* 计算正整数阶乘的递归函数*/
unsigned long int factorial ( unsigned int n )
{
    unsigned long int result;

    if ( n == 0 )
        result = 1;
    else
        result = n * factorial ( n - 1);

    return result;
}
```

练习

练习

1. 编写函数 $int\ gcd(int\ u, int\ v)$，该函数接收两个整型的参数 u 和 v，并且返回一个整型的值. 这个函数的功能是找出两个非负整数的最大公约数，并返回结果.

练习

2. 编写一个函数 lcm，用于计算两个整数的最小公倍数，该函数接收两个整型参数，返回它们的最小公倍数. 可以用下面的公式来计算:

$$lcm(u,v) = uv/gcd(u,v) \quad (u,v >= 0)$$

数组

数组（array）是若干同类变量的集合，允许通过统一的名字引用其中的变量. 数组的特定元素通过下标（index）访问. 在 C 语言里，数组都是由连续内存区构成，最低地址对应首元素，最高地址对应末元素. 数组可以是一维的，也可以是多维的.

C 语言中，最常用的数组是字符串（string），就是以空（null）字符结尾的一维数组.

一维数组

定义一维数组的一般形式是:

$$type \quad var\_name[size];$$

与其他变量类似, 数组也必须直接声明, 使编译程序为其分配内存. 这里, *type* 定义数组的基类型（ base type ）, 即数组中各元素的类型. *size* 定义数组中可以放多少个元素. 例如:

double balance[100];

定义了具有 100 个元素的 double 型数组 balance .

通过下标引用数组名可以访问元素, 这只需要在数组名后的方括号内放入元素的下标即可. 例如:

balance[3] = 12.23;

为 balance 中的 3 号元素赋值 12.23 .
在 C 语言中, 所有的数组都把 0 作为其首元素的下标. 因此, 当书写:

char p[10];

时, 就声明字符数组 p[0] ~ p[9] 的 10 个元素.

保存数组所需要的内存量直接与基类型和数组大小有关. 对一维数组而言, 以字节为单位的总内存量可由下式计算得到:

总字节数= sizeof (基类型) * 数组长度

C 语言不检查数组边界, 程序可以在数组两边越界, 写入其他变量, 甚至写入程序代码段. 作为程序员, 必要时应该自己加入边界检查. 例如, 下面的程序可以正确通过编译, 但由于 for 循环使 count 被溢出, 本身不正确.

```
int count[10], i;

for ( i = 0; i < 100; i ++ )
    count[i] = i;
```

数组应用举例

定义数组并通过一个循环结构来进行数组元素的初始化

下面的程序处理的是一个拥有 10 个元素的整形数组 n，例程首先通过 for 循环语句将数组 n 的元素全部初始化成零，然后按照列表格式将它们打印出来. 第一个 printf 打印语句打印的是两列数据的标题，这两列数据随后也是通过 for 循环语句被逐行打印出来.

```c
#include <stdio.h>

int main( void )
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i; /* counter */

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = 0; /* set element at location i to 0 */
   } /* end for */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array n in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */
}
```

定义数组的同时，通过初始化列表来实现数组元素的初始化

可以在定义数组的同时，对数组元素进行初始化，即在定义语句的后面加上一个等号和一对花括号 {}，在花括号内填写用逗号分隔的初始化列表. 例如下面的程序中用 10 个初始值来对拥有 10 个元素的整型数组 n 进行初始化.

```
#include <stdio.h>

int main( void )
{
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   int i; /* counter */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */

}
```

如果初始化列表中提供的初始值个数少于数组拥有的元素个数，则余下的数组元素将被初始化为 0 . 例如，下面语句将 n 的 10 个元素全部初始化为 0 .

```
int n[10] = {0};
```

需要注意的是：自动数组不能自动地初始化为零. 至少要将第一个数组元素初始化为零，这样余下的元素才会被自动的初始化为零.
在通过使用初始化列表来实现数组元素初始化的数组定义语句中，如果忽略了数组元素个数的填写，则系统将会把初始化列表中提供的初始值的个数作为数组所拥有的元素总数. 例如：

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

就将创建一个拥有 5 个元素的整型数组 n .

用符号常量来定义数组的大小并通过计算来进行数组元素的初始化

下面的程序将一个拥有 10 个元素的整型数组 s 的元素分别初始化为 $2, 4, 6, \cdots, 20$ 然后按照列表格式打印出来.

其中，我们使用 #define 来初始化数组元素的个数. 采用符号常量来定义数组的大小有助于提高程序的可扩展性. 如果将程序中表示数组大小的符号常量 SIZE 的替换文本 10 改成 1000，那么在不改变程序中执行语句的情况下，程序的功能就由原先的“对拥有 10 个元素的整型数组进行初始化”提升为“对拥有 1000 个元素的整型数组进行初始化”.

```
#include <stdio.h>
#define SIZE 10

int main( void )
{
    int s[ SIZE ];
    int j;

    for ( j = 0; j < SIZE; j++ ) {
        s[ j ] = 2 + 2 * j;
    }

    printf( "%s%13s\n", "Element", "Value" );

    for ( j = 0; j < SIZE; j++ ) {
        printf( "%7d%13d\n", j, s[ j ] );
    }

    return 0;
}
```

计算数组元素值的总和

```
#include <stdio.h>
#define SIZE 12

int main( void )
{
    int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
    int i;
    int total = 0;

    for ( i = 0; i < SIZE; i++ ) {
        total += a[ i ];
    }

    printf( "Total of array element values is %d\n", total );

    return 0;
}
```

用数组来统计民意调查的结果

下面举的例子要采用数组来统计在一次民意调查中收到的数据：

40 名学生被召集以打分的形式评价学生食堂中饭菜的质量. 分数是 1 到 10 , 1 分表示非常糟糕, 10 分表示非常满意. 现在用一个整型数组来存储这些学生打出的分数, 统计这次调查的结果.

在这个问题中, 我们引入两个数组, 一个整型数组 responses 拥有 40 个元素, 用来保存学生打出的分数, 一个整型数组 frequency 拥有 11 个元素, 用来统计学生可能打出的分数, 其中我们忽略它的第一个元素 frequency[0] , 将分数 1 放在 frequency[1] 中.

```
#include <stdio.h>
#define RESPONSE_SIZE 40
#define FREQUENCY_SIZE 11

int main( void )
{
   int answer; /* counter to loop through 40 responses */
   int rating; /* counter to loop through frequencies 1-10 */

   int frequency[ FREQUENCY_SIZE ] = { 0 };

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

   for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
      ++frequency[ responses [ answer ] ];
   }
```

```
    printf( "%s%17s\n", "Rating", "Frequency" );

    for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
        printf( "%6d%17d\n", rating, frequency[ rating ] );
    }
    return 0;

}
```

向函数传递一维数组

在 C 语言中，不能把整个数组作为传入函数的变元，只能用不带下标的数组名向函数传入数组指针. 例如:

```
int main()
{
    int i[10];

    func1(i);

    /* ... */
}
```

将数组 i 的首地址传入函数 func1() .

在接受数组的函数中，定义数组形式参数的方法有三种：指针，定尺寸数组，无尺寸数组. 例如，func1() 可以定义为：

```
void func1(int* x)              /* pointer */
{
    /* ... */
}

void func1(int x[10])             /* sized array */
{
    /* ... */
}

void func1(int x[])               /* unsized array  */
{
    /* ... */
}
```

数组名就是数组第一个元素的地址

```
#include <stdio.h>

int main( void )
{
    char array[ 5 ]; /* define an array of size 5 */

    printf( "    array = %p\n&array[0] = %p\n    &array = %p\n",
        array, &array[ 0 ], &array );

    return 0;

}
```

分别将整个数组及其元素传递给函数

```
#include <stdio.h>
#define SIZE 5

void modifyArray( int b[], int size );
void modifyElement( int e );

int main( void )
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* initialize a */
   int i; /* counter */

   printf( "Effects of passing entire array by reference:\n\nThe "
           "values of the original array are:\n" );

   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[ i ] );
   }

   printf( "\n" );
```

```
    modifyArray( a, SIZE );

    printf( "The values of the modified array are:\n" );

    for ( i = 0; i < SIZE; i++ ) {
        printf( "%3d", a[ i ] );
    }

    printf( "\n\n\nEffects of passing array element "
            "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );

    modifyElement( a[ 3 ] );

    printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );

    return 0;
}
```

```
void modifyArray( int b[], int  size  )
{
   int  j;  /* counter */

   for  (  j  = 0; j  < size;  j++ ) {
      b[  j  ] *= 2;
   }
}

void modifyElement( int e )
{
   printf(  "Value in modifyElement is %d\n", e *= 2 );
}
```

二维数组

C 语言支持多维数组. 最简单的多维数组是二维的, 即一维数组构成的数组. 定义大小为 10, 20 的整型二维数组 d 时, 可以写为:

```
int d [10][20];
```

引用数组 d 的 1, 3 点时, 书写成:

```
d [1][3];
```

下面的程序将数值 1 到 12 装入一个二维数组, 然后打印之.

```
#include<stdio.h>
int main()
{
    int t, i, num[3][4];
    for( t = 0; t < 3; t ++)
        for( i = 0; i < 4; i ++)
            num[t][i] = (t * 4) + i + 1;

    for( t = 0; t < 3; t ++){
        for( i = 0; i < 4; i ++)
            printf("%3d", num[t][i]);
        printf("\n");
    }
    return 0;
}
```

把二维数组作为函数的变元时，实际只传送指向指向数组第一元素的指针. 然而，在接收二维数组变元的函数中，至少应该定义第二维的长度，因为 C 编译程序必须了解每列的长度后才能正确的对数组进行下标操作. 例如，接收 10, 10 二维数组的函数必须定义成:

```
void func1(int x[ ][10])
{
    /* ... */
}
```

为了处理函数中形如:

x [2][4]

的语句，编译程序必须了解第二维的准确长度. 如果不知道第二维的长度，编译程序无法确定第二行从哪里开始.

指针

指针是一种保存变量地址的变量. 对于成功的 C 程序设计而言, 正确理解并使用指针是至关重要的. 因为: 第一, 指针为函数提供修改调用变量的手段; 第二, 指针支持 C 动态分配子程序; 第三, 指针可以改善某些子程序的效率; 最后, 指针为动态数据结构 (例如二叉树和链表) 提供支持.

指针是 C 语言最强的特性之一, 同时又是最危险的特性之一. 例如, 含无效的指针可以使系统瘫痪. 更甚之, 不正确的使用指针容易引入难以排除的程序错误.

定义指针变量

假定我们定义了如下的一个叫做 count 的变量:

int count = 10;

我们可以使用下面的方法定义另外一个变量 int_pointer，可以使用它以间接的方式来存取 count 的值:

int *int_pointer;

以上语句中的 * 代表 int_pointer 是一个整型指针变量. 这样程序就可以用 int_pointer 来间接的存取一个或多个整型数值.

一元运算符 & 称为地址运算符, 用于取一个对象的地址. 因此, 仿照我们给出的 count 和 int_pointer , 我们可以写出下面的语句:

```
int_pointer = &count;
```

建立 int_pointer 和 count 之间的间接引用关系. 上面的表达式将建立一个指向变量 count 的指针, 而不是变量 count 的值赋予变量 int_pointer .

为了获取指针变量 int_pointer 指向的变量 count 所包含的内容, 我们可以用指针运算符, 即星号 * . 因此, 如果我们定义 x 为整型变量, 则下面的语句:

```
x = *int_pointer;
```

将 int_pointer 通过间接关系所指向的值赋予变量 x . 因为 int_pointer 先前被设定为指向 count , 所以这个语句将把包含在变量 count 的值 10 , 赋予变量 x .

下面看一个关于指针变量的例子:

```
#include<stdio.h>

int main()
{
    char c = 'Q';
    char *char_pointer = &c;

    printf ("%c %c\n", c, *char_pointer);

    c = '/';
    printf ("%c %c\n", c, *char_pointer);

    *char_pointer = '(';
    printf ("%c %c\n", c, *char_pointer);

    return 0;
}
```

再看一个再表达式中使用指针变量的例子:

```
#include<stdio.h>
int main()
{
    int i1, i2;
    int *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf ("i1 = %d, i2 = %d, *p1 = %d, *p2 = %d\n", i1, i2, *
        p1, *p2);

    return 0;
}
```

指针与函数参数

因为 C 语言中是以传值的方式将参数传递给被调用函数，因此，被调用函数不能直接修改主调函数中变量的值. 例如如果有函数 swap 交换两个元素的值. 若函数定义为下面形式:

```
void swap (int x, int y)   /* 错误定义的函数*/
{
    int temp;

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

则下面的调用语句无法达到交换的目的:

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

如何实现目标呢? 可以使主调函数将指向所要交换的变量的指针传递给被调函数, 即:

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

swap 函数的所有参数都声明为指针, 并通过指针来间接访问它们指向的操作数.

```
void swap (int *px, int *py)  /* 交换 *px 和 *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

注意: 指针参数使得被调用函数能够访问和修改主调函数中对象的值.

指针与数组

指针和数组的关系十分密切，通过数组下标完成的操作都可以通过指针来实现. 一般来说，用指针编写的程序比用数组下标编写的程序执行速度快，但是，用指针实现的程序理解起来会困难一些. 如下声明:

int a[10], *ip;

定义了一个长度为 10 的数组 a 和一个指向整型对象的指针 ip . 那么赋值语句

ip = &a[0];

则可以将指针 ip 指向数组 a 的第 0 个元素，即 ip 的值为数组元素 a[0] 的地址.

如果指针 ip 指向数组中的某个特定元素, 那么 ip+1 将指向下一个元素, ip+i 将指向 ip 所指向数组元素之后的第 i 个元素. 因此, 如果指针 ip 指向 a[0] , 那么 *(ip+1) 引用的是数组元素 a[1] 的内容, ip+i 是数组元素 a[i] 的地址, *(ip+i) 引用的是数组元素 a[i] 的内容.

无论数组 a 中的元素的类型或数组的长度是多少, 上面的结论都成立. “指针加一”就意味着 ip+1 指向 ip 所指向的对象的下一个对象.

下标和指针运算之间具有密切的对应关系. 根据定义, 数组类型的变量或表达式的值是该数组第 0 个元素的地址. 执行赋值语句

pa = &a[0];

后, ip 和 a 具有相同的值. 因为数组名所代表的就是该数组最开始的一个元素的地址, 所以, 上面的语句也可以写成

ip = a;

同样的, 对数组元素 a[i] 的引用也可以写出 *(a+i) 这种形式.

当把数组名传递给一个函数时，实际上传递的是该数组的第一个元素的地址. 在被调用函数中，该参数是一个局部变量，因此，数组名参数必须是一个指针，也就是一个存储地址值的变量. 例如:

```
/* strlen 函数: 返回字符串 s 的长度*/
int strlen (char *s)
{
    int n;

    for ( n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

下面的程序中 arraySum 函数用来计算一个整数数组所包含的所有元素的和.

```
#include<stdio.h>
int arraySum (int array[], const int n)
{
    int sum = 0, *ptr;
    int *const arrayEnd = array + n;
    for( ptr = array; ptr < arrayEnd; ++ptr)
        sum += *ptr;
    return sum;
}

int main()
{
    int arraySum (int array[], const int n);
    int value[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5};
    printf ("The sum is %d\n", arraySum(values, 10));
    return 0;
}
```

下面的程序中，使用 4 种方法来引用数组元素: 数组下标表示，将数组名当做指针的指针(偏移量表示)，指针下标表示，以及指针的指针(下标表示).

```
#include <stdio.h>

int main( void )
{
   int b[] = { 10, 20, 30, 40 };
   int *bPtr = b;
   int i;
   int offset;

   printf( "Array b printed with:\nArray subscript notation\n" );

   for ( i = 0; i < 4; i++ ) {
      printf( "b[ %d ] = %d\n", i, b[ i ] );
   }

   printf( "\nPointer/offset notation where\n"
           "the pointer is the array name\n" );
```

```
    for ( offset = 0; offset < 4; offset++ ) {
        printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
    }

    printf( "\nPointer subscript notation\n" );

    for ( i = 0; i < 4; i++ ) {
        printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
    }

    printf( "\nPointer/offset notation\n" );

    for ( offset = 0; offset < 4; offset++ ) {
        printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
    }
    return 0;
}
```

指向字符串的指针

指向数组的指针最常见的应用是将指针指向字符串. 下面我们看一个 copyString 的函数, 用来将一个字符串复制到另一个中. 如果使用普通下标的表示方法, 则可以编写如下:

```
void copyString (char to[], char from[])
{
    int i;

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

如果用指针来编写 copyString，我们就不用下标变量 i 了. 下面的程序是指针版本的 copyString 函数:

```
#include<stdio.h>
 void copyString (char *to, char *from)
 {
     from( ; *from != '\0'; ++from, ++to )
          *to = *from;
     *to = '\0';
 }
 int main()
 {
      void copyString (char *to, char *from);
      char string1 [] = "A string to be copied.";
      char string2 [50];
      copyString (string2, string1);
      printf ("%s\n", string2);
      copyString (string2, "So is this.");
      printf ("%s\n", string2);
      return 0;
}
```

下面在看一个字符串比较的函数 strcmp(s,t) . 该函数比较字符串 s 和 t , 并且按照 s 以字典序的小于, 等于或大于 t 的结果分别返回负整数, 0或正整数. 该返回值是 s 和 t 由前向后逐字比较时遇到的第一个不相等字符处的字符的差值.

```
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];
}
```

下面是用指针的方式实现的 strcmp 函数:

```
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

练习

用指针的方式实现函数 strcat . 函数 strcat(s, t) 将 t 指向的字符串复制到 s 指向的字符串的尾部.

指针数组

由于指针本身也是变量，所以它们也可以象其他变量一样存储在数组中．我们来看一段程序说明它的用法．
考虑这样一个问题：编写一个函数 month_name(n)，它返回一个指向第 n 个月名字的字符串的指针．

```
char *month_name(int n)
{
    static char *name[] = {
        " Illegal _month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

这是内部 static 类型数组的一种的应用. month_name(n) 函数中包含一个私有的字符串数组, 当它被调用时, 返回一个指向正确元素的指针.

其中, name 的声明是一个一维数组, 数组的元素为字符指针. name 数组的初始化通过一个字符串列表实现, 列表中的每个字符串赋值给数组相应位置的元素. 第 i 个字符串的所有字符存储在存储器中的某个位置, 指向它的指针存储在 name[i] 中.

指针与多维数组

在 C 语言中，很难区别二维数组与指针数组. 比如上面的例子中的 name . 假如有如下两个定义:

```
int a[10][20];
int *b[10];
```

那么，从语法上说，a[3][4] 和 b[3][4] 都是对一个 int 对象的合法引用. 但是 a 是一个真正的二维数组，它分配了 200 个 int 类型长度的存储空间，并通过常规的矩阵下标来访问数组元素. 而 b 的定义仅仅分配了 10 个指针，并没有对它们初始化，它们的初始化必须以显式的方式进行，比如静态初始化或者代码初始化.

指针数组的一个优点是数组的每一行的长度可以不同，也就是说，b 的每个元素不必都指向一个具有 20 个元素的向量.

命令行参数

在支持 C 语言的环境中，可以在程序开始执行时将命令行参数传递给程序. 调用主函数 main 时，它带有两个参数. 第一个参数（习惯上称为 argc，用于参数计数）的值表示运行程序时命令行中参数的数目；第二个参数（称为 argv，用于参数向量）是一个指向字符串数组的指针，其中每一个字符串对应一个参数. 通常使用多级指针.
最简单的例子是 UNIX 命令 echo，它将命令行参数回显在屏幕上，如命令

echo hello, world

将打印下列输出：

hello, world

按照 C 语言的约定, argv[0] 的值是启动该程序的程序名, 因此 argc 的值至少为 1 . 如果 argc 的值为 1 , 则说明程序名后面没有命令行参数. 在上面的例子中, argc 的值为 3 , argv[0], argv[1] 和 argv[2] 的值分别是 “echo” , “hello,” 以及 “world” . 另外, ANSI 标准要求 argv[argc] 的值必须是一个空指针.

我们用 C 语言实现 UNIX 中的命令 echo , 这里将 argv 看成一个字符指针数组:

```
#include<stdio.h>

int 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;
}
```

因为 argv 是一个指向指针数组的指针，所以，可以通过指针而非数组下标的方式处理命令行参数. 这个程序的第二个版本是在对 argv 进行自增运算，对 argc 进行自减运算的基础上实现的，其中 argv 是一个指向 char 类型的指针的指针:

```
#include<stdio.h>

int main(int argc, char *argv[])
{
    while ( --argc > 0 )
        printf("%s%s", *++argv, ( argc >1) ? " " : "");
    printf("\n");
    return 0;
}
```

函数作为参数

在 C 语言中，指向函数的指针具有多种用途，例如，可以将它作为参数，可以用于数组中，也可以作为函数的返回值.
假设我们想用一些来实现一种计算过程. 考虑下面的公式:

$$\sum_{k=m}^{n} f^2(k)$$

在一个例子中，$f(k) = \sin(k)$. 在另一个例子中，$f(k) = 1/k$.
下面这个函数用于完成这个任务:

sum_sqr.c:

```
#include"sum_sqr.h"

double sum_square(double f(double x), int m, int n)
{
    int k;
    double sum = 0.0;

    for (k = m; k <= n; ++k)
        sum += f(k) * f(k);
    return sum;
}
```

在函数定义的头部，第一个参数声明告诉编译器 f 是个函数. 标识符 x 只具有提示作用，编译器将会忽略它. 而当一个函数出现在参数声明中时，编译器就会把它理解为指针. 下面是与此等价的函数的头部:

double sum_square(**double** (∗f)(**double**), **int** m, **int** n)

我们把第一个参数理解为“f是个指向函数的指针，它所指向的函数接收一个 double 类型的参数，并返回一个 double 类型的值”.

括号是必要的，它把 * 和 f 绑定在一起.

在 sum_square() 函数定义的函数体中，我们可以把指针 f 直接当成是个函数，也可以对它进行解引用. 例如，可应用

```
sum += (*f)(k) * (*f)(k);
```

来代替

```
sum += f(k) * f(k);
```

下面我们编写一个完整的程序，在 main() 函数中，我们将使用来自数学函数库的 sin() 函数，以及我们自己所编写的 f() 函数.

sum_sqr.h:

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

double f(double x);
double sum_square(double f(double x), int m, int n);
```

main.c:

```
#include"sum_sqr.h"

int main()
{
    printf("%s%.7f\n%s%.7f\n",
        "First computation", sum_square(f, 1, 1000),
        "Second computation", sum_square(sin, 2, 13));
    return 0;
}
```

ffunc.c:

```
#include"sum_sqr.h"

double f(double x)
{
    return 1.0 / x;
}
```

例子: 使用二分法寻找函数的根

问题: 如何寻找一个特定的实数值函数的根

如果实数 x 满足等式 $f(x) = 0$, 那么 x 就是 f 的根. 如果不存在求根公式, 我们必须使用数学方法来寻根.

假设 f 是个连续的实数值函数在$[a, b]$区间上, 如果 f(a) 和 f(b) 符号相反, 那么 f 在 $[a, b]$ 区间内存在一个根.

二分法步骤如下: 假设区间$[a, b]$的中点是m, 如果$f(m) = 0$, 则m 就是函数的根. 如果不是, 则观察 $f(a)$ (或者 $f(b)$) 和 $f(m)$ 是否反号, 然后继续这个过程, 当区间足够小的时候, 我们就可以用它的中点的值来近似 f 的近似值.

下面的程序用来寻找下面这个多项式的根:

$$x^5 - 7x - 3$$

find_root.h:

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

typedef double dbl;
extern int cnt;
extern const dbl eps;

dbl bisection(dbl f(dbl x), dbl a, dbl b);
dbl f(dbl x);
```

main.c:

```
#include"find_root.h"

int cnt = 0;
const dbl eps = 1e-13;

int main()
{
    dbl a = -10.0;
    dbl b = 10.0;
    dbl root;

    assert(f(a) *f(b) <= 0);
    root = bisection(f, a, b);
    printf("%s%d\n%s%.15f\n%s%.15f\n",
        "No. of fction calls :", cnt,
        "Approximate root:", root,
        "Function value:", f(root));
    return 0;
}
```

bisection.c:

```
#include"find_root.h"
dbl bisection(dbl f(dbl x), dbl a, dbl b)
{
    dbl m = (a + b)/2.0;
    ++cnt;
    if(f(m) == 0.0 || b - a < eps)
        return m;
    else if (f(a) * f(m) < 0.0)
        return bisection(f, a, m);
    else
        return bisection(f, m, b);
}
```

fct.c:

```
#include"find_root.h"
dbl f(dbl x)
{
    return (x*x*x*x*x*x - 7.0 * x - 3.0);
}
```

练习: 开普勒方程

问题:在 17 世纪早期，Johannes Kepler 希望能够对下面这个方程进行求解:

$$m = x - e * \sin(x)$$

编写一个程序，当参数值取值为 $m = 2.2$ 和 $e = 0.5$ 时对 Kepler 方程式进行求解.
注意: 应用到前面例子中的函数 bisection()，以及指向函数的指针的方法.

动态内存分配

使用 calloc() 函数和 malloc() 函数进行动态内存分配

C 在标准函数库中提供了 calloc() 函数和 malloc() 函数，它们的原型位于 stdlib.h 头文件中. calloc 这个名字表示“连续分配” (contiguous allocation) , malloc 这个名字表示“内存分配” (memory allocation).

我们可以使用 calloc() 函数和 malloc() 函数动态的创建数组，字符串，结构和联合的内存空间. 下面是一个例子，显示了如何使用 calloc() 动态的为数组分配空间:

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

int main()
{
    int *a;   /* to be used as an array */
    int n;    /* the size of the array */
     ......   /* get n from somewhere */

    a = calloc(n, sizeof(int)); /* get space for a */
     ......   /* use a as an array */
}
```

calloc() 函数为 n 个元素的数组分配连续的内存空间, 这些内存空间的所有位置都被初始化为 0 . malloc() 函数和它类似, 区别是 malloc() 函数不会对分配的内存初始化.
使用 calloc() 函数和 malloc() 函数动态分配的内存空间在函数退出时并不会返回给系统. 程序员必须显式的使用 free() 函数来释放这些内存空间. 例如下面的调用形式:

```
free(ptr);
```

使得 ptr 所指向的内存空间被释放. 如果 ptr 为 NULL , 这个函数不会产生任何效果. 如果 ptr 并不是 NULL , 它必须指向以前由 calloc() 函数或 malloc() 函数所分配的内存空间的首地址.

矩阵的动态分配

工程师和科学家广泛使用矩阵. 下面我们来解释如何动态的创建矩阵, 把它创建成指针数组的形式, 以便传递给函数, 使后者对不同大小的矩阵都适用.

为什么二维数组无法满足要求

例如, 如果我们需要一个$3 * 3$的矩阵, 我们可以使用下面的声明:

```
double a[3][3];
```

它为一个二维数组分配空间. 如果我们希望寻找 a 的行列式. 在矩阵 a 被填充之后, 我们希望能使用类似下面的方法:

```
det = determinant(a);
```

determinant() 函数的定义应该如下面的方式:

```
double determinant(double a[][3])
{
    ...
```

由于编译器需要 3 这个值才能创建正确的存储映射函数，因此我们的行列式函数只能用于 $3*3$ 的矩阵. 如果我们想计算 $4*4$ 的矩阵的行列式，我们需要编写一个新的函数的定义，这是无法接受的. 我们希望编写一个适用于任何大小的正方矩阵的行列式函数.

用指针数组创建矩阵

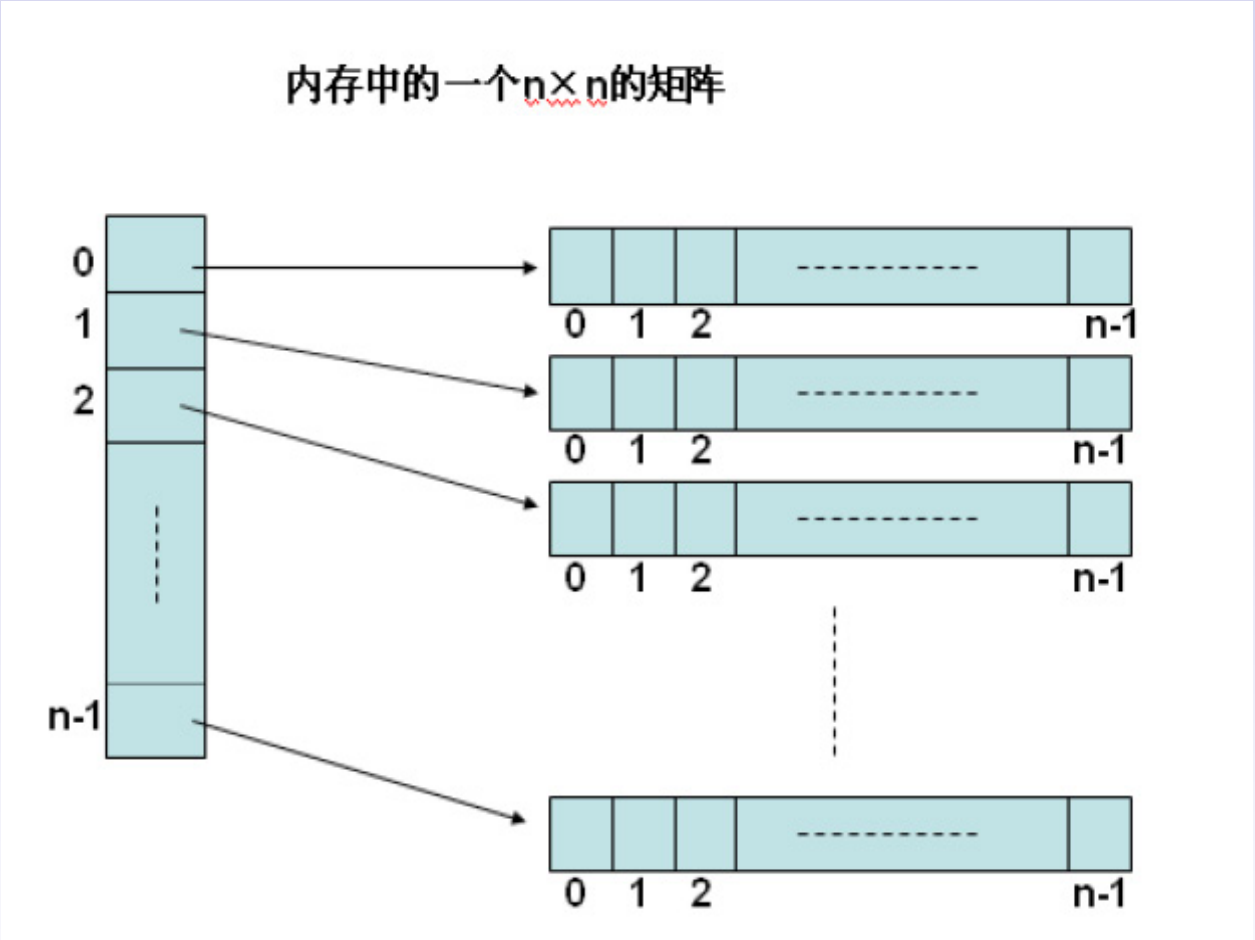
通过使用 double 类型的指针的指针，我们可以创建我们所需要的任意大小的矩阵，并且可以把它传递一个函数，使后者可以对任意大小的矩阵进行操作. 首先是创建矩阵空间的代码:

```
int i, j, n;
double **a, det, tr;

...                          /* get n from somewhere */

a = calloc(n, sizeof(double *));
for(i = 0; i < n; ++i)
    a[i] = calloc(n, sizeof(double));
```

矩阵的大小并不需要预先知道. 我们可以从用户那里读取 n , 也可以从文件获取, 或者通过计算产生. 它不需要以常量的形式出现在程序中. 一旦我们知道了矩阵的大小, 我们就可以使用标准库函数 calloc() 动态的创建矩阵的空间. 所以, 下面这条语句:

a = calloc(n, **sizeof(double** ∗));

它为 a 分配空间, 可以把 a 看成是个长度为 n 的 double 类型的指针数组. 在 for 循环中, a 的每个元素被赋值为内存中一个长度为 n 的 double 类型的数组的首地址.



注意：尽管矩阵 a 在内存中是以指针数组的形式存储的，但是我们仍然可以使用普通的矩阵表达式 a[i][j] 来访问第 i 行，第 j 列的元素. 由于 a 的类型是 $double^{**}$，因此 a[i] 的类型是 $double^{*}$，它可以看作是矩阵的第 i 行. 由于 a[i] 的类型是 $double^{*}$，因此 a[i][j] 的类型是 $double$，它是矩阵的第 i 行第 j 列的元素.

既然我们已经为矩阵的元素进行了赋值，我们就可以把这个矩阵作为参数传递给各个函数，当然只要传递数组名就可以了.

调整下标范围

在数学中, 向量和矩阵的下标通常是从 1 而不是从 0 开始. 我们对上面的代码进行调整, 以满足这个习惯. 下面这个函数用于创建一个长度为 n 的向量(一维数组):

```
double *get_vector_space(int n)
{
    int i;
    double *v;

    v = calloc(n, sizeof(double));
    return (v-1);
}
```

由于返回的指针值向左偏移，因此调用环境中的下标将从 1 到 n，而不是从 0 到 n-1．现在，我们就可以使用下面的代码:

```
int n;
double *v;

 ......
v = get_vector_space(n);
for(i = 1; i <= n; ++i)
    v[i] = rand()%19 - 9
 ......
```

除了对指针进行偏移之外，我们也可以选择分配更多的存储空间，然后将部分空间弃之不用.

```
v = calloc(n+1, sizeof(double));
for(i = 1; i <= n; ++i)
    v[i] = i;
 ......
```

当然，对指针进行偏移的技巧更好一点，因为它并没有浪费空间. 下面我们用这个方法来编写一个函数，为一个下标从 1 而不是 0 开始的 $m \times n$ 矩阵分配空间.

```
double **get_matrix_space{int m, int n}
{
    int i;
    double **a;

    a = calloc(m, sizeof(double));
    --a;
    for(i = 1; i <= m; ++i){
        a[i] = calloc(n, sizeof(double));
        --a[i];
    }
    return a;
}
```

如果希望以前通过 calloc() 分配的空间可以再次由系统所使用，需要调用 free() 函数来释放已经分配的空间，那么对于上面的矩阵，我们需要仔细的还原 get_martrix_space() 函数所产生的指针偏移.

```
void release_matrix_space(double **a, int m)
{
    int i;
    for(i = 1; i <= m ; ++i)
        free(a[i]+1);
    free(a+1);
}
```

关键字 const 和指针

我们已经知道如何用 const 修饰变量或数组的声明，以告诉编译器，程序将不会修改这个变量或数组的值. 对于指针来说，(使用 const 修饰符时) 要考虑两件事情: 一是指针是否会被修改，二是指针所指向的值是否会被修改. 假定有如下声明:

```
char c = 'X';
char *charPtr = &c;
```

指针变量 charPtr 被设为指向变量 c，如果它总是指向 c，则可被声明为一个指针常量如下:

```
char * const charPtr = &c;
```

意思为“charPtr”是一个指向字符的指针常量. 所以下面的语句:

```
charPtr = &d;   /* 不是有效的*/
```

将会产生错误信息.

相反的，如果 charPtr 指向的位置的值不会通过使用指针变量 charPtr 被改变，则可以使用如下声明：

const char *charPtr = &c;

意思为“charPtr”指向一个字符常量. 当然了，这并不意味着这个值不能够通过变量 c 所修改. 这个声明只是意味着，它将不会因类似下面的语句而被修改：

```
*charPtr = 'Y';    /* 不是有效的*/
```

在指针变量和它所指向的单元都不改变的情况下，可以应用下面的声明：

const char * **const** *charPtr = &c;

第一个 const 表明指针指向的单元的内容不会被改变，第二个 const 表明指针本身不会被改变.

扑克牌洗牌和发牌模拟

下面我们使用随机数生成器来设计一个模拟扑克牌洗牌发牌的程序.

使用 4 数组 deck 来表示 52 张扑克牌. 数组行对应于花色名: 第 0 行代表红桃, 第 1 行代表方块, 第 2 行代表草花, 第 3 行代表黑桃. 数组的列对应与牌面. 我们还要使用字符串数组 suit 来表示 4 种花色名, 以及字符串数组 face 来表示 13 种牌面值.

模拟扑克牌洗牌的算法如下: 首先, 将数组 deck 清零, 然后, 行 $(0 \sim 3)$ 和列 $(0 \sim 12)$ 用产生的的随机数分别来选定, 将牌的顺序号 1 写入数组元素 deck[row][column] 中, 以表示这张扑克牌将是洗牌后的第一张牌. 重复以上处理过程, 依次将牌的顺序号 $2, 3, \cdots, 52$ 随机地写入数组 deck 中, 以表示洗牌后的第 $2, 3, \cdots, 52$ 张都是哪张牌.

由于只是将牌的顺序号写入随机选择的数组 deck 中, 所以有可能会出现一张牌被选择两次的情形, 即当 deck[row][column] 被选择时, 它的值已为非 0 值, 这时, 选择就应被忽略, 然后随机地产生其他的行和列的值, 直到发现一个没有被选择过的牌为止. 最后, 1 到 52 之间的数占据了数组 deck 中的每个元素位置. 扑克牌就全部洗完了.

由于洗过的牌会被随机的重复选择, 因此, 这个洗牌算法的执行是不确定的. 这种现象被称为不确定性延迟 (Indefinite postponement) .

问题: 自己设计算法, 考虑如何可以避免不确定性延迟.

为了发第一张牌，我们要在数组中搜索 deck[row][column] 值为 1 的数组元素. 这是一个嵌套的 for 循环语句来实现的，其中行由 0 变化到 3，列由 0 变化到 12. 那么这个数组元素对应的扑克牌到底是什么呢？由于 4 种花色已经被事先放在数组 suit 中了，所以，为了得到这张牌的花色，打印出字符串 suit[row] 就行了. 同理，为了得到这张牌的面值，打印出字符串 face[column] 就行了.

下面我们按照“自顶向下，逐步求精”的方法来完成上述过程. 这个“顶”可以简单的表示为：

洗发 52 张扑克牌

第一次求精得到如下结果:

初始化数组 suit

初始化数组 face

初始化数组 deck

洗牌

分发 52 **张牌**

对“洗牌”进行扩展, 得到如下结果:

for 52 **张牌中的每一张牌**

将牌的顺序号随机的放入被选择的尚未被占据的数组 deck **中**

对“分发 52 张牌”进行扩展, 得到如下结果:

for 52 **张牌中的每一张牌**

在数组 deck **中查找牌的顺序号, 并打印这张牌的牌面和花色**

对“将牌的顺序号随机的放入被选择的尚未被占据的数组 deck 中”进行扩展，得到如下结果：

随机的选择一个 deck **中的位置**

while **选择的位置在此之前已经被选择过**

    重新随机地选择一个 deck **中的位置**

将扑克牌的顺序号写入被选择的 deck **中的相应位置**

对“在数组 deck 中查找牌的顺序号，并打印这张牌的牌面和花色”进行扩展，得到如下结果：

for **数组** deck **中的每个位置**

    if **该位置位置存有的扑克牌顺序号为待查找的顺序号**

        打印扑克牌的牌面和花色

综合以上分析，得到第三次完整的求精结果如下：

初始化数组 suit
初始化数组 face
初始化数组 deck
for 52 张牌中的每一张牌
    随机的选择一个 deck 中的位置
    while 选择的位置在此之前已经被选择过
        重新随机地选择一个 deck 中的位置
    将扑克牌的顺序号写入被选择的 deck 中的相应位置
for 52 张牌中的每一张牌
    for 数组 deck 中的每个位置
        if 该位置位置存有的扑克牌顺序号为待查找的顺序号
            打印扑克牌的牌面和花色

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

/* prototypes */
void shuffle( int wDeck[][ 13 ] );
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] );

int main( void )
{
   /* initialize suit array */
   const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades
      " };

   /* initialize face array */
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };
```

```
    /* initialize deck array */
    int deck[ 4 ][ 13 ] = { 0 };

    srand( time( 0 ) ); /* seed random-number generator */

    shuffle ( deck );
    deal( deck, face, suit );

    return 0; /* indicates successful termination */

}
```

```
void shuffle( int wDeck[][ 13 ] )
{
   int row;     /* row number */
   int column; /* column number */
   int card;    /* counter */

   for ( card = 1; card <= 52; card++ ) {
     do {
         row = rand() % 4;
         column = rand() % 13;
      } while( wDeck[ row ][ column ] != 0 );

      wDeck[ row ][ column ] = card;
   }
}
```

```
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] )
{
   int card;    /* card counter */
   int row;     /* row counter */
   int column; /* column counter */

   for ( card = 1; card <= 52; card++ ) {
      for ( row = 0; row <= 3; row++ ) {
         for ( column = 0; column <= 12; column++ ) {
            if ( wDeck[ row ][ column ] == card ) {
               printf( "%5s of %-8s%c", wFace[ column ], wSuit[
         row ],
                     card % 2 == 0 ? '\n' : '\t' );
            }
         }
      }
   }
}
```

练习:

1. 修改上面的程序, 使其洗发牌函数能够处理一手五张牌的扑克游戏, 然后编写下列函数:

a). 判断这一手牌中是否包含一个对子.

b). 判断这一手牌中是否包含两个对子.

c). 判断这一手牌中是否包含三张同级的牌(例如, 3 张 J).

d). 判断这一手牌中是否包含四张同级的牌.

e). 判断这一手牌中是否包含一个同花.

f). 判断这一手牌中是否包含一个顺子.

g). 判断这一手牌中是否包含一个葫芦(三个同级的牌带两个同级的牌).

2. 利用上面的程序，在发完多手牌之后，最终计算所得的各种牌型的概率是多少. 例如，同花顺的数学概率为 0.00198 你得到的值是否接近这个值?并且利用你的程序来判断上面题目中各种牌型的大小.

3. 利用上面的函数，编写一个能够处理两手五张牌扑克游戏程序，该程序分别评价每一手牌，然后判断哪一手更好.

4. 修改上面的程序，使其能够模拟庄家. 庄家一手五张牌是扣着的，所以，玩家不能够看到庄家的牌. 该程序先评价庄家手里的牌，然后根据牌的好坏，庄家可能抽出一张，两张或者更多张牌，来替换掉原先手中相应的废牌. 程序将重新评价手里的牌.

5. 修改上面的程序，使其能够自动模拟庄家发牌. 但是玩家也允许自行决定对自己手中的牌进行替换. 程序将评价两手牌，决出胜负. 然后你与计算机玩20次，看看谁胜的多，并根据比赛结果优化你的程序.

结构

结构是一个或多个变量的集合，这些变量可能为不同的类型，为了处理的方便而将这些变量组织在一个名字之下．由于结构将一组相关的变量看作一个单元而不是各自独立的实体，因此结构有助于组织复杂的数据．

假如我们要在程序中存储一个二维平面上点 p 的坐标，则我们用一个整型变量 x 保存 p 的横坐标，用一个整型变量 y 保存 p 的纵坐标．

使用这种方法表示点，需要在程序中同时关注两个不同的变量，虽然这两个变量所保存的数值在逻辑上是关联的．如果我们能构把这两个变量编为一组，那么就好的多．结构正是用于这个目的．

结构的基本知识

我们可以定义一个结构 point，该结构包含两个成员，分别用于保存横坐标和纵坐标. 定义结构的语句如下:

```
struct point {
    int x;
    int y;
};
```

上述定义的结构包含两个成员变量: x 和 y . 从某种程度上说, 上面的语句实际上定义了一个新的数据类型.

有了上面的 point 结构以后, 我们声明类型为 struct point 的变量, 如下面的语句:

struct point vertex;

或者将两个声明放在同一行上, 如下:

struct point vertex, midpoint;

这样就定义了两个 struct point 类型的变量 vertex 和 midpoint . 结构的初始化可以在定义的后面用初值表进行. 初值表中同每个成员对应的初值必须是常量表达式, 例如:

struct point vertex = { 100, 100 };

如果我们要使用结构变量的某个成员，可以使用圆点操作符“.”，例如，把结构变量 vertex 的成员变量 x 设置为 25，可以使用下面的语句：

```
vertex.x = 25;
```

要注意的是：在结构变量名和点号之间以及点号与成员变量名之间是不允许有空格的.

下面这段程序用到了前面说的关于结构的一些知识点，程序中定义了了一个记录时间的结构包含三个整型成员变量分别存储年，月，日三个变量.

```
#include<stdio.h>
int main()
{
    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 5;
    today.day = 20;
    today.year = 2006;

    printf ("Today's date is %d/%d/%.2d.\n", today.month, today.
        day, today.year % 100);

    return 0;
}
```

结构与函数

结构和合法操作只有几种:

- 作为一个整体复制和赋值,
- 通过 & 运算符取地址, 访问其成员.

其中, 复制和赋值包括向函数传递参数以及从函数返回值. 结构之间不可以进行比较. 可以用一个常量成员值列表初始化结构, 自动结构也可以通过赋值进行初始化.

下面我们编写几个对点和矩形进行操作的函数. 至少可以通过 3 种可能的方法传递结构: 一是分别传递各个结构成员, 二是传递整个结构, 三是传递指向结构的指针.

首先来定义矩形的结构. 我们可以考虑用对角线上的两个点来定义矩形, 相应的结构定义如下:

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

结构 rect 包含两个 point 类型的成员. 如果我们声明 screen 变量:

```
struct rect screen;
```

则可以用语句

```
screen.pt.x
```

引用 screen 的成员 pt1 的 x 坐标.

再看一个函数 makepoint，它带有两个整型参数，并返回一个 point 类型的结构：

```
/* makepoint 函数: 通过 x, y 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;
}
```

现在可以使用 makepoint 函数动态地初始化任意结构，也可以向函数提供结构类型地参数. 例如:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

下面需要编写一些函数对点执行算术运算. 例如:

```
/* addpoint 函数: 将两个点相加*/
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

其中, 函数的参数和返回值都是结构类型. 因为结构类型的参数和其他类型的参数一样, 都是通过值传递的, 所以我们可以直接将相加的结果赋值给 p1 .

下面来看另外一个例子. 函数 ptinrect 判断一个点是否在给定的矩形内部. 我们做这样一个约定: 矩形包括其左侧边和底边, 但不包括顶边和右侧边.

```
/* ptinrect 函数: 如果点 p 在矩形 r 内, 则返回 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;
}
```

这里假设矩形中 pt1 的坐标小于 pt2 的坐标.

如果传递给函数的结构很大，使用指针方式的效率通常比复制整个结构的效率要高．结构指针类似于普通变量指针．声明

```
struct point *pp;
```

将 pp 定义为一个指向 struct point 类型对象的指针．如果 pp 指向一个 point 结构，那么 *pp 即为该结构，而 (*pp).x 和 (*pp).y 则是结构成员．可以按照下面的方法使用 pp：

```
struct point origin, *pp;

pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

结构指针的使用频率非常高，为了使用方便，C 语言提供了另一种简写方法. 假定 p 是一个指向结构的指针，可以用

p −>结构成员

这种形式引用相应的结构成员. 这样，就可以用下面的形式改写前面的一行代码:

```
printf ("origin is (%d, %d)\n", pp->x, pp->y);
```

在所有运算符中，下面 4 个运算符的优先级最高：结构运算符“.”和“->”，用于函数调用的“()”以及用于下标的“[]”，因此，它们同操作数之间的结合也最紧密. 例如，对于结构声明

```
struct {
    int len;
    char *str;
} *p;
```

表达式

```
++p -> len
```

将增加 len 的值，而不是增加 p 的值，因为，其中隐含的括号关系是

```
++(p->len)
```

结构数组

结构可以让我们将逻辑上相关的一组值组织起来，作为一个单独的变量处理. 例如，使用 time 结构，就不需要跟踪三个独立的变量 hour, minutes, seconds ，而只需要考虑一个变量就可以了.
那么如果我们要处理 10 个时间变量，可以将数组和结构结合起来. C 语言并不限制只能在数组中保存简单的数据类型. 我们同样可以在数组中保存结构，比如:

```
struct time experiments[10];
```

定义了一个结构数组，该数组包含 10 个元素，而每个元素都是一个 struct time 类型的变量.

我们可以使用习惯的方法引用结构数组中的元素，比如，要将 experiments 数组中的第二个结构变量的值赋为 12:30:00，可以使用下面的语句：

```
experiments[2].hour = 12;
experiments[2].minutes = 30;
experiments[2].seconds = 00;
```

如果要将 experiments 数组中的第 4 号元素传递给函数 checkTime，我们可以使用如下的语句：

```
checkTime(experiments[4]);
```

当然，函数 checkTime 必须接受 struct time 类型的参数，其定义可能如下所示：

```
void checkTime (struct time t0)
{
    ...
}
```

对结构数组进行初始化和多维数组初始化是类似的，例如下面的语句：

```
struct time runTime [5] =
    { {12, 0, 0}, {12, 30, 0}, {13, 15, 0} };
```

声明了一个拥有 5 个元素的结构数组 runTime，并且前三个元素初始化为 12:00:00, 12:30:00 和 13:15:00．下面的语句仅仅初始化了结构数组的第 2 号元素：

```
struct time runTime[5] =
    { [2] = {12, 0, 0} };
```

而下面的语句，则仅仅初始化了结构数组第 1 号元素的成员变量 hour 和 minute．

```
static struct time runTime[5] = { [1].hour =12, [1].minutes = 30 };
```

下面这段程序反映了结构数组的用法:

```
#include<stdio.h>
struct time
{
    int hour;
    int minutes;
    int seconds;
};
int main()
{
    struct time timeUpdate (struct time now);
    struct time testTimes[5] =
            { {11, 59, 59}, {12, 0, 0}, {1, 29, 59},
            {23, 59, 59}, {19, 12, 27} };
    int i;
```

```
    for ( i = 0; i < 5; ++i ) {
        printf ("Time is %.2i:%.2i:%.2i", testTimes[i].hour,
                testTime[i].minutes, testTimes[i].seconds);

        testTimes[i] = timeUpdate (testTimes[i]);

        printf (" ... one second later it's %.2i:%.2i:%.2i\n",
                testTimes[i].hour, testTime[i].minutes,
                testTimes[i].seconds);
    }
    return 0;
}
```

```
/* 以秒为单位更新时间的函数*/
struct time timeUpdate (struct time now)
{
    ++now.seconds;
    if ( now.seconds == 60 ) {  /* 下一分钟*/
        now.seconds = 0;
        ++now.minutes;

        if ( now.minutes == 60 ) {      /* 下一小时*/
            now.minutes = 0;
            ++now.hour;

            if ( now.hour == 24 )   /* 午夜*/
                now.hour = 0;
        }
    }
    return now;
}
```

类型定义 (typedef)

C 语言提供了一个称为 typedef 的功能, 它用来建立新的数据类型名, 例如, 声明

typedef int Length;

将 Length 定义为与 int 具有同等意义的名字. 类型 Length 可用于类型声明, 类型转换等, 它和类型 int 完全相同, 例如:

```
Length len, maxlen;
Length *lengths[];
```

这里举一个更复杂的例子:

```
typedef struct tnode *Treeptr;

typedef struct tnode{
    char *word;
    int count;
    Treeptr left ;
    Treeptr right ;
} Treenode;
```

上述类型定义创建了两个新类型关键字: Treenode (一个结构) 和 Treeptr (一个指向该结构的指针) .

需要注意的是: typedef 声明并没有创建一个新类型, 它只是为某个已存在的类型增加了一个新的名称而已. typedef 声明也没有增加任何新的语义: 通过这种方式声明的变量与通过普通声明方式声明的变量具有完全相同的属性.

除了表达方式更简洁之外, 使用 typedef 还有两个重要的原因. 首先, 它可以使程序参数化, 提高程序的可移植性. 第二个作用是为程序提供更好的说明性, Treeptr 类型显然比一个声明为指向复杂结构的指针更容易让人理解.

练习

编写一个名为 *elapsed_time* 的函数，该函数接受两个类型为 **struct** *time* 的参数，返回一个 **struct** *time* 类型的参数，用于表示两个参数之间流逝的时间（使用时，分，秒的形式）. 比如下面的语句

elasped_time (time1, time2)

如果 *time1* 代表 *3:45:15*，而 *time2* 代表 *9:44:03*，那么返回的值应该是 *5* 小时 *58* 分 *48* 秒.

输入和输出

下面我们来解释如何使用标准函数库中的一些输入和输出函数，包括 printf() 和 scanf() 相关的函数，用于处理文字和字符串.

在那些数据位于磁盘等外部存储的应用程序中，基本的文件输入和输出功能是非常重要的. 我们将说明如何打开需要处理的文件已经如何使用文件指针.

输出函数 printf()

printf() 函数有两个灵活的特性: 首先, 它可以打印任意长度的参数列表. 其次, 打印是由简单的转换格式控制的. printf() 的参数列表由两部分组成:

- 控制字符串;
- 其他参数.

在下面这个例子中:

```
printf("she sells %d %s for %f", 99, "sea shells", 3.77);
```

控制字符串为: "she sells %d %s for %f"
其他参数为: 99, "sea shells", 3.77

输出函数 scanf()

scanf() 函数有两个非常出色的能够灵活运用的特性. 第一个特性是可以扫描任意长度的参数列表, 第二个特性是输入由简单的转换格式所控制. scanf() 函数的参数列表由两部分组成:

- 控制字符串;
- 其他参数.

在下面这个例子中:

```
char a, b, c, s[100];
int n;
double x;

scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

控制字符串为: "%c%c%c%d%s%lf"
其他参数为: &a, &b, &c, &n, s, &x

对于文件的简单操作

标识符 FILE 是在 stdio.h 中定义的，它是一种特殊的结构，它的成员描述了一个文件的当前状态. 要使用文件，程序员并不需要知道与这个结构相关的任何细节.

stdio.h 文件还定义了 3 个文件指针，分别是 stdin，stdout和stderr. 尽管它们都是指针，但我们有时仍然把它们称为文件.

在 C 中的写法	名称	注释
stdin	标准输入文件	连接到键盘
stdout	标准输出文件	连接到屏幕
stderr	标准错误文件	连接到屏幕

文件处理函数的原型出现在 stdio.h 头文件中. 下面是 fprintf() 和 canf() 函数的原型:

```
int  fprintf (FILE *fp, const char *format, ...);
int  fscanf (FILE *fp, const char *format, ...);
```

下面这种形式的语句

```
fprintf ( file_ptr , control_string , other_arguments);
```

把输出写入到由 file_ptr 所指向的文件. 控制字符串 (control_string) 和其他参数 (other_arguments) 的用法与 printf() 中一致. 例如:

```
fprintf (stdout, ...) ; 等价于 printf (...) ;
```

fopen() 和 fclose() 函数

文件可以抽象的理解为字符流. 在文件被打开之后, 我们就可以使用标准函数库中的文件处理函数来访问这个字符流. 下面我们来看一下如何来使用 fopen() 和 fclose() .

文件具有几个重要的属性: 它们具有名称; 它们必须被打开和关闭; 它们可以进行读取、写入和添加. 从概念上说, 我们在打开文件之前, 不能对它执行任何操作. 当文件被打开时, 我们可以从头部或者尾部访问它. 为了防止意外的误用, 我们必须告诉操作系统我们所执行的是 3 种操作 (读取、写入或添加) 中的哪一种. 当我们完成了对文件的操作之后, 就需要关闭文件. 例如下面的部分代码:

```
#include<stdio.h>

int main(void)
{
    int a, sum = 0;
    FILE *ifp, *ofp;

    ifp = fopen("my_file", "r");      /* open for reading */
    ofp = fopen("outfile", "w");      /* open for writing */
    ...
```

这段代码在当前目录打开两个文件: 用于读取的 my_file 和用于写入的 outfile . 标识符 ifp 是“输入文件”的助记符, ofp 是“输出文件”的助记符.

假设 my_file 文件中包含了一些整数，如果我们想对它求和，并把结果写入到 outfile 中，我们可以使用下面的代码：

```
while(fscanf(ifp, "%d", &a) == 1)
    sum += a;
fprintf(ofp, "The sum is %d.\n", sum);
```

在完成了对文件的使用后，我们可以关闭文件：

```
fclose(ifp);
```

这个操作将关闭由 ifp 所指向的文件.

fopen() 函数调用以一种特定的模式打开指定的文件，并返回指向这个文件的文件指针.

模式	含　义
"r"	打开文本文件，用于读取
"w"	打开文本文件，用于写入
"a"	打开文本文件，用于添加
"rb"	打开二进制文件，用于读取
"wb"	打开二进制文件，用于写入
"ab"	打开二进制文件，用于添加

如果以读取模式打开一个无法读取或不存在的文件, fopen() 操作就会失败. 这时, fopen()返回一个 NULL 指针. 如果以写入模式打开一个不存在的文件, fopen() 就会创建这个文件. 如果这个文件已经存在, 原来的文件就会被覆盖. 如果以添加模式打开一个不存在的文件, fopen() 就会创建这个文件. 如果这个文件已经存在, 写入就会从这个文件的尾部开始进行.

下面我们编写一个对文件进行空间加倍的程序来说明一些文件处理函数的用法. 在 main() 函数中, 我们打开以命令行参数形式传递的文件, 用于读取和写入. 在打开文件后, 我们调用 double_space() 函数来完成空间加倍的操作.

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

void double_space(FILE *, FILE *);
void prn_info(char *);

int main(argc, char **argv)
{
    FILE *ifp, *ofp;

    if (argc != 3){
        prn_info(argv[0]);
        exit(1);
    }
    ifp = fopen(argv[1], "r");
    ofp = fopen(argv[2], "w");
    double_space(ifp, ofp);
    fclose(ifp);
    fclose(ofp);
    return 0;
}
```

```
void double_space(FILE *ifp, FILE *ofp)
{
    int c;

    while((c = getc(ifp)) != EOF){
        putc(c, ofp);
        if(c == '\n')
            putc('\n', ofp);
    }
}

void prn_info(char *pgm_name)
{
    printf("\n%s%s%s\n\n%s%s\n\n",
        "Usage:   ", pgm_name, "  infile   outfile",
        "The contents of infile will be double-space",
        "and written to outfile.");
}
```

练习:

1. 写一个程序, 将一个整数文件中的数据乘以 10 以后写到另一个文件中.
2. 编写一个程序, 在屏幕上显示一个文件的内容, 一次显示 20 行. 输入文件应该以命令行参数的形式提供. 这个程序应该在输入一个回车符后显示接下来的 20 行内容.