# 第2章　类型、运算符与表达式

变量与常量是程序中所要处理的两种基本数据对象。说明语句中列出了所要使用的变量的名字及该变量的类型，可能还要给出该变量的初值。运算符用于指定要对变量与常量进行的操作。表达式则用于把变量与常量组合起来产生新的值。一个对象的类型决定着该对象可取值的集合以及可以对该对象施行的运算。本章将要对这些构件进行详细讨论。

ANSI C语言标准对语言的基本类型与表达式做了许多小的修改与增补。所有整数类型现在都有signed（有符号）与unsigned（无符号）两种形式，且可以表示无符号常量与十六进制字符常量。浮点运算可以以单精度进行，另外还可以使用更高精度的 long double类型。字符串常量可以在编译时连接。枚举现在也成了语言的一部分，这是经过长期努力才形成的语言特征。对象可以说明成const（常量），这种对象的值不能进行修改。语言还对算术类型之间的自动强制转换规则做了扩充，使这一规则可以适合更多的数据类型。

## 2.1　变量名

对变量与符号常量的名字存在着一些限制，这一点在第 1章中没有指出来。名字由字母与数字组成，但其第一个字符必须为字母。下划线_也被看做是字母，它有时可用于命名比较长的变量名以提高可读性。由于库函数通常使用以下划线开头的名字，因此不要将这类名字用做变量名。大写字母与小写字母是有区别的， x与X是两个不同的名字，一般把由大写字母组成的名字用做符号常量。

在内部名字中至少前 31个字符是有效的。对于函数名与外部变量名，其中所包含的字符的数目可以小于31个，这是因为它们可能会被语言无法控制的汇编程序和装配程序使用。对于外部名，ANSI C标准保证了唯一性仅对前 6个字符而言并且不区分大小写。诸如 if、else、int、float等关键词是保留的，不能把它们用做变量名。所有关键词中的字符都必须小写。

在选择变量名时比较明智的方法是使所选名字的含义能表达变量的用途。我们倾向于局部变量使用比较短的名字（尤其是循环控制变量，亦叫循环位标），外部变量使用比较长的名字。

## 2.2　数据类型与大小

在C语言中只有如下几个基本数据类型：

| | |
|---|---|
| char | 单字节，可以存放字符集中一个字符。 |
| int | 整数，一般反映了宿主机上整数的自然大小。 |
| float | 单精度浮点数。 |
| double | 双精度浮点数。 |

此外，还有一些可用于限定这些基本类型的限定符。其中 short与long这两个限定符用于限定整数类型：

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

在这种说明中，int可以省去，一般情况下许多人也是这么做的。

引入这两个类型限定符的目的是为了使 short与long提供各种满足实际要求的不同长度的整数。int通常反映特定机器的自然大小，一般为 16位或32位，short对象一般为16位，long对象一般为32位。各个编译程序可以根据其硬件自由选择适当的大小，唯一的限制是， short与int对象至少要有16位，而long对象至少要有 32位；short对象不得长于 int对象，而int对象则不得长于long对象。

类型限定符signed与unsigned可用于限定char类型或任何整数类型。经unsigned限定符限定的数总是正的或0，并服从算术模$2^n$定律，其中n是该类型机器表示的位数。例如，如果 char对象占用8位，那么unsigned char变量的取值范围为0~255，而signed char变量的取值范围则为- 128~127（在采用补码的机器上）。普通char对象是有符号的还是无符号的则取决于具体机器，但可打印字符总是正的。

long double类型用于指定高精度的浮点数。如同整数一样，浮点对象的大小也是由实现定义的，float、double与long double类型的对象可以具有同样大小，也可以表示两种或三种不同的大小。

在标准头文件<limits.h>与<float.h>中包含了有关所有这些类型的符号常量以及机器与编译程序的其他性质。这些内容将在附录 B中讨论。

练习2-1　编写一个程序来确定 signed及unsigned的char、short、int与long变量的取值范围，可以通过打印标准头文件中的相应值来完成，也可以直接计算来做。后一种方法较困难一些，因为要确定各种浮点类型的取值范围。

## 2.3　常量

诸如1234一类的整数常量是int常量。long常量要以字母 l或L结尾，如123456789L。一个整数常量如果大到在int类型中放不下，那么也被当做 long常量处理。无符号常量以字母 u或U结尾，后缀ul或UL用于表示unsigned long常量。

浮点常量中必须包含一个小数点（如 123.4）或指数（如1e-2）或两者都包含，在没有后缀时类型为double。后缀f与F用于指定float常量，而后缀l或L则用于指定long double常量。

整数值除了用十进制表示外，还可以用八进制或十六进制表示。如果一个整数常量的第一个数字为0，那么这个数就是八进制数；如果第一个数字为 0x或0X，那么这个数就是十六进制数。例如，十进制数31可以写成八进制数037，也可以写成十六进制数 0x1f或0X1F。在八进制与十六进制常量中也可以带有后缀　l或L（long，表示长八进制或十六进制常量）以及后缀　u或U（unsigned，表示无符号八进制或十六进制常量），例如，0XFUL是一个unsigned long 常量（无符号长整数常量），其值等于十进制数15。

字符常量是一个整数，写成用单引号括住单个字符的的形式，如　'x'。字符常量的值是该字符在机器字符集中的数值。例如，在 ASCII字符集中，字符 '0' 的值为48，与数值0没有关系。如果用字符 '0' 来代替像48一类的依赖于字符集的数值，那么程序会因独立于特定的值而更易于阅

读。虽然字符常量一般用来与其他字符进行比较，但字符常量也可以像整数一样参与数值运算。

有些字符用字符常量表示，这种字符常量是诸如“ \n ”（换行符）的换码序列，换码序列看起来像两个字符，但只用来表示一个字符。此外，我们可以用

```
'\ooo'
```

来指定字节大小的位模式，*ooo*是1~3个八进制数字（0…7）。位模式还可以用

```
'\xhh'
```

来指定，*hh*是一个或多个十六进制数字（0…9，a…f，A…F）。因此，可以如下写：

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

也可以用十六进制写

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

下面是所有的换码序列：

| | | | |
|---|---|---|---|
| \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 | *xhh* | 十六进制数 |
| \v | 纵向制表符 | | |

字符常量 '\0' 表示其值为0的字符，即空字符。我们用 '\0' 来代替0，以在某些表达式中强调字符的性质，但其数字值就是0。

常量表达式是其中只涉及到常量的表达式。这种表达式可以在编译时计算而不必推迟到运行时，因而可以用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字面值，是用双引号括住的由 0个或多个字符组成的字符序列。例如：

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

或：

```
""   /* 空字符串 */
```

双引号不是字符串的一部分，它只用于限定字符串。在字符常量中使用的换码序列同样也可以用在字符串中，在字符串中用 \" 表示双引号字符。编译时可以将多个字符串常量连接起来：

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

等价于

```
"hello, world"
```

这种表示方法可用于将比较长的字符串分成若干源文件行。

从技术角度看，字符串常量就是字符数组。在内部表示字符串时要用一个空字符 '\0' 来结尾，故用于存储字符串的物理存储单元数比括在双引号中的字符数多一个。这种表示方法意味着， C 语言对字符串的长度没有限制，但程序必须扫描完整个字符串才能决定这个字符串的长度。标准库函数strlen(s) 用于返回其字符串变元 s的长度（不包括末尾的 '\0'。下面是我们设计的一个版本：

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

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

strlen等字符串函数均说明在标准头文件 <string.h>中。

请仔细区分字符常量与只包含一个字符的字符串的区别： 'x'与 "x"不相同。前者是一个整数，用于产生字母 x在机器字符集中的数值（内部表示值）。后者是一个只包含一个字符（即字母 x）与一个 '\0'的字符数组。

另外还有一种常量，叫做枚举常量。枚举是常量整数值的列表，如同下面一样：

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

在enum说明中第一个枚举名的值为 0，第二个为 1，如此等等，除非指定了显式值。如果不是所有值都指定了，那么未指定名字的值依着最后一个指定值向后递增，如同下面两个说明中的第二个说明：

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

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                /* FEB的值为2，MAR的值为3，等等。*/
```

不同的枚举中的名字必须各不相同，同一枚举中各个名字的值不要求不同。

枚举是使常量值与名字相关联的又一种方便的方法，其相对于 #define语句的优势是常量值可以由自己控制。虽然可以说明 enum类型的变量，但编译程序不必检查在这个变量中存储的值是否为该枚举的有效值。枚举变量仍然提供了做这种检查的机会，故其比 #define更具优势。此外，调试程序能以符号形式打印出枚举变量的值。

## 2.4 说明

除了某些可以通过上下文做的隐式说明外，所有变量都必须先说明后使用。说明中不仅要

指定类型，还要包含由一个或多个该类型的变量组成的变量表。例如：

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

同一类型的变量可以以任何方式分散在多个说明中，上面两个说明也可以等价地写成如下五个说明：

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

后一种形式需要占用较多的空间，但这样不仅便于向各个说明中增添注解，也便于以后的修改。

变量在说明时可以同时初始化。如果所说明的变量名后跟一个等号与一个表达式，那么这个表达式被作为初始化符。例如：

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

如果所涉及的变量不是自动变量，那么只初始化一次，而且从概念上讲应该在程序开始执行之前进行，此时要求初始化符必须为常量表达式。显式初始化的自动变量每当进入其所在的函数或分程序时就进行一次初始化，其初始化符可以是任何表达式。外部变量与静态变量的缺省初值为0。未经显式初始化的自动变量的值为未定义值（即为垃圾）。

在变量说明中可以用const限定符限定，该限定符用于指定该变量的值不能改变。对于数组，const限定符使数组所有元素的值都不能改变：

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

const说明也可用于数组变元，表明函数不能改变数组的值：

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

如果试图修改const限定的值，那么所产生的后果取决于具体实现。

## 2.5 算术运算符

二元算术运算符包括+、-、*、/以及取模运算符%。整数除法要截取掉结果的小数部分。表达式

```
x % y
```

的结果是x除以y的余数，当y能整除x时，x % y的结果为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，此外，能被400整除的年份也是闰年。因此，有

```
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%不能作用于float或double对象。在有负的运算分量时，整数除法截取的方向以及取模运算结果的符号于具体机器，在出现上溢或下溢时所要采取的动作也取决于具体机。

二元＋和－运算符的优先级相同，但它们的优先级比 *、/和%的优先级低，而后者又比一元＋和－运算符低。算术运算符采用从左至右的结合规则。

本章末尾的表2-1总结了所有运算符的优先级和结合律。

## 2.6 关系运算符与逻辑运算符

关系运算符有如下几个：

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

所有关系运算符具有相同的优先级。优先级正好比它们低一级的是相等运算符：

```
==   !=
```

关系运算符的优先级比算术运算符低。因而表达式

```
i < lim -1
```

等于

```
i < (lim - 1)
```

更有趣的是逻辑运算符 &&与||。由&&与||连接的表达式从左至右计算，并且一旦知道结果的真假值就立即停止计算。绝大多数 C程序利用了这些性质。例如，下面这个循环语句取自第１章的输入函数getline：

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

在读一个新字符之前必须先检查一下在数组 s中是否还有空间存放这个字符，因此首先必须测试是否i < lim -1。而且，如果这一测试失败（即i < lim -1不成立），那么就没有必要继续读下一字符。

类似地，如果在调用 getchar函数之前就对 c是否为EOF进行测试，那么也是令人遗憾的，因此，函数调用与赋值都必须在对 c中的字符进行测试之前完成。

&&运算符的优先级比 ||运算符的优先级高，但两者都比关系运算符和相等运算符的优先级低。从而，像

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

之类的表达式就不需要另外加圆括号了。但是，由于 !=运算符的优先级高于赋值运算符 =的优先级，在子表达式

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

中，圆括号还是需要的，这样才能达到我们所希望的先把函数值赋给 c再与 '\n' 进行比较的效果。

按照定义，如果关系表达式与逻辑表达式的计算结果为真，那么它们的值为 1；如果为假，那么它们的值为0。

一元求反运算符 !用于将非0运算分量转换成 0，把0运算分量转换成 1。该运算符通常用在诸如

```
if ( !valid )
```

一类的构造中，一般不用

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

来代替。要想笼统地说哪个更好比较难。诸如 !valid一类的构造读起来好听一点（“如果不是有效的”），但这种形式在比较复杂的情况下可能难于理解。

练习2-2 不使用&&或||运算符编写一个与上面的for循环语句等价的循环语句。

## 2.7 类型转换

当一个运算符的几个运算分量的类型不相同时，要根据一些规则把它们转换成某个共同的类型。一般而言，只能把“比较窄的”运算分量自动转换成“比较宽的”运算分量，这样才能不丢失信息，例如，在诸如

```
f + i
```

一类的表达式的计算中要把整数变量i的值自动转换成浮点类型。不允许使用没有意义的表达式，例如，不允许把float表达式用作下标。可能丢失信息的表达式可能会招来警告信息，如把较长整数类型的值赋给较短整数类型的变量，把浮点类型赋给整数类型，等等，但不是非法表达式。

由于char类型就是小整数类型，在算术表达式中可以自由地使用 char类型的变量或常量。这就使得在某些字符转换中有了很大的灵活性。一个例子是用于将数字字符串转换成对应的数值的函数atoi：

```
/* atoi：将字符串s转换成整数 */
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;
}
```

正如第1章所述，表达式

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

用于求s[i]中存储的字符所对应的数字值，因为 '0'、'1'、'2' 等的值形成不一个连续的递增序列。

将字符转换成整数的另一个例子是函数 lower，它把ASCII字符集中的字符映射成对应的小写字母。如果所要转换的字符不是大写字母，那么 lower函数返回原来的值。

```
/* lower：把字符c转换成小写字母；仅对ASCII字符集 */
int lower (int c)
{
        if (c >= 'A' && c<= 'Z' )
            return  c + 'a' - 'A';
    else
```

```
        return c;
}
```

这个函数是为ASCII字符集设计的。在ASCII字符集中，大写字母与对应的小写字母像数字值一样有固定的距离，并且每一个字母都是连续的——在A至Z之间只有字母。然而，后一个结论对于EBCDIC字符集不成立，故这一函数在EBCDIC字符集不只是转换了字母。

附录B中介绍的标准头文件 <ctype.h>定义了一组用于进行独立于字符集的测试和转换的函数。例如，tolower(c)函数用于在c为大写字母时将之转换成小写字母，故 tolower是上述lower函数的替代函数。同样条件，

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

可以用

```
isdigit(c)
```

代替。

我们从现在起要使用<ctype.h>中定义的函数。

在将字符转换成整数时有一点比较微妙。 C语言没有指定char类型变量是无符号量还是有符号量。当把一个 char类型的值转换成 int类型的值时，其结果是不是为负整数？结果视机器的不同而有所变化，反映了不同机器结构之间的区别。在某些机器上，如果字符的最左一位为 1，那么就被转换成负整数（称做“符号扩展”）。在另一些机器上，采取的是提升的方法，通过在最左边加上0把字符提升为整数，这样转换的结果总是正的。

C语言的定义保证了机器的标准打印字符集中的字符不会是负的，故在表达式中这些字符总是正的。但是，字符变量存储的位模式在某些机器上可能是负的，而在另一些机器上却是正的。为了保证程序的可移植性，如果要在 char变量中存储非字符数据，那么最好指定 signed或unsigned限定符。

关系表达式（如i > j）和由&&与||连接的逻辑表达式的值在其结果为真时为 1，在其结果为假时为0。因此，赋值语句

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

在c的值为数字时将d置为1，否则将d置为0。然而，诸如isdigit一类的函数在变元为真时返回的可能是任意非0值。在if、while、for等语句的测试部分，“真”的意思是“非0”，从这个意义上看，它们没有什么区别。

我们很希望能进行隐式算术类型转换。一般而言，如果诸如 + 或 * 等二元运算符的两个运算分量具有不同的类型，那么在进行运算之前先要把“低”的类型提升为“高”的类型。附录A.6节严格地给出了转换规则。然而如果没有无符号类型的运算分量，那么只要使用如下一组非正式的规则就够了：

如果某个运算分量的类型为 long double，那么将另一个运算分量也转换成 long double类型；
否则，如果某个运算分量的类型为 double，那么将另一个运算分量也转换成 double类型；
否则，如果某个运算分量的类型为 float，那么将另一个运算分量也转换成 float类型；
否则，将char与short类型的运算分量转换成 int类型。
然后，如果某个运算分量的类型为 long，那么将另一个运算分量也转换成 long类型。

注意，在表达式中float类型的运算分量不自动转换成double类型，这与原来的定义不同。一般而言，数学函数（如定义在标准头文件 <math.h>中的函数）要使用双精度。使用 float类型的主要原因是为了在使用较大的数组时节省存储单元，有时也为了节省机器执行时间（双精度算术运算特别费时）。

当表达式中包含unsigned类型的运算分量时，转换规则要复杂一些。主要问题是，在有符号值与无符号值之间的比较运算取决于机器，因为它们取决于各个整数类型的大小。例如，假定int对象占16位，long对象占32位，那么，-1L < 1U，这是因为int类型的-1U被提升为signed long类型；但-1L > 1UL，这是因为-1L被提升为unsigned long类型，因此它是一个比较大的正数。

在进行赋值时也要进行类型转换，= 右边的值要转换成左边变量的类型，后者即赋值表达式结果的类型。

如前所述，不管是否要进行符号扩展，字符值都要转换成整数值。

当把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丢掉。于是，当程序

```
int  i;
char c;

i = c;
c = i;
```

执行后，c的值保持不变，无论是否要进行符号扩展。然而，如果把两个赋值语句的次序颠倒一下，那么执行后可能会丢失信息。

如果x是float类型且、i是int类型，那么

```
x = I
```

与

```
i = x
```

这两个赋值表达式在执行时都要引起类型转换 ,当把float类型转换成int类型时要把小数部分截取掉；当把float类型转换成int类型时，是四舍五入还是截取取决于具体实现。

由于函数调用的变元是表达式，当把变元传递给函数时也可能引起类型转换。在没有函数原型的情况下，char与short类型转换成int类型，float类型转换成double类型，这就是即使在函数是用char与float类型的变元表达式调用时仍把参数说明成 int与double类型的原因。

最后，在任何表达式中都可以进行显式类型转换（即所谓的“强制转换”），这时要使用一个叫做强制转换的一元运算符。在如下构造中，表达式被按上述转换规则转换成由类型名所指名的类型：

（类型名）表达式

强制转换的精确含义是，表达式首先被赋给类型名指定类型的某个变量，然后再将其用在整个构造所在的位置。例如，库函数 sqrt需要一个double类型的变元，但如果在其他地方作了不适当的处理，那么就会产生无意义的结果（ sqrt是在<math.h>中说明的一个函数）。因而，如果n是整数，那么可以用

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

使得在把n传递给sqrt函数之前先把n的值转换成double类型。注意，强制转换只是以指名的类型产生n的值，n本身的值没有改变。强制转换运算符与其他一元运算符具有相同的优先级，如同本章末尾的表中所总结的那样。

如果变元是通过函数原型说明的，那么在通常情况下，当该函数被调用时，系统对变元自动进行强制转换。于是，对于sqrt的函数原型

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

调用

```
root = sqrt(2);
```

不需要强制转换运算符就自动将把整数2强制转换成double类型的值2.0。

在标准库中包含了一个用于实现伪随机数发生器的函数rand与一个用于初始化种子的函数srand。在前一个函数中使用了强制转换：

```
unsigned long int next = 1;

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

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

练习2-3 编写函数htoi(s)，把由十六进制数字组成的字符串（前面可能包含0x或0X）转换成等价的整数值。字符串中允许的数字为：0~9，a~f、以及A~F。

## 2.8 加一与减一运算符

C语言为变量增加与减少提供了两个奇特的运算符。加一运算符++用于使其运算分量加1，减一运算符--用于使其运算分量减1。我们常常用++运算符来使变量的值加1，如在下述语句中一样：

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

++与--这两个运算符奇特的方面在于，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++）。在这两种情况下，效果都是使n加1。但是，它们之间仍存在一点区别，表达式

```
++n
```

在n的值被使用之前先使n加1，而表达式

```
n++
```

则是在n的值被使用之后再使n加1。这意味着，在该值被使用的上下文中，++n和n++的效果是不同的。如果n的值是5，那么

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

将x的值置为5，而

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

则将x的值置为6。在这两个语句执行完后，n的值都是6。加一与减一运算符只能作用于变量，诸如

```
(i + j)++
```

一类的表达式是非法的。

在除了加1的运算效果，不需要任何具体值的地方，如表达式

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

中，++作为前缀与后缀效果是一样的。在有些情况下需要特别指定。例如，考虑下面的函数squeeze(s,c)，它用于从字符串s中把所有出现的字符c都删除掉：

```
/* squeeze：从s中删除掉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';
}
```

每当出现一个不等于c的字符时，就把它拷贝到j的当前值所指向的位置，并将j的值加1，以准备处理下一个字符。其中的if语句完全等价于

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

具有类似构造的另一个例子是第1章的getline函数。我们可以将这个函数中的if语句

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

用更为精致的if语句

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

代替。

作为第三个例子，再看一下标准函数 strcat(s,t)，它用于把字符串 t连接到字符串 s的后面。strcat函数假定在s中有足够的空间来保存这两个字符串连接的结果。下面所编写的这个函数没有返回任何值（在标准库中，这个函数要返回一个指向新字符串的指针）：

```
/* strcat：把字符串t连接到字符串s的后面；s必须有足够大的空间  */
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')    /*  拷贝t  */
         ;
}
```

在将t中的字符逐个拷贝到s后面时，用后缀运算符 ++ 作用于i与j，以保证在循环过程中i与j均指向下一个位置。

练习2-4　重写squeeze(s1,s2)函数，把字符串s1中与字符串s2中字符匹配的各个字符都删除掉。

练习2-5　编写函数any(s1,s2)，它把字符串s2中任一字符在字符串s1中的第一次出现的位置作为结果返回。如果s1中没有包含s2中的字符，那么返回- 1。（标准库函数strpbrk具有同样的功能，但它返回的是指向该位置的指针。）

## 2.9 按位运算符

C语言提供了六个用于位操作的运算符，这些运算符只能作用于整数分量，即只能作用于有符号或无符号的char、short、int与long类型：

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

按位与运算符&经常用于屏蔽某些位，例如：

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

用于将n除7个低位外的各位置成0。

按位或运算符 | 用于打开某些位，例如：

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

用于将x中与SET_ON中为1的位对应的那些位也置为1。

按位异或运算符^用于在其两个运算分量的对应位不相同时置该位为 1，否则，置该位为0。

我们必须将按位运算符 &和 | 同逻辑运算符&&和 || 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的结果是0，而x && y的值则为1。

移位运算符<<与>>分别用于将左运算分量左移与右移由右运算分量所指定的位数（右运算分量的值必须是正的）。于是，表达式x << 2用于将x的值左移2位，右边空出的2位用0填空，这个表达式的结果等于左运算分量乘以 4。当右移无符号量时，左边空出的部分用 0填空；当右移有符号的量时，在某些机器上对左边空出的部分用符号位填空（即“算术移位”），而在另一些机器上对左边空出的部分则用0填空（即“逻辑移位”）。

一元~ 运算符用于求整数的反码，即它分别将运算分量各位上的 1转换成0，0转换成1。例如：

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

用于将x的最后六位置为0。注意，表达式x &~077是独立于字长的，它要比诸如 x & 0177700一类的表达式好，后者假定x是16位的量。这种可移植的形式并没有增加额外开销，因为 ~ 077是常量表达式，可以在编译时求值。

为了对某些按位运算符做进一步说明，考虑函数 getbits(x, p, n)，它用于返回x从p位置开始的（右对齐的） n位的值。假定第 0位是最右边的一位， n与p都是符合情理的正值。例如，getbits(x, 4, 3)返回右对齐的第4、3、2共三位：

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

其中的表达式 x >> (p+1-n) 将所希望的位段移到字的右边。 ~ 0 将所有位都置为1，~ 0 << n将(~) 0左移n位，将最右边的n位用0填空。再对这个表达式求反，将最右边 n位置为1，其余各位置为0。

练习2-6 编写一个函数setbits(x, p, n, y)，返回对x做如下处理得到的值：x从第p位开始的n位被置为y的最右边n位的值，其余各位保持不变。

练习2-7 编写一个函数invert(x, p, n)，返回对x做如下处理得到的值：x从第p位开始的n位被求反（即，1变成0，0变成1），其余各位保持不变。

练习2-8 编写一个函数rightrot (x, n)，返回将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

在一个赋值表达式⊖中，如果赋值运算符左边的变量在右边紧接着又要重复一次，如：

```
i = i +2
```

那么可以将这种表达式改写成更精简的形式：

```
i += 2
```

---

⊖ 作者这里所说的赋值运算符实际上专指复合赋值运算符（ +=、-=、*=、/=、%=、>>=、<<=、&=、^=与!=），没有包含简单赋值运算符（=）。严格来讲，赋值运算符应包含简单赋值运算符与复合赋值运算符两类。——译者注

其中的运算符 += 叫做赋值运算符。

大多数二元运算符（即有左右两个运算分量的运算符）都有一个对应的赋值运算符 *op*=，*op* 是下面这些运算符中的一个：

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

且表达式

表达式 *op* = 表达式$_2$

等价于

表达式$_1$ = (表达式$_1$) *op* ( 表达式$_2$ )

区别在于，在前一种形式中表达式$_1$只计算一次。注意，表达式$_1$与表达式$_2$两边的圆括号，它们是不可少的，如：

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

的意思是：

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

而不是：

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

例如，下面的函数bitcount用于统计其整数变元中值为1的位的个数：

```
/* bitcount：统计x中值为1的位数 */
int bitcount (unsigned x )
{
     int  b;

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

将x说明为无符号整数是为了保证：当将 x右移时，不管该函数运行于什么机器上，左边空出的各位能用0（而不是符号位）填满。

除了简明外，这类赋值运算符还有一个其表示方式与人们的思维习惯比较接近的优点。我们通常会说“把2加到i上”或“i加上2”，而不会说“取i，加上2，再把结果放回到i中”，因此，表达式i += 2比i = i + 2好。此外，对于诸如

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

等更复杂的表达式，这种赋值运算符使程序代码更易于理解，读者不需要煞费苦心地去检查两个长表达式是否完全一样，也无需为两者为什么不一样而感到疑惑不解。而且，这种赋值运算符还有助于编译程序产生高效的目标代码。

我们已经看到，赋值语句[⊖]有一个值，而且可以用在表达式中。最常见的例子是：

---

⊖ ANSI C中没有使用赋值语句这一术语，这里似乎应叫做赋值表达式。而且语句以分号结束，作为语句不能出现在表达式中。 —— 译者注

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

其他赋值运算符（即复合赋值运算符+=、-=等）也可以用在表达式中，尽管这种用法比较少。

在所有这类表达式中，赋值表达式的类型就是左运算分量的类型，值也是在赋值后左运算分量的值。

练习2-9 在求反码时，表达式x &= ( x - 1 )用于把x最右边的值为1的位删除掉。请解释一下这样做的道理。用这一方法重写bitcount函数，使之执行得更快一点。

## 2.11 条件表达式

语句

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

用于求a与b中的最大值并将之放到z中。作为另一种方法，可以用条件表达式（使用三元运算符?:）来写这段程序及类似的代码段。在表达式

表达式$_1$ ? 表达式$_2$ : 表达式$_3$

中，首先计算表达式$_1$，如果其值不等于0（即为真），则计算表达式$_2$的值，并以该值作为本条件表达式的值；否则计算表达式$_3$的值，并以该值作为本条件表达式的值。在表达式$_2$与表达式$_3$中，只有一个会被计算到。因此，以上语句可以改写成：

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

应该注意到，条件表达式就是一种表达式，它可以用在其他表达式能用的所有地方。如果表达式$_2$与表达式$_3$具有不同的类型，那么结果的类型由本章前面讨论的转换规则决定。例如，如果f为float类型，n为int类型，那么表达式

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

的类型为float，无论n是不是正的。

条件表达式中用于括住第一个表达式的圆括号并不是必需的，这是因为条件运算符 ?: 的优先级非常低，仅高于赋值运算符。但我们还是建议使用圆括号，因为这可以使表达式的条件部分更易于阅读。

条件表达式可用于编写简洁的代码。例如，下面的循环语句用于打印一个数组的 n个元素，每行打印10个元素，每一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跟一个空格，这看起来有点麻烦，但要比相应的if-else结构紧凑。下面是另一个使用条件运算符的好例子：

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

练习2-10 重写用于将大写字母转换成小写字母的函数lower，用条件表达式替代其中的if-

else结构。

## 2.12 运算符优先级与表达式求值次序

表2-1总结了所有运算符的优先级与结合律规则，包括尚未讨论的一些规则。同一行的各个运算符具有相同的优先级，纵向看越往下优先级越低。例如，*、/与%三者具有相同的优先级，它们的优先级都比二元+与-运算符高。运算符( )指函数调用。运算符->与.用于访问结构成员，第6章将讨论这两个运算符以及sizeof（对象大小）运算符。第5章将讨论运算符*（用指针间接访问）与&（对象的地址），第3章将讨论逗号（,）运算符。

表2-1 运算符优先级与结合律

| 运 算 符 | 结 合 律 |
|---|---|
| ( ) [ ] -> . | 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 |

注：一元+、-与*运算符的优先级比相应二元运算符高。

注意，按位运算符 &、^ 与 | 的优先级比相等运算符 == 与 != 低。这意味着，在诸如

```
if  ( (x & MASK) == 0)
   …
```

中，位测试表达式必须用圆括号括起来，才能得到正确的结果。

如同大多数语言一样，C语言没有指定同一运算符的几个运算分量的计算次序（&&、||、?:与，除外）。例如，在诸如

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

一类的语句中，f( )可以在g( )之前计算，也可以在g( )之后计算。因此，如果函数f或g中改变了另一个函数所要使用的变量的值，那么x的结果值可能依赖于这两个函数的计算次序。为了保证特定的计算次序，可以把中间结果保存到临时变量中。

同样，在函数调用中各个变元的求值次序也是未指定的。因而，函数调用语句

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

对不同的编译程序可能会产生不同的结果（视 n加一运算是在power调用之前还是之后而定）。为了解决这一问题，可把该语句改写成

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

函数调用、嵌套的赋值语句、加一与减一运算符都有可能引起“副作用”——作为表达式求值的副产品，改变了某些变量的值。在涉及到副作用的表达式中，对作为表达式一部分的本来的求值次序存在着微妙的依赖关系。下面的表达式语句是这种使人讨厌的情况的一个典型例子：

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

问题是，数组下标的值i是旧值还是新值。编译程序对之可以有不同的解释，并视不同的解释产生不同的结果。C语言标准故意留下了许多诸如此类的问题未作具体规定。何时处理表达式中的副作用（对变量赋值）是各个编译程序的事情，因为最好的求值次序取决于机器结构。（标准明确规定了所有变元的副作用都必须在该函数被调用之前生效，但这对上面对 printf函数的调用没有什么好处。）

从风格角度看，在用任何语言编写程序时，编写依赖于求值次序的代码不是一种好的程序设计习惯。很自然地，我们需要知道哪些事情需要避免，但如果不知道它们在各种机器上是如何执行的，那么不要试图去利用特定的实现。