第14章

预处理器

编译一个 C 程序涉及很多步骤。其中第 1 个步骤被称为预处理(preprocessing)阶段。C 预处理器(preprocessor)在源代码编译之前对其进行一些文本性质的操作。它的主要任务包括删除注释、插入被#include 指令包含的文件的内容、定义和替换由#define 指令定义的符号以及确定代码的部分内容是否应该根据一些条件编译指令进行编译。

14.1 预定义符号

表 14.1 总结了由**预处理器**定义的符号。它们的值或者是字符串常量，或者是十进制数字常量。__FILE__和__LINE__在确认调试输出的来源方面很有用处。__DATE__和__TIME__常常用于在被编译的程序中加入版本信息。__STDC__用于那些在 ANSI 环境和非 ANSI 环境都必须进行编译的程序中结合条件编译（本章稍后描述）。

表 14.1　　预处理器符号

符　号	样例值	含义
__FILE__	“name.c”	进行编译的源文件名
__LINE__	25	文件当前行的行号
__DATE__	“Jan 31 1997”	文件被编译的日期
__TIME__	“18:04:30”	文件被编译的时间
__STDC__	1	如果编译器遵循 ANSI C，其值就为 1，否则未定义

14.2 #define

你已经见过#define 指令的一些简单用法，就是为数值命名一个符号。在本节，我将介绍#define 指令的更多用途。首先让我们观察一下它的更为正式的描述。

```
#define  name   stuff
```

有了这条指令以后，每当有符号 name 出现在这条指令后面时，预处理器就会把它替换成 stuff。

K&R C

早期的 C 编译器要求#出现在每行的起始位置，不过它的后面可以跟一些空白。在 ANSI C 中，这条限制被取消了。

替换文本并不仅限于数值字面值常量。使用#define 指令，你可以把**任何**文本替换到程序中。这里有几个例子：

```
#define reg             register
#define do_forever      for(;;)
#define CASE            break;case
```

第 1 个定义只是为关键字 register 创建了一个简短的别名。这个较短的名字使各个声明更容易通过制表符进行排列。第 2 条声明用一个更具描述性的符号来代替一种用于实现无限循环的 for 语句类型。最后一个#define 定义了一种简短记法，以便在 switch 语句中使用。它自动地把一个 break 放在每个 case 之前，这使得 switch 语句看上去更像其他语言的 case 语句。

如果定义中的 stuff 非常长，它可以分成几行，除了最后一行之外，每行的末尾都要加一个反斜杠，如下面的例子所示：

```
#define DEBUG_PRINT     printf( "File %s line %d:" \
                            " x=%d, y=%d, z=%d", \
                            __FILE__, __LINE__, \
                            x, y, z )
```

我利用了相邻的字符串常量被自动连接为一个字符串这个特性。当你调试一个存在许多涉及一组变量的不同计算过程的程序时，这种类型的声明非常有用。你可以很容易地插入一条调试语句打印出它们的当前值。

```
x *= 2;
y += x;
z = x * y;
DEBUG_PRINT;
```

警告：

这条语句在 DEBUG_PRINT 后面加了一个分号，所以你不应该在宏定义的尾部加上分号。如果你这样做了，结果就会产生两条语句——一条 printf 语句后面再加一条空语句。有些场合只允许出现一条语句，如果放入两条语句就会出现问题，例如：

```
if( ... )
        DEBUG_PRINT;
else
            ...
```

你也可以使用#define 指令把一序列语句插入到程序中。这里有一个完整循环的声明：

```
#define PROCESS_LOOP                        \
        for( i = 0; i < 10; i += 1 ){       \
                sum += i;                   \
                if( i > 0 )                 \
                        prod *= i;          \
        }
```

提示：

不要滥用这种技巧。如果相同的代码需要出现在程序的几个地方，通常更好的方法是把它实现为一个函数。本章后面我将详细讨论#define 宏和函数之间的优劣。

14.2.1　宏

#define 机制包括了一个规定，允许把参数替换到文本中，这种实现通常称为**宏**(macro)或定义宏(defined macro)。下面是宏的声明方式：

```
#define  name(parameter-list)   stuff
```

其中，parameter-list（参数列表）是一个由逗号分隔的符号列表，它们可能出现在 stuff 中。参数列表的左括号必须与 name 紧邻。如果两者之间有任何空白存在，参数列表就会被解释为 stuff 的一部分。

当宏被调用时，名字后面是一个由逗号分隔的值的列表，每个值都与宏定义中的一个参数相对应，整个列表用一对括号包围。当参数出现在程序中时，与每个参数对应的实际值都将被替换到 stuff 中。

这里有一个宏，它接受一个参数：

```
#define SQUARE(x)        x * x
```

如果在上述声明之后，你把

```
SQUARE( 5 )
```

置于程序中，预处理器就会用下面这个表达式替换上面的表达式：

```
5 * 5
```

警告：

但是，这个宏存在一个问题。观察下面的代码段：

```
a = 5;
printf("%d\n", SQUARE( a + 1 ) );
```

乍一看，你可能觉得这段代码将打印 36 这个值。事实上，它将打印 11。想知道为什么？请观察被替换的宏文本。参数 x 被文本 a + 1 替换，所以这条语句实际上变成了

```
printf("%d\n", a + 1 * a + 1 );
```

现在问题清楚了：由替换产生的表达式并没有按照预想的次序进行求值。

在宏定义中加上两个括号，这个问题便很轻松地解决了：

```
#define SQUARE(x)        (x) * ( x )
```

在前面那个例子里，预处理器现在将用下面这条语句执行替换，从而产生预期的结果。

```
printf("%d\n", ( a + 1 ) * ( a + 1 ) );
```

这里有另外一个宏定义。

```
#define  DOUBLE(x)        (x) + (x)
```

定义中使用了括号，用于避免前面出现的问题。但是，使用这个宏，可能会出现另外一个不同的错误。下面这段代码将打印出什么值？

```
a = 5;
printf("%d\n", 10 * DOUBLE( a ) );
```

警告：

看上去，它好像将打印 100，但事实上它打印的是 55。再一次，通过观察宏替换产生的文本，我们能够发现问题所在：

```
printf("%d\n", 10 * ( a ) + ( a ) );
```

乘法运算在宏所定义的加法运算之前执行。这个错误很容易修正：在定义宏时，你只要在整个表达式两边加上一对括号就可以了。

```
#define  DOUBLE(x)        ( (x) + (x) )
```

提示：

所有用于对数值表达式进行求值的宏定义都应该用这种方式加上括号，避免在使用宏时，由于参数中的操作符或邻近的操作符之间不可预料的相互作用。

下面是一对有趣的宏：

```
#define  repeat          do
#define   until(x)       while( ! (x) )
```

这两个宏创建了一种“新”的循环，其工作过程类似于其他语言中的 repeat/until 循环。它按照下面这样的方式使用：

```
repeat {
        statements
} until( i >= 10 );
```

预处理器将用下面的代码进行替换。

```
do {
        statements
} while( ! ( i >= 10 ) );
```

表达式 i>=10 两边的括号用于确保在!操作符执行之前先完成这个表达式的求值。

提示：

创建一套#define 宏，用一种看上去很像其他语言的方式编写 C 程序是完全可能的。在绝大多数情况下，你应该避免这种诱惑，因为这样编写出来的程序使其他 C 程序员很难理解。他们必须时常查阅这些宏的定义以便弄清实际的代码是什么意思。即使每个和这个项目生命期各个阶段相关的人都熟悉那种被模仿的语言，这个技巧仍然可能引起混淆，因为准确地模仿其他语言的各个方面是极为困难的。

14.2.2 #define 替换

在程序中扩展#define 定义符号和宏时，需要涉及几个步骤。

1．在调用宏时，首先对参数进行检查，看看是否包含了任何由#define 定义的符号。如果是，它们首先被替换。

2．替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被它们的值所替代。

3．最后，再次对结果文本进行扫描，看看它是否包含了任何由#define 定义的符号。如果是，就重复上述处理过程。

这样，宏参数和#define 定义可以包含其他#define 定义的符号。但是，宏不可以出现递归。

当预处理器搜索#define 定义的符号时，字符串常量的内容并不进行检查。你如果想把宏参数插入到字符串常量中，可以使用两种技巧。首先，邻近字符串自动连接的特性使我们很容易把一个字符串分成几段，每段实际上都是一个宏参数。这里有一个这种技巧的例子：

```
#define PRINT(FORMAT,VALUE)                \
        printf( "The value is " FORMAT "\n", VALUE )
...
PRINT( "%d", x + 3 );
```

这种技巧只有当字符串常量作为宏参数给出时才能使用。

第 2 个技巧使用预处理器把一个宏参数转换为一个字符串。#argument 这种结构被预处理器翻译为“argument”。这种翻译可以让你像下面这样编写代码：

```
#define PRINT(FORMAT,VALUE)                \
        printf( "The value of " #VALUE   \
        " is " FORMAT "\n", VALUE )
...
PRINT( "%d", x + 3 );
```

它将产生下面的输出：

```
The value of x + 3 is 25
```

##结构则执行一种不同的任务。它把位于它两边的符号连接成一个符号。作为用途之一，它允许宏定义从分离的文本片段创建标识符。下面这个例子使用这种连接把一个值添加到几个变量之一：

```
#define ADD_TO_SUM( sum_number, value ) \
        sum ## sum_number += value
...
ADD_TO_SUM( 5, 25 );
```

最后一条语句把值 25 加到变量 sum5。注意这种连接必须产生一个合法的标识符。否则，其结果就是未定义的。

14.2.3　宏与函数

宏非常频繁地用于执行简单的计算，比如在两个表达式中寻找其中较大（或较小）的一个：

```
#define MAX( a, b )      ( (a) > (b) ? (a) : (b) )
```

为什么不用函数来完成这个任务呢？有两个原因。首先，用于调用和从函数返回的代码很可能比实际执行这个小型计算工作的代码更大，所以使用宏比使用函数在程序的规模和速度方面都更胜一筹。

但是，更为重要的是，函数的参数必须声明为一种特定的类型，所以它只能在类型合适的表达式上使用。反之，上面这个宏可以用于整型、长整型、单浮点型、双浮点数以及其他任何可以用>操作符比较值大小的类型。换句话说，宏是与类型无关的。

和使用函数相比，使用宏的不利之处在于每次使用宏时，一份宏定义代码的拷贝都将插入到程序中。除非宏非常短，否则使用宏可能会大幅度增加程序的长度。

还有一些任务根本无法用函数实现。让我们仔细观察定义于程序 11.1a 的宏。这个宏的第 2 个参数是一种类型，它无法作为函数参数进行传递。

```
#define MALLOC(n, type) \
        ( (type *)malloc( (n) * sizeof( type ) ) )
```

你现在可以观察一下这个宏确切的工作过程。下面这个例子中的第 1 条语句被预处理器转换为第 2 条语句。

```
pi = MALLOC( 25, int );
pi = ( ( int * )malloc( ( 25 ) * sizeof( int ) ) );
```

同样，请注意宏定义并没有用一个分号结尾。分号出现在调用这个宏的语句中。

14.2.4 带副作用的宏参数

当宏参数在宏定义中出现的次数超过一次时，如果这个参数具有副作用，那么当你使用这个宏时就可能出现危险，导致不可预料的结果。**副作用**就是在表达式求值时出现的永久性效果。例如，下面这个表达式

```
x + 1
```

可以重复执行几百次，它每次获得的结果都是一样的。这个表达式不具有副作用。但是

```
x++
```

就具有副作用：它增加 x 的值。当这个表达式下一次执行时，它将产生一个不同的结果。MAX 宏可以证明具有副作用的参数所引起的问题。观察下列代码，你认为它将打印出什么？

```
#define MAX( a, b )      ( (a) > (b) ? (a) : (b) )
...
x = 5;
y = 8;
z = MAX( x++, y++ );
printf( "x=%d, y=%d, z=%d\n", x, y, z );
```

这个问题并不轻松。记住第 1 个表达式是一个条件表达式，用于确定执行另两个表达式中的哪一个，剩余的那个表达式将不会执行。其结果是：x=6，y=10，z=9。

和往常一样，只要检查一下用宏替换后产生的代码，这个奇怪的结果就变得一目了然了。

```
z = ( ( x++ ) > ( y++ ) ? ( x++ ) : ( y++ ) );
```

虽然那个较小的值只增值了一次，但那个较大的值却增值了两次——第 1 次是在比较时，第 2 次在执行?符号后面的表达式时出现。

副作用并不仅限于修改变量的值。下面这个表达式

```
getchar()
```

也具有副作用。调用这个函数将“消耗”输入的一个字符，所以该函数的后续调用将得到不同的字符。如果用户的意图并不是想“消耗”输入字符，那么就不能重复调用这个函数。

考虑下面这个宏。

```
#define EVENPARITY( ch )                          \
        ( ( count_one_bits( ch ) & 1 ) ?          \
        ( ch ) | PARITYBIT : ( ch ) )
```

它使用了程序 5.1 的 count_one_bits 函数，该函数返回它的参数的二进制位模式中 1 的个数。这个宏的目的是产生一个具有偶校验[1]的字符。它首先计数字符中位 1 的个数，如果结果是一个奇数，PARITYBIT 值（一个值为 1 的位）与该字符执行 OR 操作，否则该字符就保留不变。但是，当这个宏以下面这种方式使用时，请想象一下会发生什么？

```
ch = EVENPARITY( getchar() );
```

---

[1] 奇偶校验(parity)是一种错误检测机制。在数据被存储或通过通信线路传送之前，为一个值计算（并添加）一个校验位，使数据的二进制模式中 1 的个数为一个偶数。以后，数据可以通过计算它的位 1 的个数来验证其有效性。如果结果是奇数，那么数据就出现了错误。这个技巧被称为偶校验(even parity)。奇校验(odd parity)的工作原理相同，只是计算并添加校验位之后，数据的二进制位模式中 1 的个数是奇数。

这条语句看上去很合理：读取一个字符并计算它的校验位。但是，它的结果是失败的，因为它实际上读入了两个字符！

14.2.5　命名约定

#define 宏的行为和真正的函数相比存在一些不同的地方，表 14.2 对此进行了总结。由于这些不同之处，所以让程序员知道一个标识符究竟是一个宏还是一个函数是非常重要的。不幸的是，使用宏的语法和使用函数的语法是完全一样的，所以语言本身并不能帮助你区分这两者。

提示：

为宏定义（对于绝大多数由#define 定义的符号也是如此）采纳一种命名约定是很重要的，上面这种混淆就是促使人们这样做的原因之一。一个常见的约定就是把宏名字全部大写。在下面这条语句中，

```
value = max( a, b );
```

max 究竟是一个宏还是一个函数并不明显。你很可能不得不仔细察看源文件以及它所包含的所有头文件来找出它的真实身份。另一方面，请看下面这条语句

```
value = MAX( a, b );
```

命名约定使 MAX 的身份一清二楚。如果宏使用可能具有副作用的参数时，这个约定尤为重要，因为它可以提醒程序员在使用宏之前先把参数存储到临时变量中。

表 14.2　　宏和函数的不同之处

属　　性	#define 宏	函　　数
代码长度	每次使用时，宏代码都被插入到程序中。除了非常小的宏之外，程序的长度将大幅度增长	函数代码只出现于一个地方；每次使用这个函数时，都调用那个地方的同一份代码
执行速度	更快	存在函数调用/返回的额外开销
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境里，除非它们加上括号，否则邻近操作符的优先级可能会产生不可预料的结果	函数参数只在函数调用时求值一次，它的结果值传递给函数。表达式的求值结果更容易预测
参数求值	参数每次用于宏定义时，它们都将重新求值。由于多次求值，具有副作用的参数可能会产生不可预料的结果	参数在函数被调用前只求值一次。在函数中多次使用参数并不会导致多种求值过程。参数的副作用并不会造成任何特殊的问题
参数类型	宏与类型无关。只要对参数的操作是合法的，它可以使用于任何参数类型	函数的参数是与类型有关的。如果参数的类型不同，就需要使用不同的函数；即使它们执行的任务是相同的

14.2.6　#undef

这条预处理指令用于移除一个宏定义。

```
#undef  name
```

如果一个现存的名字需要被重新定义，那么它的旧定义首先必须用#undef 移除。

14.2.7　命令行定义

许多 C 编译器提供了一种能力，允许你在命令行中定义符号，用于启动编译过程。当我们根据

同一个源文件编译一个程序的不同版本时，这个特性是很有用的。例如，假定某个程序声明了一个某种长度的数组。如果某个机器的内存很有限，这个数组必须很小，但在另一个内存充裕的机器上，你可能希望数组能够大一些。如果数组是用类似下面的形式进行声明的，

```
int        array[ARRAY_SIZE];
```

那么，在编译程序时，ARRAY_SIZE 的值可以在命令行中指定。

在 UNIX 编译器中，-D 选项可以完成这项任务。我们可以用两种方式使用这个选项。

```
-Dname
-Dname=stuff
```

第 1 种形式定义了符号 name，它的值为 1。第 2 种形式把该符号的值定义为等号后面的 stuff。用于 MS-DOS 的 Borland C 编译器使用相同的语法提供相同的功能。请查阅你的编译器文档，获取和你的系统有关的信息。

回到我们的例子，在 UNIX 系统中，编译这个程序的命令行可能是下面这个样子：

```
cc -DARRAY_SIZE=100 prog.c
```

这个例子说明了在程序中使用诸如数组长度这样的参数化量的另一个好处。如果在数组的声明中，它的长度以字面值常量的形式给出，或者如果需要在循环内部用一个字面值常量作为限量访问数组，这种技巧就无法使用。在你需要引用数组长度的地方，都必须使用符号常量。

提供符号命令行定义的编译器通常也提供在命令行中去除符号的定义。在 UNIX 编译器上，-U 选项用于执行这项任务。指定-Uname 将导致程序中符号 name 的初始定义被忽略。当它与条件编译结合使用时，这个特性是很有用的。

14.3 条件编译

在编译一个程序时，如果我们可以选择某条语句或某组语句进行翻译或者被忽略，常常会显得很方便。只用于调试程序的语句就是一个明显的例子。它们不应该出现在程序的产品版本中，但是你可能并不想把这些语句从源代码中物理删除，因为如果需要一些维护性修改时，你可能需要重新调试这个程序，还需要这些语句。

条件编译(conditional compilation)就是用于实现这个目的。使用条件编译，你可以选择代码的一部分是被正常编译还是完全忽略。用于支持条件编译的基本结构是#if 指令和与其匹配的#endif 指令。下面显示了它最简单的语法形式。

```
#if  constant-expression
          statements
#endif
```

其中，constant-expression（常量表达式）由预处理器进行求值。如果它的值是非零值（真），那么 statements 部分就被正常编译，否则预处理器就安静地删除它们。

所谓常量表达式，就是说它或者是字面值常量，或者是一个由#define 定义的符号。如果变量在执行期之前无法获得它们的值，那么它们如果出现在常量表达式中就是非法的，因为它们的值在编译时是不可预测的。

例如，将你所有的调试代码都以下面这种形式出现：

```
#if DEBUG
        printf( "x=%d, y=%d\n", x, y );
#endif
```

这样，不管我们是想编译还是忽略这个代码都很容易办到。如果想要编译它，只要使用

```
#define DEBUG  1
```

这个符号定义就可以了。如果想要忽略它，只要把这个符号定义为 0 就可以了。无论哪种情况，这段代码都可以保留在源文件中。

条件编译的另一个用途是在编译时选择不同的代码部分。为了支持这个功能，#if 指令还具有可选的#elif 和#else 子句。完整的语法如下所示：

```
#if constant-expression
        statements
#elif constant-expression
        other statements ...
#else
        other statements
#endif
```

#elif 子句出现的次数可以不限。每个 constant-expression（常量表达式）只有当前面所有常量表达式的值都为假时才会被编译。#else 子句中的语句只有当前面所有的常量表达式的值都为假时才会被编译，在其他情况下它都会被忽略。

K&R C

最初的 K&R C 并不具有#elif 指令。但是，在这类编译器中，可以使用嵌套的指令来获得相同的效果。

下面这个例子取自一个以几个不同版本进行销售的程序。每个版本都有一组不同的选项特性。编写这个代码的困难在于如何让它产生不同的版本。你必须避免为每个版本编写一组不同的源文件，这个代价太大了！因为各组源文件的绝大多数代码都是一样的，而且维护这个程序将成为一个恶梦。幸运的是，条件编译可以解决这个问题。

```
if( feature_selected == FEATURE1 )
#if      FEATURE1_ENABLED_FULLY
        feature1_function( arguments );
#elif    FEATURE1_ENABLED_PARTIALLY
        feature1_partial_function( arguments );
#else
        printf( "To use this feature, send $39.95;"
            " allow ten weeks for delivery.\n" );
#endif
```

这样，我们就只需要编写一组源文件。当它们被编译时，每个当前版本所需的特性（或特性层次）符号被定义为 1，其余的符号被定义为 0。

14.3.1　是否被定义

测试一个符号是否已被定义也是可能的。在条件编译中完成这个任务往往更为方便，因为程序如果并不需要控制编译的符号所控制的特性，它就不需要被定义。这个测试可以通过下列任何一种方式进行：

```
#if      defined(symbol)
#ifdef   symbol

#if      !defined(symbol)
#ifndef  symbol
```

每对定义的两条语句是等价的，但#if 形式功能更强。因为常量表达式可能包含额外的条件，如下面所示：

```
#if X > 0 || defined( ABC ) && defined( BCD )
```

K&R C

有些 K&R C 编译器可能并未包含所有这些功能，这取决于它们的年代如何久远。

14.3.2 嵌套指令

前面提到的这些指令可以嵌套于另一个指令内部，如下面的代码段所示：

```
#if      defined( OS_UNIX )
         #ifdef  OPTION1
                 unix_version_of_option1();
         #endif
         #ifdef  OPTION2
                 unix_version_of_option2();
         #endif
#elif    defined( OS_MSDOS )
         #ifdef  OPTION2
                 msdos_version_of_option2();
         #endif
#endif
```

在这个例子中，操作系统的选择将决定不同的选项可以使用哪些方案。这个例子同时说明了预处理器指令可以在它们前面添加空白，形成缩进，从而提高可读性。

为了帮助读者记住复杂的嵌套指令，为每个#endif 加上一个注释标签是很有帮助的，标签的内容就是#if（或#ifdef）后面的那个表达式。当#if（或#ifdef）和#endif 之间的代码块非常长时，这种做法尤为有用。例如：

```
#ifdef  OPTION1
        lengthy code for option1;
#else
        lengthy code for alternative;
#endif  /* OPTION1 */
```

有些编译器允许一个符号出现于#endif 指令中，它的作用和上面这种标签类似。不过这个符号对实际代码不会产生任何作用。标准并没有提及这种做法是否合法，所以更安全的做法还是使用注释。

14.4 文件包含

你已经看到过，#include 指令使另一个文件的内容被编译，就像它实际出现于#include 指令出现的位置一样。这种替换执行的方式很简单：预处理器删除这条指令，并用包含文件的内容取而代之。这样，一个头文件如果被包含到 10 个源文件中，它实际上被编译了 10 次。

提示：

这个事实意味着使用#include 文件涉及一些开销，但基于两个十分充分的理由，你不必担心这种开销。首先，这种额外开销实际上并不大。如果两个源文件都需要同一组声明，把这些声明复制到每个源文件中所花费的编译时间跟把这些声明放入一个头文件，然后再用#include 指令把它包含于每个源文件所花费的编译时间相差无几。同时，这个开销只是在程序被编译时才存在，所以对运行时效率并无影响。但是，更为重要的是，把这些声明放于一个头文件中具有重要的意义。如果其他源文件还需要这些声明，你就不必把这些拷贝逐一复制到这些源文件中，因此它们的维护任务也变得简单了。

提示：

当头文件被包含时，位于头文件内的所有内容都要被编译。这个事实意味着每个头文件只应该包含一组函数或数据的声明。和把一个程序需要的所有声明都放入一个巨大的头文件相比，使用几个头文件，每个头文件包含用于某个特定函数或模块的声明的做法更好一些。

提示：

程序设计和模块化的原则也支持这种方法。只把必要的声明包含于一个文件中这种做法更好一些，这样文件中的语句就不会意外地访问应该属于私有的函数或变量。同时，这种方法使你不需要在数百行不相关的代码中寻找你所需要的那组声明，因此它们的维护工作也更容易一些。

14.4.1　函数库文件包含

编译器支持两种不同类型的#include 文件包含：函数库文件和本地文件。事实上，它们之间的区别很小。

函数库头文件包含使用下面的语法。

```
#include <filename>
```

对于 filename，并不存在任何限制，不过根据约定，标准库文件以一个.h 后缀[1]结尾。

编译器通过观察由编译器定义的“一系列标准位置”查找函数库头文件。你所使用的编译器的文档应该说明这些标准位置是什么，以及你怎样修改它们或者在列表中添加其他位置。例如，在典型情况下，运行于 UNIX 系统上的 C 编译器在/user/include 目录查找函数库头文件。这种编译器有一个命令行选项，允许你把其他目录添加到这个列表中，这样你就可以创建你自己的头文件函数库。同样，请查阅你使用的编译器的文档，看看你的系统在这方面是怎样规定的。

14.4.2　本地文件包含

下面是#include 指令的另一种形式。

```
#include "filename"
```

标准允许编译器自行决定是否把本地形式的#include 和函数库形式的#include 区别对待。你可以对本地头文件先使用一种特殊的处理方式，如果失败，编译器再按照函数库头文件的处理方式对它们进行处理。处理本地头文件的一种常见策略就是在源文件所在的当前目录进行查找，如果该头文

[1] 从技术上说，函数库头文件并不需要以文件的形式存储，但对于程序员而言，这并非显而易见。

件并未找到，编译器就像查找函数库头文件一样在标准位置查找本地头文件。

你可以在所有的#include 语句中使用双引号而不是尖括号。但是，使用这种方法，有些编译器在查找函数库头文件时可能会浪费少许时间。对函数库头文件使用尖括号的另一个较好的理由是它能给读者提供一些信息。使用尖括号，下面这条语句

```
#include <errno.h>
```

显然引用的是一个函数库头文件。如果使用另一种形式，

```
#include "errno.h"
```

你就无法弄清楚这个和上面相同的文件到底是一个函数库头文件还是一个本地头文件。要想弄明白它究竟是哪种类型？唯一的方法是检查执行编译过程的目录。

UNIX 系统和 Borland C 编译器所支持的一种变体形式是使用绝对路径名(absolute pathname)，它不仅指定文件的名字，而且指定了文件的位置。UNIX 系统中的绝对路径名以一个斜杠开头，如下所示：

```
/home/fred/C/my_proj/declaration2.h
```

在 MS-DOS 系统中，它所使用的是反斜杠而不是斜杠。如果一个绝对路径名出现在任何一种形式的#include，那么正常的目录查找就被跳过，因为这个路径名指定了头文件的位置。

14.4.3 嵌套文件包含

在一个将被其他文件包含的文件中使用#include 指令是可能的。例如，考虑一组读取输入并且执行各种输入有效性验证任务的函数。函数返回的是被验证后的数据，如果到达文件尾时就返回常量 EOF。

这些函数的原型将被放入一个头文件中，并且用#include 指令包含到需要使用这些函数的源文件中。但是，每个使用 I/O 函数的文件必须同时包含 stdio.h 以获得 EOF 的声明。因此，包含这些函数原型的头文件也可能包含一条：

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

包含了这个头文件就自动引入了标准 I/O 声明。

标准要求编译器必须支持至少 8 层的头文件嵌套，但它并没有限定嵌套深度的最大值。事实上，我们并没有很好的理由让#include 指令的嵌套深度超过一层或两层。

提示：

嵌套#include 文件的一个不利之处在于它使得我们很难判断源文件之间的真正依赖关系。有些程序，如 UNIX 的 make 实用工具，必须知道这些依赖关系以便决定当某些文件被修改之后，哪些文件需要重新编译。

嵌套#include 文件的另一个不利之处在于一个头文件可能会被多次包含。为了说明这种错误，考虑下面的代码：

```
#include "x.h"
#include "x.h"
```

显然，这里文件 x.h 被包含了两次。没有人会故意编写这样的代码。但下面的代码

```
#include "a.h"
#include "b.h"
```

看上去没什么问题。如果 a.h 和 b.h 都包含一个嵌套的#include 文件 x.h，那么 x.h 在此处也同样出现了两次，只不过它的形式不是那么明显而已。

多重包含在绝大多数情况下出现于大型程序中，它往往需要使用很多头文件，因此要发现这种情况并不容易。要解决这个问题，我们可以使用条件编译。如果所有的头文件都像下面这样编写：

```
#ifndef _HEADERNAME_H
#define _HEADERNAME_H 1
/*
** All the stuff that you want in the header file
*/
#endif
```

那么，多重包含的危险就被消除了。当头文件第 1 次被包含时，它被正常处理，符号_HEADERNAME_H 被定义为 1。如果头文件被再次包含，通过条件编译，它的所有内容被忽略。符号_HEADERNAME_H 按照被包含文件的文件名进行取名，以避免由于其他头文件使用相同的符号而引起的冲突。

注意前一个例子中的定义也可以写作

```
#define _HEADERNAME_H
```

它的效果完全一样。尽管现在它的值是一个空字符串而不是“1”，但这个符号仍然被定义。

但是，你必须记住预处理器仍将读入整个头文件，即使这个文件的所有内容将被忽略。由于这种处理将拖慢编译速度，所以如果可能，应避免出现多重包含，不管它是否由于嵌套的#include 文件导致。

14.5　其他指令

预处理器还支持其他一些指令。首先，当程序编译之后，#error 指令允许你生成错误信息。下面是它的语法：

```
#error  text of error message
```

下面的代码段显示了你可以如何使用这个指令。

```
#if     defined( OPTION_A )
        stuff needed for option A
#elif   defined( OPTION_B )
        stuff needed for option B
#elif   defined( OPTION_C )
        stuff needed for option C
#else
        #error No option selected!
#endif
```

另外还用一种用途较小的#line 指令，它的形式如下：

```
#line     number  "string"
```

它通知预处理器 number 是下一行输入的行号。如果给出了可选部分“string”，预处理器就把它作为当前文件的名字。值得注意的是，这条指令将修改__LINE__符号的值，如果加上可选部分，它还将修改__FILE__符号的值。

这条指令最常用于把其他语言的代码转换为 C 代码的程序。C 编译器产生的错误信息可以引用源文件而不是翻译程序产生的 C 中间源文件的文件名和行号。

#progma 指令是另一种机制，用于支持因编译器而异的特性。它的语法也是因编译器而异。有

些环境可能提供一些#progma 指令，允许一些编译选项或其他任何方式无法实现的一些处理方式。例如，有些编译器使用#progma 指令在编译过程中打开或关闭清单显示，或者把汇编代码插入到 C 程序中。从本质上说，#progma 是不可移植的。预处理器将忽略它不认识的#progma 指令，两个不同的编译器可能以两种不同的方式解释同一条#progma 指令。

最后，无效指令(null directive)就是一个#符号开头，但后面不跟任何内容的一行。这类指令只是被预处理器简单地删除。下面例子中的无效指令通过把#include 与周围的代码分隔开来，凸显它的存在。

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

我们也可以通过插入空行取得相同的效果。

14.6 总结

编译一个 C 程序的第 1 个步骤就是对它进行预处理。预处理器共支持 5 个符号，它们在表 14.1 中描述。

#define 指令把一个符号名与一个任意的字符序列联系在一起。例如，这些字符可能是一个字面值常量、表达式或者程序语句。这个序列到该行的末尾结束。如果该序列较长，可以把它分开数行，但在最后一行之外的每一行末尾加一个反斜杠。宏就是一个被定义的序列，它的参数值将被替换。当一个宏被调用时，它的每个参数都被一个具体的值替换。为了防止可能出现于表达式中的与宏有关的错误，在宏完整定义的两边应该加上括号。同样，在宏定义中每个参数的两边也要加上括号。#define 指令可以用于“重写”C 语言，使它看上去像是其他语言。

#argument 结构由预处理器转换为字符串常量“argument”。##操作符用于把它两边的文本粘贴成同一个标识符。

有些任务既可以用宏也可以用函数实现。但是，宏与类型无关，这是一个优点。宏的执行速度快于函数，因为它不存在函数调用/返回的开销。但是，使用宏通常会增加程序的长度，但函数却不会。同样，具有副作用的参数可能在宏的使用过程中产生不可预料的结果，而函数参数的行为更容易预测。由于这些区别，使用一种命名约定，让程序员很容易地判断一个标识符是函数还是宏是非常重要的。

在许多编译器中，符号可以从命令行定义。#undef 指令将导致一个名字的原来定义被忽略。

使用条件编译，你可以从一组单一的源文件创建程序的不同版本。#if 指令根据编译时测试的结果，包含或忽略一个序列的代码。当同时使用#elif 和#else 指令时，你可以从几个序列的代码中选择其中之一进行编译。除了测试常量表达式之外，这些指令还可以测试某个符号是否已被定义。#ifdef 和#ifndef 指令也可以执行这个任务。

#include 指令用于实现文件包含。它具有两种形式。如果文件名位于一对尖括号中，编译器将在由编译器定义的标准位置查找这个文件。这种形式通常用于包含函数库头文件时。另一种形式，文件名出现在一对双引号内。不同的编译器可以用不同的方式处理这种形式。但是，如果用于处理本地头文件的任何特殊处理方法无法找到这个头文件，那么编译器接下来就使用标准查找过程来寻找它。这种形式通常用于包含你自己编写的头文件。文件包含可以嵌套，但很少需要进行超过一层或两层的文件包含嵌套。嵌套的包含文件将会增加多次包含同一个文件的危险，而且使我们更难以确定某个特定的源文件依赖的究竟是哪个头文件。

#error 指令在编译时产生一条错误信息，信息中包含的是你所选择的文本。#line 指令允许你告诉编译器下一行输入的行号，如果它加上了可选内容，它还将告诉编译器输入源文件的名字。因编译器而异的#progma 指令允许编译器提供不标准的处理过程，比如向一个函数插入内联的汇编代码。

14.7　警告的总结

1. 不要在一个宏定义的末尾加上分号，使其成为一条完整的语句。
2. 在宏定义中使用参数，但忘了在它们周围加上括号。
3. 忘了在整个宏定义的两边加上括号。

14.8　编程提示的总结

1. 避免用#define 指令定义可以用函数实现的很长序列的代码。
2. 在那些对表达式求值的宏中，每个宏参数出现的地方都应该加上括号，并且在整个宏定义的两边也加上括号。
3. 避免使用#define 宏创建一种新语言。
4. 采用命名约定，使程序员很容易看出某个标识符是否为#define 宏。
5. 只要合适就应该使用文件包含，不必担心它的额外开销。
6. 头文件只应该包含一组函数和（或）数据的声明。
7. 把不同集合的声明分离到不同的头文件中可以改善信息隐藏。
8. 嵌套的#include 文件使我们很难判断源文件之间的依赖关系。

14.9　问题

1. 预处理器定义了 5 个符号，给出了进行编译的文件名、文件的当前行号、当前日期和时间以及编译器是否为 ANSI C 编译器。为每个符号举出一种可能的用途。
2. 说出两个使用#define 定义的名字替代字面值常量的优点。
3. 编写一个用于调试的宏，打印出任意的表达式。它被调用时应该接受两个参数。第 1 个是 printf 格式码，第 2 个是需要打印的表达式。
4. 下面的程序将打印出什么？在展开#define 内容时必须非常小心！

```
#define MAX(a,b)          (a)>(b)?(a):(b)
#define SQUARE(x)         x*x
#define DOUBLE(x)         x+x

main()
{
        int     x, y, z;

        y = 2; z = 3;
        x = MAX(y,z);
/* a */ printf( "%d %d %d\n", x, y, z );

        y = 2; z = 3;
        x = MAX(++y,++z);
```

```
/* b */ printf( "%d %d %d\n", x, y, z );

        x = 2;
        y = SQUARE(x);
        z = SQUARE(x+6);
/* c */ printf( "%d %d %d\n", x, y, z );

        x = 2;
        y = 3;
        z = MAX(5*DOUBLE(x),++y);
/* d */ printf( "%d %d %d\n", x, y, z );
}
```

5. putchar 函数定义于文件 stdio.h 中，尽管它的内容比较长，但它是作为一个宏实现。你认为它为什么以这种方式定义？

6. 下列代码是否有错？如果有，错在何处？

```
/*
** Process all the values in the array.
*/
result = 0;
i = 0;
while( i < SIZE ){
        result += process( value[ i++ ] );
}
```

7. 下列代码是否有错？如果有，错在何处？

```
#define SUM( value )    ( ( value ) + ( value ) )
int     array[SIZE];
...
/*
** Sum all the values in the array.
*/
sum = 0;
i = 0;
while( i < SIZE )
        sum += SUM( array[ i++ ] );
```

8. 下列代码是否有错？如果有，错在何处？

在文件 header1.h 中：

```
#ifndef  _HEADER1_H
#define  _HEADER1_H
#include  "header2.h"
    其他声明
#endif
```

在文件 header2.h 中：

```
#ifndef  _HEADER2_H
#define  _HEADER2_H
#include  "header1.h"
    其他声明
#endif
```

9. 在一次提高程序可读性的尝试中，一位程序员编写了下面的声明。

```
#if sizeof( int ) == 2
        typedef long int32;
#else
        typedef int int32;
#endif
```

这段代码是否有错？如果有，错在何处？

14.10　编程练习

★★ 1. 你所在的公司向市场投放了一个程序，用于处理金融交易并打印它们的报表。为了扩展潜在的市场，这个程序以几个不同的版本进行销售，每个版本都有不同选项的组合——选项越多，价格就越高。你的任务是为一个打印函数实现代码，这样它可以很容易地进行编译，产生程序的不同版本。

你的函数名为 print_ledger。它接受一个 int 参数，没有返回值。它应该调用一个或多个下面的函数，具体依取决于该函数被编译时哪个符号（如果有的话）被定义。

如果这个符号被定义...	那么你就调用这个函数
OPTION_LONG	print_ledger_long
OPTION_DETAILED	print_ledger_detailed
(无)	print_ledger_default

每个函数都接受单个 int 参数。把你收到的值传递给你应该调用的函数。

★★ 2. 编写一个函数，返回一个值，提示运行这个函数的计算机的类型。这个函数将由一个能够运行于许多不同计算机的程序使用。

我们将使用条件编译来实现这个魔术。你的函数应该叫作 cpu_type，它不接受任何参数。当你的函数被编译时，在下面表中“已定义”列中的符号之一可能会被定义。你的函数应该从“返回值”列中返回对应的符号。如果左边列中的所有符号均未定义，那么函数就返回 CPU_UNKNOWN 这个值。如果超过一个的符号被定义，那么其结果就是未定义的。

定义符号	返回值
VAX	CPU_VAX
M68000	CPU_68000
M68020	CPU_68020
I80386	CPU_80386
X6809	CPU_6809
X6502	CPU_6502
U3B2	CPU_3B2
(无)	CPU_UNKNOWN

“返回值”列中的符号将被#define 定义为各种不同的整型值，其内容位于头文件 cpu_type.h 中。