第3章 C预处理器

C预处理器是一种简单的宏处理器。从概念上说，它在编译器读取源程序之前对C程序的源文本进行处理。在C的有些实现中，预处理器实际上是一个独立的程序。它读取最初的源文件，并写入到一个新的“经过预处理”的源文件，后者可以作为C编译器的输入。在C的其他实现中，由一个程序一次完成对源文件的预处理和编译。

3.1 预处理器命令

预处理器是由特殊的预处理器命令行控制的，它们是以#符号开头的源文件行。不包含预处理器命令的行称为源程序文本行。表3-1列出了预处理命令。

预处理器一般从源文件中删除所有的预处理器命令行，并在源文件中执行这些预处理命令所指定的转换操作。例如，对一个出现在源程序文本中的宏调用进行展开。经过预处理所产生的源代码文本必须仍然是一个合法的C程序。

预处理器代码行的语法与C语言其他部分的语法完全独立（尽管存在一些相似之处）。例如，把一个宏定义展开为一个语法上不完整的代码片段是完全可能的，只要这个片段在这个宏被调用时的所有上下文环境中都是合理的（也就是被正确地结束）。

表3-1 预处理器命令

命　令	含　义	参见章节
#define	定义一个预处理器宏	3.3
#undef	取消一个预处理器宏的定义	3.3.5
#include	插入另一个源文件的文本	3.4
#if	根据一个常量表示式的值，有条件地包含一些文本	3.5.1
#ifdef	根据一个宏名是否已被定义，有条件地包含一些文本	3.5.3
#ifndef	根据与#ifdef正好相反的测试，有条件地包含一些文本	3.5.3
#else	如果它前面的#if、#ifdef、#ifndef或#elif测试失败，就包含一些文本	3.5.1
#endif	终止条件文本	3.5.1
#line	提供用于编译器信息的行号	3.6
#elif[a]	如果它前面的#if、#ifdef、#ifndef或#elif测试失败，根据另一个常量表达式的值，包含一些文本	3.5.2
defined[a]	如果一个名称已经被定义为预处理器宏，这个预处理器函数返回1，否则返回0。它由#if和#elif所使用	3.5.5
#操作符[b]	用一个包含参数值的字符串常量替换一个宏参数	3.3.8
##操作符[b]	组合2个相邻的标记，创建单个标记	3.3.9
#pragma[b]	指定依赖编译器的信息	3.7
#error[b]	用指定的信息产生一个编译时错误	3.8

a 它原来并不是C的一部分，但现在在ISO和非ISO的C编译器中极为常见。

b 标准C新增的。

3.2 预处理器词法约定

预处理器并不会对源文本进行解析，但是会把它分解为标记，以便查找宏调用。预处理器的词法约定与编译器有点不一样。预处理器能够认识正常的C标记，另外还会把其他在C中并没有被认为是合法C标记的字符也看成为“ 标记 ”。因此，预处理器能够认识文件名、判断是否存在空格以及确定行尾标志符的位置。

以#开始的行被看成是预处理器命令，命令的名称必须紧随#字符之后。标准C允许在有些源代码行的#字符前后出现空格，但一些旧式的编译器不允许这种做法。如果一行中唯一的非空白字符就是#字符，它被标准C看成是null指令，被当成空行处理。有些旧式的编译器可能会采取不同的行为。

在预处理器命令行中，命令名称的后面可能包含了该命令的参数。如果一个预处理器命令不接受任何参数，则命令行的剩余部分应该是空，或者是空白字符或注释。许多ISO标准之前的编译器会无声无息地忽略预期的参数之后的所有字符（如果有）。这可能导致移植性问题。预处理器命令的参数一般受到宏替换的影响。

预处理器行是在宏扩展之前被认识的。因此，如果一个宏被展开之后是一些看上去像预处理器命令这样的东西，这个命令将不会被标准C编译器（以及大多数其他C编译器）所认识（有些老的UNIX系统违反了这个规则）。

例子

下面这些代码的效果是不在被编译的程序中包含math.h文件：

```
/* This example doesn't work as one might think! */
#define GETMATH #include <math.h>
GETMATH
```

展开后的标记序列

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

不会被预处理，而是被编译为（错误的）C代码。

如第2.1.2节所述，所有的源文件行（包括预处理器命令行）都可以在行末加个反斜杠（\）进行续行。这个操作发生在对预处理器命令进行扫描之前。

例子

下面的预处理器命令

```
#define err(flag,msg)  if (flag) \
   printf(msg)
```

与下面的命令相同

```
#define err(flag,msg)  if (flag) printf(msg)
```

如果下面第1行的行末添加了一个反斜杠字符，则这两行

```
#define BACKSLASH \
#define ASTERISK *
```

将被看成是单个预处理器命令

```
#define BACKSLASH #define ASTERISK *
```

如第2.2节所解释的那样，预处理器会把注释当成空白，注释内部的行分隔符并不会终止预处理器命令。

参考：注释 第2.2节；行的终止和延续 第2.1节；标记 第2.3节。

3.3 定义和替换

预处理器命令#define使一个名称（标识符）被定义为预处理器的一个宏。一种称为宏体的标记序列与这个名称相关联。当宏的名称在程序的源文本或者在其他预处理器命令的参数中被确认时，它就被当做是对这个宏的调用，宏名被宏体的一份拷贝所代替。如果这个宏被定义为接受参数，则宏名后面的实际参数就用于替换宏体中的形式参数。

例子

如果带有2个参数的宏sum被定义为

```
#define sum(x,y)  x+y
```

则预处理器就会对源程序行

```
result = sum(5,a*b);
```

执行简单的文本替换（也许并不是程序员所预想的）

```
result = 5+a*b;
```

由于预处理器并不会区分保留字和其他标识符，因此在原则上使用C的保留字作为预处理器的宏名也是可行的。但是，这种做法通常是一种不良的编程实践。预处理器不会认识注释、字符串或字符常量或#include文件名内部的宏名。

3.3.1 类似对象的宏定义

#define命令具有两种形式，取决于被定义的宏名后面是不是紧随一个左括号。最简单的类似对象的格式没有左括号：

```
#define  名称  标记序列opt
```

类似对象的宏不接受参数。它只是通过名称来调用的。在源程序文本中遇到宏名时，这个宏名就被宏体（即相关联的标记序列，可能为空）所替换。#define命令的语法并不要求在宏名后面使用等号或其他任何特殊的分隔符标记。宏体就是在宏名之后开始的。

类似对象的宏特别适用于在程序中引入名称常量，使诸如表的长度这样的“神奇数字”可以在一个地方编写，然后通过名称在其他地方被引用。这样，以后修改这个数字就非常方便了。

类似对象的宏的另一个重要用途是在外部所定义的函数和变量上隔离因编译器而异的限制。第2.5节提供了这方面的一个例子。

例子

下面是一些典型的宏定义：

```
#define BLOCK_SIZE 0x100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define EOT '\004'
#define ERRMSG "*** Error %d: %s.\n"
```

一个常见的编程错误是错误地包含了一个等号：

```
#define NUMBER_OF_TAPE_DRIVES =  5  /* Probably wrong. */
```

这是合法的定义，但它导致NUBER_OF_TAPE_DRIVES被定义为“ = 5 ”而不是“ 5 ”。如果接着编写了下面的代码片段：

```
if (count != NUMBER_OF_TAPE_DRIVES) …
```

它将被展开为：

```
if (count != = 5) …
```

它在语法上是非法的。基于相似的原因，程序员还要避免使用不必要的分号：

```
#define NUMBER_OF_TAPE_DRIVES 5 ;  /* Probably wrong. */
```

参考：复合赋值操作符 第7.9.2节；操作符和分隔符 第2.4节。

3.3.2 定义带参数的宏

更为复杂的、与函数相似的宏定义在一对括号内声明了用逗号分隔的形式参数的名称：

#define 名称(标识符列表$_{opt}$) 标记序列$_{opt}$

其中，标识符列表是个逗号分隔的形式参数名称列表。在C99中，标识符列表中还可以出现省略号（...，三个点），表示可变长度的参数列表。这个问题将在第3.3.10节讨论。在此之前，我们只考虑固定数量的参数列表。

左括号必须紧随宏名之后，中间不能有空格。如果宏名和左括号之间被一个空格所分隔，则这个宏被定义为不接受任何参数，并且宏体从这个左括号开始。

形式参数的名称必须是标识符，并且必须各不相同。宏体内并不一定要提到任何参数的名称（尽管一般都会提到）。类似函数的宏可以具有空的形参列表（即0个形式参数）。这种类型的宏适用于模拟不接受任何参数的函数。

类似函数的宏所接受的实参数量与形参相同。调用这种宏的形式是宏名、一个左括号、与每个形参对应的实际参数标记序列以及一个右括号。实际参数标记序列是由逗号分隔的（当一个不接受任何参数的类似函数的宏被调用时，必须提供一个空的实际参数列表）。当一个宏被调用时，宏名和左括号之间或者实际参数之间可以出现空格（有些旧式的或者有缺陷的预处理器实现并不允许实际参数标记列表延续多行，除非这些行的行末以用一个“ \ ”符号进行续行）。

实际参数标记序列可以包含括号，只要它们被正确地嵌套并且左右括号的数量平衡。并且，它可以包含逗号，只要每个逗号出现在一组括号内部（这个限制是为了防止这些逗号与分隔实际参数的逗号混淆）。类似，花括号和方括号也可以出现在宏的参数中，但它们不能包含逗号，并且并不一定要平衡。出现在字符常量和字符串常量内部的括号和逗号不会在计算括号平衡时被考虑，也不会分隔实际参数。

在C99中，宏的参数可以是空的，也就是不包含任何标记。

例子

下面是一个把两个参数相乘的宏定义：

```
#define product(x,y)  ((x)*(y))
```

在下面这条语句中，它被调用了两次：

```
x = product(a+3,b) + product (c, d);
```

product宏的参数可以是函数调用（或宏调用）。函数参数列表中的逗号并不会影响宏参数的解析：

```
return product( f(a,b), g(a,b) ); /* OK */
```

例子

getchar宏具有空的形参列表：

```
#define getchar()  getc(stdin)
```

当它被调用时，就向它提供空的实参列表：

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

（getchar、stdin和EOF是在标准头文件stdio.h中定义的。）

例子

我们还定义了一个接受一条任意的语句为参数的宏：

```
#define insert(stmt)  stmt
```

下面这个调用

```
insert( {a=1; b=1;} )
```

能够正确地工作。但是，如果我们把两条赋值语句修改为一条包含两个赋值表达式的语句：

```
insert( {a=1, b=1;} )
```

则预处理器则会显示错误信息，表示提供了太多需要插入的宏参数。为了修正这个问题，我们可以把它写为：

```
insert( {(a=1, b=1);} )
```

例子

定义在语句上下文环境中使用的类似函数的宏可能存在一些陷阱。下面这个函数交换它的两个参数x和y的值。假定这两个参数的类型可以转换为unsigned long并且在转换回去之后参数的值不会发生变化，并且不涉及标识符_temp。

```
#define swap(x, y) { unsigned long _temp=x; x=y; y=_temp; }
```

问题在于在swap后面加上一个分号是一种很自然的做法，就像真的把swap看成是个函数一样：

```
if (x > y) swap(x, y); /* Whoops! */
else x = y;
```

这将产生错误，因为宏展开包含了一个额外的分号（第8.1节）。下面，我们把展开后的语句放在单独一行中，更清晰地显示这个问题：

```
if (x > y) { unsigned long _temp=x; x=y; y=_temp; }
;
else x = y;
```

避免这个问题的一个较为聪明的方式是把宏体定义为一条do-while语句，后者可以接受在末尾添加分号（第8.6.2节）。

```
#define swap(x, y) \
   do { unsigned long _temp=x; x=y; y=_temp; } while (0)
```

当遇到一个类似函数的宏调用时，在参数处理之后，整个宏调用都被替换为经过处理的宏体的一份拷贝。参数处理是按下面的方式进行的。实际参数标记字符串与对应的形式参数名称相关联。此时将创建宏体的一份拷贝，其中每个形式参数的名称被与它相关联的实际参数标记序列所代替。然后，这份宏体拷贝就替换了这个宏调用，用经过处理的宏体的一份拷贝替换宏调用的过程称为宏展开。经过处理的宏体的拷贝称为该宏调用的展开。

例子

考虑这个宏定义，它提供了一种常规的方式，通过一个循环，从一个特定值到某个限值（含此限值）进行计数：

```
#define incr(v,low,high)  \
    for ((v) = (low); (v) <= (high); (v)++)
```

为了打印出1-20的整数的立方表，可以编写下面的代码：

```
#include <stdio.h>
int main(void)
{
    int j;
    incr(j, 1, 20)
       printf("%2d %6d\n", j, j*j*j);
    return 0;
}
```

对incr宏的调用被展开为下面这个循环：

```
    for ((j) = (1); (j) <= (20); (j)++)
```

多余的括号保证了复杂的实际参数并不会被编译器错误地解释（参见第3.3.6节）。

参考：do语句 第8.6.2节；语句的语法 第8.1节；unsigned long 第5.1.2节；空格 第2.1.2节。

3.3.3 宏表达式的重新扫描

当一个调宏用被展开之后，对宏调用的扫描就在宏展开的开始位置继续，使宏的名称在被展开宏的内部可以被认识，以便实现进一步的宏替换。宏替换并不是在#define命令的任何部分中执行的，甚至也不是在宏体中执行的，而是在处理命令以及定义宏名时执行的。对于某些特定的宏调用，只有当宏体被展开之后，宏体中的宏名才能被认识。

当一个类似函数的宏调用被扫描时，它的实际参数标记字符串也不会执行宏替换。只有当对展开的宏体重新进行扫描时，预处理器才能认识实际参数标记字符串中的宏名，前提是对应的形式参数在宏体中实际出现了1次或多次（因此导致实际参数标记字符串在展开的宏体中出现1次或多次）。

例子

假设有下面的定义：

```
#define plus(x,y)  add(y,x)
#define add(x,y)  ((x)+(y))
```

下面这个调用：

```
plus(plus(a,b),c)
```

是按照下面的方式展开的。

步骤	结果
1（最初）	plus (plus (a, b), c)
2	add (c, plus (a, b))
3	((c) + (plus (a, b)))
4	((c) + (add (b, a)))
5（最终）	((c) + (((b) + (a)))

在标准C中，出现在自身的展开体中的宏（直接出现或者通过某些嵌套宏展的中间序列）并不会被重新展开。这就允许程序员根据一个宏的旧定义对它进行重新定义。旧式的C预处理器在传统中并不会检测这种递归，因此试图继续展开，直到被某种系统错误所终止。

例子

下面这个宏修改平方根函数的定义，以一种不同常规的方式处理负的参数：

```
#define sqrt(x)  ((x)<0 ? sqrt(-(x)) : sqrt(x))
```

在标准C中，只要这个宏的求值次数不超过1次，它的效果就和预期的相同。但是，它在旧式的编译器中可能会产生错误。下面这个宏也有类似的问题。

```
#define char unsigned char
```

读者可以参见第7.4.3节中一个使用宏来追踪函数调用的有趣例子。

3.3.4 预定义的宏

标准C的预处理器需要定义一些类似对象的宏（表3-2）。这些宏的名称都是以两个下划线字符开始和结束的。程序员不能取消这些预定义宏的定义（使用#undef）或对它们进行重新定义。

__LINE__和__FILE__宏适用于打印某些类型的错误信息。__DATE__和__TIME__宏可以记录编译的时间。__DATE__和__TIME__的宏在编译时为常量值。__LINE__和__FILE__宏的值是由编译器确定的，但受#line指令（第3.6节）更改的控制。C99的预定义标识符__func__（第2.6.1节）的目的与__LINE__相似，但它实际上是个代码块作用域变量，而不是一个宏。它提供了外层函数的名称。

__STDC__和__STDC__VERSION宏适用于编写与标准C和非标准C编译器兼容的代码。__STDC__HOSTED__宏是C99所引入的，用于区分宿主环境和独立环境。其余的C99宏提示了编译器遵循其他相关的国际标准的浮点数和宽字符工具（推荐遵循这些标准，但并不是必须）。

表3-2 预定义的宏

宏	值
__LINE__[a]	当前源程序行的行号，用十进制整数常量表示
__FILE__[a]	当前源文件的名称，用字符串常量表示
__DATE__	编译时的日历日期，用“Mmm dd yyyy”形式的字符串常量表示。Mmm是由asctime产生的
__TIME__	编译时的日历时间，用“hh:mm:ss”形式的字符串常量表示，后者是由asctime返回的
__STDC__	当且只当编译器遵循ISO标准时，它的值是十进制常量1
__STDC__VERSION__	如果编译器遵循C89修正案1，则这个宏的值是199409L。如果编译器遵循C99，则这个宏的值是199901L。在其他情况下，这个宏的值是未定义的

（续）

宏	值
__STDC__HOSTED__	（C99）如果当前是宿主系统，这个宏的值是1。如果当前是独立系统，这个宏的值是0
__STDC__IEC__559__	（C99）如果浮点实现遵循IEC 60599标准，这个宏的值为1，否则为未定义
__STDC__IEC__559__COMPLEX__	（C99）如果复数运算的实现遵循IEC 60559，则这个宏的值为1，否则这个宏的值是未定义的
__STDC__ISO10646__	（C99）定义为一个长整数常量，yyyymmL表示wchar_t值遵循进行了修正和增补的ISO 10646标准，就像特定的年和月一样。否则为未定义

a 这些宏在非ISO的编译器中很常用。

编译器一般会定义其他的宏，与环境的相关信息进行通信，例如程序被编译时所在的计算机的类型。至于哪些宏被定义则取决于编译器，尽管UNIX系统一般预定义了unix。和内置的宏不同，这些宏可以被取消定义。标准C要求编译器特定的宏名以一个前导的下划线开始，后面可以是一个大写字母或另一个下或线（unix宏并不符合这个标准）。

例子

预定义宏适用于一些类型的错误信息：

```
if (n != m)
    fprintf(stderr,"Internal error: line %d, file %s\n",
        __LINE__, __FILE__ );
```

还有一些编译器所定义的宏可以用于隔离宿主或目标特定的代码。例如，Microsoft Visual C++把__WIN32定义为1：

```
#ifdef __WIN32
    /* Code for Win32 environment */
#endif
```

__STDC和__STDC__VERSION__宏适用于编写那些需要兼容标准C和非标准C编译器的程序：

```
#ifdef __STDC__
  /* Some version of Standard C */
#if defined(__STDC_VERSION__) && __STDC_VERSION__>=199901L
    /* C99 */
#elif defined(__STDC_VERSION__) && __STDC_VERSION__>=199409L
    /* C89 and Amendment 1 */
#else
    /* C89 but not Amendment 1 */
#endif
#else /* __STDC__ not defined */
  /* Not Standard C */
#endif
```

参考：asctime工具 第20.3节；复数运算 第23章；fprintf 第15.11节；独立系统和宿主系统 第1.4节；#ifdef 预处理器命令 第3.5.3节；#if预处理器命令 第3.5.1节；取消宏的定义 第3.3.5节；wchar_t 第24.1节。

3.3.5 取消宏定义和重新定义宏

#undef命令可以取消定义一个名称为宏：

```
#undef name
```

这个命令使预处理器忘记name的所有宏定义。取消一个当前未定义宏的定义并不是错误。当一个名称被取消定义之后，就可以向它提供一个全新的定义（使用#define），而不会产生任何错误。在#undef命令内部，并不会执行宏替换。

标准C以及其他许多编译器允许对宏进行温和的重定义。也就是说，新定义与原来的定义必须逐标记相同。新定义所包含的空白字符的位置必须与原定义相同，尽管具体的空白字符可以不同。我们认为程序员不应该依赖这种温和的重定义。更好的风格是让所有的程序入口都使用同一个单独的定义点，包括宏（有些旧式的C编译器可能不允许任何类型的重定义）。

例子

在下面的定义中，NULL的重定义是允许的，但对FUNC的所有重定义都是非法的（第1个重定义包含了原定义中未曾出现的空格，第2个重定义修改了两个标记）。

```
# define NULL 0
# define FUNC(x) x+4
# define NULL /* null pointer */ 0
# define FUNC(x) x + 4
# define FUNC(y) y+4
```

例子

当程序员由于法律的原因无法知道是否存在以前的定义时，可以使用#ifndef命令测试是否存在一个现有的定义，以避免对它进行重定义：

```
#ifndef MAXTABLESIZE
#define MAXTABLESIZE 1000
#endif
```

在那些允许命令中的宏定义调用C编译器的实现中，这种用法特别有用。例如，下面这个UNIX对C的调用提供了5000作为MAXTABLESIZE宏的初始定义。随后，C程序员应该像前面一样检查这个定义：

```
cc -c -DMAXTABLESIZE=5000 prog.c
```

尽管标准C并不允许，但一些旧式的预处理器实现在处理#define和#undef时就像维护一个定义堆栈一样。当一个名称用#define进行重新定义时，它的旧定义被压入到堆栈中，然后用新定义替换这个旧定义。当一个名称用#undef取消定义时，当前的定义就被丢弃，最近的一个定义（如果有）就被恢复。

参考：#define命令 第3.3节；#ifdef和ifndef命令 第3.5.3节。

3.3.6 宏展开的优先级错误

宏的操作完全是标记的文本替换。只有在宏展开过程结束之后，才会把宏体解析为声明、表达式或语句。如果不加注意，就有可能导致令人吃惊的结果。作为规则，对出现在宏体内部的每个参数加上括号总是最安全的。如果整个宏体在语法上是一个表达式，它的两边也应该加上括号。

例子

考虑下面这个宏定义：

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

它的思路是SQUARE接受一个参数表达式，并产生一个新的表达式，计算这个参数的平方。例如，SQUARE(5)被展开为5*5。但是，表达式SQUARE(z + 1)被展开为z + 1 * z + 1，解析的结果是z + (1 * z) + 1，而不是(z + 1) * (z + 1)。下面这个SQUARE定义可以避免这个问题：

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

为了防止对诸如(short) SQUARE(z + 1)这样的表达式作出错误解释，外层的括号是必要的。

参考：类型转换表达式 第7.5.1节；表达式的优先级 第7.2.1节。

3.3.7 宏参数的副作用

宏也可能因为副作用而产生问题。由于宏的实际参数可能会被文本复制，它们的执行次数可能不止一次，因此实际参数的副作用可能会产生不止一次。反之，与宏调用看上去很相似的真正函数调用对实参表达式的求值次数正好只有一次，因此表达式的副作用也正好出现一次。为了避免这种问题，在使用宏时必须小心。

例子

考虑前一个例子的SQUARE宏以及（几乎）完成相同任务的square函数：

```
int square(int x) { return x*x; }
```

这个宏可以求浮点数的平方根，而这个函数只能求整数的平方根。另外，调用函数的运行速度要稍慢于调用宏。但是，和可能产生的副作用相比，这些差别是微不足道的。在下面的程序片段中：

```
a = 3;
b = square(a++);
```

变量b获得的值是9，而变量a的最终值是4。但是，在下面这个表面看上去相似的程序片段中：

```
a = 3;
b = SQUARE(a++);
```

变量b值可能是12，变量a的最终值可能是5，这是因为后面这个程序片段被展开为：

```
a = 3;
b = ((a++)*(a++));
```

（12和5可能是b和a的最终值，这是因为标准C编译器可能以不同的方式对表达式((a++)*(a++)进行求值。参见第7.12节。）

参考：增值操作符++ 第7.4.4节。

3.3.8 把标记转换为字符串

标准C具有一种机制，把宏的参数（在展开之后）转换为字符串常量。在此之前，在许多C预处理器中，程序员必须依赖语言的一个空子，以不同的方式实现与此相同的效果。

在标准C中，出现在宏定义内部的#标记被当做单目的“字符串化”操作符，它的后面必须

是个宏形式参数的名称。在宏展开期间，#和形式参数的名称被一个对应的包含在双引号内的实际参数所代替。在创建字符串时，实参的标记列表中的每个空白字符序列被单个空格字符所代替，所有内嵌的引号或反斜杠字符前面都加上一个反斜杠，以保留它们在字符串中的原来含义。参数起始和结束位置的空白字符被忽略，因此空的实参（即使是逗号分隔的空白字符）被展开为空字符串""。

例子

考虑TEST宏的标准C定义：

```
#define TEST(a,b) printf( #a "<" #b "=%d\n", (a)<(b) )
```

语句TEST(0, 0xFFFF)；TEST('\n', 10)；将被展开为

```
printf("0" "<" "0xFFFF" "=%d\n", (0)<(0xFFFF) );
printf("'\\n'" "<" "10" "=%d\n", ('\n')<(10) );
```

在相邻的字符串连接之后，它们成为：

```
printf("0<0xFFFF=%d\n", (0)<(0xFFFF) );
printf("'\\n'<10=%d\n", ('\n')<(10) );
```

一些非标准C编译器将会替换字符串和字符常量“ 内部 ”的宏形式参数。标准C禁止这种做法。

例子

在不遵循标准C的编译器中，可以按照下面的方式编写TEST宏：

```
#define TEST(a,b) printf( "a<b=%d\n", (a)<(b) )
```

展开TEST(0, 0xFFFF)的结果类似于字符串化的结果：

```
printf("0<0xFFFF=%d\n", (0)<(0xFFFF) );
```

展开TEST(0, 0xFFFF)时几乎肯定会丢掉那个额外的反斜杠，printf函数的输出将被预料之外的行分隔符所破坏：

```
printf("'\n'<10=%d\n", ('\n')<(10) );
```

在非ISO编译器中，空白字符的处理也可能因编译器而异，这也是避免依赖这个特征的原因之一，除非是在标准C编译器中。

3.3.9 宏展开中的标记合并

在标准C中，合并标记以形成新标记的行为是由宏定义中的合并操作符##所控制的。在宏替换列表中（在扫描更多的宏之前），##操作符两边的两个标记被合并为单个标记。##操作符的两边必须存在标记，它不能出现在替换列表的开始或结束位置。如果这些组合无法形成一个合法的标记，则结果是未定义的。

```
#define TEMP(i)  temp ## i
TEMP(1) = TEMP(2 + k) + x;
```

在预处理之后，它成为：

```
temp1 = temp2 + k + x;
```

在前面这个例子中，在展开TEMP() + x时，会出现一种奇怪的情况。这个宏定义是合法的，

但最终##的右边却没有标记（除非是+，但这并不是我们所需要的）。这个问题的解决方案是以特殊的方式处理形式参数i，使它根据##展开为一个特殊的“空”标记。因此，TEMP() + x的展开结果将是预期的temp + x。

标记连接不能用于产生统一字符名称。

和把宏参数转换为字符串（第3.3.8节）一样，在许多非标准C编译器中，程序员可以依赖C语言的一个空子来实现类似的合并功能。尽管C语言的最初定义明确地把宏体描述为标记序列而不是字符序列，然而许多C编译器在展开和重新扫描宏体时是把它们看成是字符序列。这样至少存在一个很显而易见的空子，就是在编译器处理注释并将它们全部删除时（而不是用一个空格替换它们）。程序员在编写程序时可以聪明地利用这个空子实现标记粘贴的效果。

例子

考虑下面这个例子：

```
#define INC ++
#define TAB internal_table
#define INCTAB table_of_increments
#define CONC(x,y) x/**/y
CONC(INC,TAB)
```

标准C把CONC的宏体解释为两个标记：x和y，并用一个空格分隔（注释被转换为空格）。CONC(INC, TAB)这个调用被展开为两个标记INC TAB。但是，有些非标准编译器简单地删除注释，然后重新在宏体中扫描标记，这样CONC(INC, TAB)就被展开为单个标记INCTAB。

步骤	标准C所执行的展开	非标准C可能执行的展开
1	CONC(INC,TAB)	CONC(INC,TAB)
2	INC/**/TAB	INC/**/TAB
3	INC TAB	INCTAB
4	++ internal_table	table_of_increments

参考：递值操作符++ 第7.5.8节；统一字符名称 第2.9节。

3.3.10 宏的可变参数列表

在C99中，类似函数的宏可以用一个省略号来表示它的最后一个形式参数或唯一的那个形式参数。它表示这个宏可以接受可变数量的实参。

```
#define 字(标记、符列表, ...) 标记序列opt
#define 名字 (...) 标记序列 opt
```

当这样的一个宏被调用时，至少必须提供与identifier-list中的标识符一样多的实际参数。缀尾的参数（包括用于分隔的逗号）被合并为一个单独的预处理标记序列，称为可变参数。出现在宏定义的替换列表中的标识符__VA_ARGS__被看成是个宏形参，与它对应的实参就是合并后的可变参数。也就是说，__VA_ARGS__被额外参数列表（包括逗号分隔符）所替换，只有形参列表中存在...的宏定义中才能出现__VA_ARGS__。

具有可变数量的参数的宏经常用于作为接受可变数量的参数的函数的接口，例如printf。通过使用#字符串化操作符，它们还可以用于把一个参数列表转换为单个字符串，而不必把参数放在括号中。

例子

下面这些指令创建了一个my_printf宏，它可以把参数写入到标准错误流或标准输出流。

```
#ifdef DEBUG
#define my_printf(...) fprintf(stderr, __VA_ARGS__)
#else
#define my_printf(...) printf(__VA_ARGS__)
#endif
```

它可以按照下面这种方式使用：

```
my_printf("x = %d\n", x);
```

例子

假设有下面这个定义：

```
#define make_em_a_string(...) #__VA_ARGS__
```

下面这个调用

```
make_em_a_string(a, b, c, d)
```

被展开为字符串

```
"a, b, c, d"
```

3.3.11 其他问题

有些非标准编译器并不会对宏定义和宏调用执行严格的错误检查，包括允许宏体中的不完整标记由宏调用之后所出现的文本补充完整。有些编译器所缺乏的错误检查机制并不能被合理地利用。标准C重新确定了宏体必须是形式完整的标记序列。

例子

例如，在一个非ISO编译器中，下面这个片段

```
#define FIRSTPART "This is a split
…
printf(FIRSTPART string."); /* Yuk! */
```

在预处理之后，将会在源文本中产生

```
printf("This is a split string.");
```

3.4 文件包含

#include预处理器命令对指定源文本文件的完整内容进行预处理，用这些内容替换这条#include指令。在标准C中，#include命令具有下面这3种形式：

```
#include <h字符序列>
#include " q字符序列 "
#include 预处理器标记                    (标准C)
```

h字符序列：

除了 > 和行末字符之外的所有字符序列

q字符序列：

除了 "和行末字符之外的所有字符序列

预处理器标记：

任何C标记序列，或者无法被解释为标记的不是以<或"开头非空白字符

在前两种形式的#include指令中，定界符之间的字符应该是编译器所定义的某种形式的文件名。在结尾的>或"之后只能出现空白字符。所有的C编译器都支持这两种形式的#include指令。文件名受到标准C的三字符组替换和源代码行延续的影响，但除此之外不会再对文件名中的字符进行其他处理。

在第3种形式的#include指令中，预处理器标记采用常规的宏展开，展开结果必须与前两种形式之一匹配（包括引号或尖括号）。这种形式的#include较为少见，在非标准的编译器中可能并未实现，或者按照不同的方式所实现。

例子

下面是使用第3种形式的#include指令的一个例子：

```
#if some_thing==this_thing
#   define IncludeFile "thisname.h"
#else
#   define Includefile <thatname.h>
#endif
…
#include Includefile
```

这种风格可以用于本地化的惯用形式，但注重与旧式编译器保持兼容的程序员应该用#include指令代替前面所出现的#define指令：

```
#if some_thing==this_thing
#   include "thisname.h"
#else
#   include <thatname.h>
#endif
```

众所周知，文件名的语法依赖于操作系统，但是，标准C要求所有编译器允许出现在#include指令中的文件名由字母和数字组成（以字母开头），然后是个点号和单个字母。C99允许在点号之前至少出现8个字符和数字，但C89只保证点号前至少可以出现5个字母。这种“允许”意味着这种形式的文件名必须映射到一个依赖于系统的文件。

由双引号所定界的文件名和由尖括号所定界的文件名的区别在于编译器查找文件的方式。这两种形式都在一组由编译器所定义的位置（两者可能不同）搜索文件。一般情况下，下面的形式

```
#include <文件名>
```

根据编译器所定义的规则，在一些标准位置搜索被包含的文件。这些标准位置通常包含了编译器自己所提供的头文件，例如stdio.h。下面这种形式

```
#include "文件名"
```

也在一些标准位置搜索被包含的文件，但通常是在搜索了程序员所设置的当前目录之后才在这些位置进行搜索。在C语言之外，编译器常常提供了一些标准的方式指定搜索这些文件的位置。一般的思路是"..."形式表示程序员自己编写的头文件，而<...>形式表示编译器所提供的标准头文件。

事实上，像stdio.h这样的标准头文件在标准C中也是作为特殊情况处理的。标准C要求编译器认识#include命令中由< >定界的标准库头文件名，而不管这些文件名是否指定了真正的文件名。它们可以作为特殊情况处理，C编译器简单地“知道”它们的内容。出于这个原因，标准调用它们多个标准头文件，而不是一个标准头文件。在本书中，我们用到了两种方式。

被包含的文件可以包含#include命令。这种#include命令的嵌套深度是由编译器所指定的，但标准C要求至少支持8层（C99要求至少支持15层）嵌套。包含文件的位置可能会影响被嵌套文件的搜索规则。

例子

假设我们编译一个C程序first.c，它位于系统目录/near。first.c文件包含了下面这几行：

```
// In /near/first.c
#include "/far/second.h"
```

它指定了在/far目录寻找second.h文件。second.h文件包含了下面这几行：

```
// In /far/second.h
#include "third.h"
```

它并没有指定目录。编译器将选择最初的工作目录中的/near/third.h文件，还是选择包含它的目录中的/far/third.h文件呢？有些UNIX C编译器将找到/far/third.h。C的最初描述似乎建议找到的应该是/near/third.h文件。大多数编译器让程序员指定进行搜索的目录列表，用于那些未指定目录的包含文件。

参考：字符串常量 第2.7.4节；三字符组 第2.1.4节。

3.5 条件编译

预处理器的条件编译指令允许预处理器根据一个经过计算所得出的条件传递或删除几行源文本代码。

3.5.1 #if、#else和#endif命令

下面这些预处理器命令一起使用时，允许在编译时根据#if、#else和#endif所设置的条件包含或排除部分源文本行。它们是按照下面的方式使用的：

```
#if   常量表达式
        文本行组1
#else   文本行组2
#endif
```

常量表达式受宏替换的影响，它的求值结果必须是个算术常量值。第7.1.1节讨论了对表达式的一些限制。“文本行组”可以是任何数量、任何类型的文本行，甚至是其他预处理器文本行或者根本没有任何文本行。#else命令以及它后面的文本行组可以被省略，相当于#else命令后面没有任何文本行。在文本行组中还可以包含一组或多组#if - #else - #endif命令。

像前面这样的一组命令进行处理的结果是一组文本行被传递给编译器进一步处理，而另一组文本行则被丢弃。首先，#if命令中的常量表达式被求值。如果它的值不是0，则“文本行组1”被传递给编译器进行编译，而“文本行组2”（如果存在）则被丢弃。否则，“文本行组1”就被丢弃。并且，如果存在#else命令，则“文本行组2”被传递给编译器进行编译。但是，如果不存

在#else命令，就没有任何文本组行被传递给编译器。第3.5.4节和第7.11节详细描述了可以在#if命令中使用的常量表达式。

被丢弃的文本行组不会被预处理器处理。它内部的宏替换不会被执行，预处理器命令也被忽略。唯一的例外是在被丢弃的文本行组内部，将会确认#if、#else、#ifndef、#elif、#else和#endif命令，以便对它们进行计数（这是唯一的目的）。维护条件编译命令的正确嵌套是有必要的。这意味着被丢弃的行也会被扫描并分解为标记，并且能够确认其中的字符串常量和注释，以便对它们进行正确的定界。

如果#if或#elif的常量表达式中出现了一个未定义的宏名，它就被整数常量0所替换。这意味着“ #ifdef 名称 ”这条命令和“ #if 名称 ”这条命令将具有相同的效果，只要被定义的宏名具有非零的算术常量值。我们认为在这种情况下使用#ifdef或预定义的操作符是更好的方法，但标准C也支持#if的这种用法。

参考：defined 第3.5.5节；#elif 第3.5.2节；#ifdef 第3.5.3节。

3.5.2 #elif命令

#elif命令在标准C编译器以及那些较为现代的非ISO编译器中存在。它是一种很方便的命令，因为它能够简化一些预处理器条件。这是按照下面的方式使用的：

```
#if   常量表达式1                    （或#ifdef，或#ifndef）
        文本行组1
#elif  常量表达式2
        文本行组2
…
#elif 常量表达式n
        文本行组n
#else
        最后一组文本行
#endif
```

这些命令序列的处理方式是最多只有一组文本行被传递给编译器进行编译，其他各组文本行都被丢弃。首先，#if命令中的常量表达式1被求值。如果它的值不是0，则“ 文本行组1 ”被传递给编译器进行编译，所有其他到匹配的#endif为止的各组文本行都被丢弃。如果#if命令中的常量表达式1的值不是0，则第1条#elif语句中的常量表达式2被求值。如果这个值不是0，则“ 文本行组2 ”被传递给编译器进行编译。在一般情况下，每个常量表达式i按顺序被求值，直到有一个产生了非零值。然后，预处理器就把产生这个非零值的常量表达式所在的命令后面的文本行组传递给编译器，并忽略其他命令的常量表达式，并丢弃其他所有文本行组。如果没有常量表达式i为非零值，并且存在一条#else语句，则#else命令后面的那组文本行被传递给编译器。但是，如果此时不存在#else命令，则不会有任何文本组行被传递给编译器。在#elif命令中所使用的常量表达式可以与#if命令中所使用的常量表达式相同（参见第3.5.4节和第7.11节）。

在一组被丢弃的文本行中，#elif命令就像#if、#else、#ifndef、#else和#endif一样被确认，唯一的目的是为了对它们进行计数。维护条件编译命令的正确嵌套是十分必要的。

宏替换是在#elif命令后面的一条命令中执行的，因此可以在常量表达式中使用宏调用。

例子

尽管在合适的时候使用#elif命令是非常方便的，但也可以只使用#if、#else和#endif来实现它

的功能。下面显示了一个例子。

使用#elif	不使用#elif
#if 常量表达式1 　　文本行组1 #elif 常量表达式2 　　文本行组2 #else 最后一组文本行 #endif	#if 常量表达式1 　　文本行组1 #else #if 常量表达式2 　　文本行组2 #else 　　最后一组文本行 #endif #endif

3.5.3 #ifdef和#ifndef命令

#ifdef和#ifndef命令可以用于测试一个名称是否被定义为一个预处理器宏。下面这种形式的命令行

```
#ifdef  name
```

如果name已经被定义为一个预处理器宏，它的含义相当于

```
#if  1
```

如果name没有被定义为预处理宏或者被#undef命令取消了定义，则它相当于

```
#if  0
```

#ifndef命令的含义正好相反。当name未被定义时，它的测试结果为真，否则为假。

注意，#ifdef和#ifndef只测试名称是否由#define命令所定义（或者由#undef取消定义），它们与被编译的C程序文本的声明中所出现的名称无关（有些C编译器允许使用特殊的编译器命令行参数来定义名称）。

例子

#ifndef和#ifdef可以按照几种不同的风格在C程序中使用。首先，通过一组只有一个被定义的符号实现一种预处理器时的枚举类型是常见的做法。例如，假设我们希望使用VAX、PDP11和GRAY2这组名称来指示程序被编译时所在的计算机。有些人坚持认为所有这些名称都应该被定义，其中只有1个被定义为1，其余的都定义为0：

```
#define VAX 0
#define PDP11 0
#define CRAY2 1
```

然后，可以按照下面的方式选择需要编译的依赖于计算机的源代码：

```
#if VAX
    VAX-dependent code
#endif
#if PDP11
    PDP11-dependent code
#endif
#if CRAY2
    CRAY2-dependent code
#endif
```

但是，习惯的做法是只定义一个符号：

```
#define CRAY2 1
    /* None of the other symbols is defined. */
```

然后用条件编译指令测试这些符号是否被定义：

```
#ifdef VAX
    VAX-dependent code
#endif
#ifdef PDP11
    PDP11-dependent code
#endif
#ifdef CRAY2
    CRAY2-dependent code
#endif
```

例子

#ifdef和#ifndef命令的另一个用途是提供宏的默认定义。例如，一个库文件可能只有在一个名称并没有被定义的情况下才会向它提供定义：

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
…
static int internal_table[TABLE_SIZE];
```

程序可以简单地包含这个文件：

```
#include <table.h>
```

在这种情况下，TABLE_SIZE的定义将会是100，不管是在库文件中还是在#include之后。或者，程序可以提供一个显式的定义：

```
#define TABLE_SIZE 500
#include <table.h>
```

在这种情况下，TABLE_SIZE的定义将始终是500。

用“ #if name ”代替“ #ifdef name ”或“ #ifndef name ”来测试一个名称是否被定义是一种常见的C的编程错误。这种不正确的形式经常能够实现目的，因为预处理器用常量0来替换#if表达式中并没有被定义为宏的任何名称。因此，如果name没有被定义，这3种形式都是一样的。但是，如果name被定义为0这个值，即使这个名称已经被定义，“ #if name ”的结果仍然是假。类似，如果用一个非法的表达式来定义name，则“ #if name ”将导致错误。

参考：#define 第3.3节；defined操作符 第3.5.5节；#include 第3.4节；预处理器词法约定 第3.2节；#undef 第3.3节。

3.5.4 条件命令中的常量表达式

第7.11.1节描述了可以在#if和#elif命令中使用的表达式。它们包括整数常量以及所有的整数算术、关系、位和逻辑操作符。

C99强迫所有的预处理器根据目标计算机上的最大整数类型来执行算术运算，它是stdint.h所定义的intmax_t或uintmax_t。以前，标准C并不要求编译器实现目标计算机的运算属性。

参考：intmax_t 第21.5节；uintmax_t 第21.5节。

3.5.5 defined操作符

defined操作符可以在#if和#elif表达式中使用，而不能用于别处。下面这两种形式的表达式：

```
defined name
defined( name )
```

如果name在预处理器中已经被定义，上面的表达式的值为1，否则为0。

例子

defined命令允许程序员用下面的指令

```
#if defined(VAX)
```

代替

```
#ifdef VAX
```

defined操作符用起来可能更为方便，因为它可以创建像下面这样复杂的表达式：

```
#if defined(VAX) && !defined(UNIX) && debugging
…
```

3.6 显式的行号

预处理器命令#line向C编译器提示了源程序是由其他工具所生成的，并提示源程序中的位置与用户最初所编写的文件（C源程序是通过这个文件形成的）中的行的对应关系。#line命令可以使用两种形式之一。下面这种形式

```
# line n " filename "
```

表示源代码的下一行来自用户所编写的称为filename的文件的第n行。n必须是个十进制数字序列。下面这种形式

```
# line n
```

表示源代码的下一行来自#line命令上次所提到的用户所编写的文件的第n行。最后，如果#line命令并不与前面的任一形式匹配，则它被解释为

```
# line   预处理器标记
```

在参数标记序列中将会执行宏替换，其结果必须与前面两种#line格式之一匹配。

#line命令所提供的信息用于设置预定义宏__LINE__和__FILE__的值。如果用于其他情况，则它的行为是未指定的，并且编译器可以忽略它。一般情况下，这种信息也可以出现在诊断信息中。有些生成C源代码文本作为输出的工具将使用#line，使错误信息可以与这个工具的输入文件而不是实际的C源文件相关。

有些C编译器允许预处理器独立于编译器的其他部分使用。事实上，有时候预处理器是个独立的程序，它执行之后产生一个中间文件，供真正的编译器继续进行处理。在这种情况下，预处理器可能会在中间文件中产生新的#line命令。然后，编译器可以认识这些行号，即使它们可能无法被其他的预处理器命令所认识。预处理器是否生成#line命令取决于C的实现。类似，预处理器是否在输入中传递、修改或消除#line命令也是取决于C的实现。

旧版本的C允许简单地使用“ # ”作为#line命令的同义词，允许采用下面这种形式：

```
# n filename
```

这种语法被认为是过时的，标准C不再允许这种用法。但是，由于兼容性的缘故，许多编译器继续支持这种做法。

参考：__FILE__ 第3.3.4节；__LINE__ 第3.3.4节。

3.7 pragma指令

#pragma命令是标准C新增的。下面的命令名后面可以出现任何标记序列：

```
# pragma 预处理器标记
```

#pragma指令可以由C的实现所使用，以添加新的预处理器功能或者向编译器提供因实现而异的信息。#pragma命令后面所出现的信息并无任何限制，编译器（包括预处理器）应该忽略它们所不理解的信息。除了在C99标准pragma（3.7.1节）的情况下，#pragma后的标记是否受宏扩展的影响取决于实现。

显然，不同的编译器可能会以不同的方式理解相同的信息。因此，以条件编译的形式，根据具体使用的编译器来选择需要使用的#pragma指令是明智的做法。

例子

下面的代码检查在发布#pragma命令之前是否使用了正确的编译器（tcc）、计算机以及遵循标准的C实现：

```
#if defined(_TCC) && defined(__STDC__) && defined(vax)
#pragma builtin(abs),inline(myfunc)
#endif
```

参考：defined 第3.5.3节；内存模型 第6.1.5节；#if 第3.5.1节。

3.7.1 标准pragma命令

C99引入了一些pragma指令，表示一些特殊的含义。为了区分它们，所有的标准pragma指令都必须加上前缀STDC（在任何具体实现宏扩展之前）。也就是说，下面这条指令

```
#pragma FENV_ACCESS ON
```

是一条由编译器所定义的pragma指令，但下面的指令

```
#pragma STDC FENV_ACCESS ON
```

指定了C99的pragma指令FENV_ACCESS。如果一个标准的pragma名称前面没有前缀STDC，编译器将会发出一条友好的警告信息，因为这很可能是个常见的错误。

C99所定义的仅有的标准pragma指令是FP_CONTRACT、FENV_ACCESS和CX_LIMITED_RANGE。它们都接受一个“打开-开闭”的开关作为参数。

打开-关闭开关：

```
ON
OFF
DEFAULT
```

DEFAULT参数把pragma指令设置为它的初始默认值（打开或关闭）。每个标准pragma都指

定了一个默认值（有时候，它的默认值是由编译器所定义的）。

参考：CX_LIMITED_RANGE 第23.2节；FENV_ACCESS 第22.2节。

3.7.2 标准pragma指令的位置

标准pragma指令必须遵循一定的位置规则，这样pragma指令更易于处理，并且允许pragma的嵌套。标准pragma指令可以出现在两个位置：在翻译单元的顶层，在任何外部声明之前；或者在复合语句的起始位置，在所有的显式声明和语句之前。

当位于顶层时，pragma指令的效果一直维持到当前翻译单元的末尾或者直到遇见了同一条pragma指令的另一个实例。后者可能是顶层的另一条pragma指令（此时它会覆盖第1条pragma指令），也可能是一条复合语句中的pragma指令。

当pragma指令位于复合语句的开始位置时，它将一直有效，直到该复合语句的结束（从词法上），或者直到在该复合语句中遇到同一条pragma指令的另一个实例。后者可能位于同一条复合语句的起始位置（此时它会覆盖第1条pragma指令），或者位于另一条内层复合语句中。在包含一条标准pragma指令的复合语句结束时，这个pragma将会恢复到该复合语句之前它的状态。也就是说，嵌套的标准pragma将遵循常规的变量作用域规则，区别在于它们可以在同一个作用域层次上被多次指定。

参考：作用域 第4.2.1节。

3.7.3 _Pragma操作符

C99增加了_Pragma操作符，以便更灵活地利用pragma工具。在宏展开之后，下面这种形式的操作符表达式

```
_Pragma ("字符串常量")
```

的处理方式是把字符串常量的内容（在删除两边的双引号，并把字符串常量内部的\"替换为"，把\\替换为\之后）看成是#pragma指令中所出现的预处理器标记。例如，下面这个表达式

```
_Pragma("STDC FENV_ACCESS ON")
```

将被看成是在这个位置上出现了下面这条pragma指令。

```
#pragma STDC FENV_ACCESS ON
```

#pragma指令本身必须出现在同一行中，而且它的预处理器标记并不会执行宏展开，但_Pragma可以被其他表达式所包围，并且可以通过宏展开而产生。

3.8 错 误 指 令

#error指令是标准C新增的。它的后面可以出现任何标记序列：

```
#error  预处理器标记
```

#error指令产生一条编译时错误信息，其中包含了参数中的标记，而后者又受宏展开的影响。例子

#error指令特别适用于检测程序员对预处理期间的限制的不一致行为和违规行为：

```
#if defined(A_THING) && defined(NOT_A_THING)
#error Inconsistent things!
#endif
```

```
#include "sizes.h"   /* defines SIZE */
…
#if (SIZE % 256) != 0
#error "SIZE must be a multiple of 256!"
#endif
```

在第1个#error例子中，我们并没有使用字符串常量。在第2个例子中，我们使用了字符串常量，这是因为我们并不想让SIZE标记在输出信息中展开。

参考：defined 第3.5.3节；#if 第3.5.1节。

3.9 C++兼容性

C++使用C89预处理器，因此C和C++在这方面的差别极微。

预定义宏

C++编译器预定义了__cplusplus宏，可以在源文件中使用，并且在C和C++环境中都适用。这个预定义宏并不遵循标准C的拼写约定，而是与原有的C++编译器兼容。在标准C++中，它的值是个版本号，例如199711L。

至于C++环境（在C++的当前定义中）是否定义了__STDC__，则是由编译器定义的。标准C和标准C++之间存在足够的差别，因此并不清楚是否应该定义__STDC__。

表3-2所列出的只有C99才有的宏在C++中均不存在。

例子

为了与传统C、标准C和C++兼容，应该按照下面这种方式对环境进行测试：

```
#ifdef __cplusplus
   /* It's a C++ compilation */
#else
#ifdef __STDC__
   /* It's a Standard C compilation */
#else
   /* It's a non-Standard C compilation */
#endif
#endif
```

如果读者知道自己所使用的C编译器遵循标准C，则可以采用简化的方式：

```
#if defined(__cplusplus)
   /* It's a C++ compilation */
#else
   /* It's a Standard C compilation */
#endif
```

参考：__STDC__ 第3.3.4节；__STDC__VERSION__ 第3.3.4节。

3.10 练　习

1. 下面哪个标准C的宏定义（很可能）是错误的？为什么？哪些定义在传统C中可能导致问题？

(a) `#define ident (x) x`　　(c) `#define PLUS +`

(b) `# define FIVE = 5;`　　(d) `#define void int`

2. 下面是一些宏定义和宏调用。每个宏调用分别会被标准C和传统C怎样展开呢？

	定义	调用
(a)	`#define sum(a,b) a+b`	`sum(b,a)`
(b)	`#define paste(x,y) x/**/y`	`paste(x,4)`
(c)	`#define str(x) # x`	`str(a book)`
(d)	`#define free(x )x ? free(x) : NULL`	`free(p)`

3. 下面显示了两个头文件和一个C程序文件。如果C预处理器对这个程序文件进行处理，会产生什么结果？

```
/* File blue.h */
int blue = 0;
#include "red.h"
```

```
/* File red.h */
#ifndef __red__
#define __red__
#include "blue.h"
int red = 0;
#endif
```

```
/* File test.c */
#include "blue.h"
#include "red.h"
```

4. 有人展示了下面这个宏定义，它的用意是对它的数值参数进行加倍。这个宏有什么错误？改写这个宏，使它能够执行正确的操作。

```
#define DBL(a)    a+a
```

5. 在下面这个标准C程序片段中，M(M)(A,B)展开后的结果是什么？

```
#define M(x)      M ## x
#define MM(M,y)   M = # y
M(M)(A,B)
```

6. 编写一系列的预处理器指令，如果SIZE宏没有被定义或者虽然被定义但它的值不在1-10的范围之内，它们将导致标准C程序无法通过编译。
7. 提供一个字符序列的例子，它可以作为一个单独的标记提供给预处理器，但它对于编译器却是不正确的。
8. 下面这个程序片段有什么问题？

```
if (x != 0)
   y = z/x;
else
   # error "Attempt to divide by zero, line " __LINE__
```