Macro是一段有名字的代码。当使用这个名字时，名字就会被宏的内容替代。有两种类型的宏，区别是使用的时候更像那种形式。Object-like宏，在使用时，就像一个object。function-like宏使用时就像函数调用。

可以使用任何合法的标示符来定义一个宏，甚至比如C关键字。预处理器不识别任何关键字。当你想隐藏一些关键字的时候，你可以使用这个技巧来屏蔽。【注：宏替换比关键字识别的阶段要早】预处理器的defined关键字不能定义为宏，当编译C++时，C++命名的操作不能定义为宏（3.7.4 C++ named operators）

3.1 Object-like Macros

Object-like宏是一个简单的标示符，在使用时会被替换。被称之为object-like，是因为使用的时候就像一个数据Object一样的使用。最常使用的场景是给数字常量起个名字。

用#define指令来定义一个宏。#define之后跟一个宏的名字，接着是宏代表的代码段，也称之为expansion list或者replacement list.比如

```
#define BUFFER_SIZE 1024
```

在#define之后，如下的语句

```
foo = (char*) malloc (BUFFER_SIZE);
```

C preprocessor会进行识别和扩展BUFFER_SIZE宏。C compiler会识别成如下的代码：

```
foo = (char*) malloc(1024);
```

通常，宏的名字是大写。这样当第一眼看到的时候，就知道这个是宏。

宏体(macro body)在#define行的结尾结束。如果有多行的macro body，就需要使用反斜杠(backslash-newline)。当宏扩展时，会将这些行扩展到一行。

```
#define MUNER 1, \
                2,\
                3
int x[] = { NUMERS };
    -> int x[] = {1, 2, 3};
```

这样写的问题是行号可能会有异常。【译注：参看3.10.7 newlines in Arguments】

宏体中的内容没有限制，宏体会被解析为合法的预处理tokens。括号也不一定要成对出现，宏体也可以有非法的字符（编译器在使用这个错误的宏的位置报错）。预处理器顺序的扫描程序。宏在定义的位置生效。

```
foo = X;
#define X 4
bar = X
```

会生成如下的代码：

```
foo = X;   【译注：这个时候还没有定义，不会发生宏扩展】
bar = 4;
```

当预处理器扩展宏名字，宏的扩展会在宏引用的地方进行。

```
#define TABLESIZE BUFSIZE
#define BUFSIZE 1024
TABLESIZE
    -> BUFSIZE
    ->1024
```

TABLESIZE首先扩展为BUFSIZE，然后最终扩展为1024。BUFSIZE在TABLESIZE定义之前没有定义。TABLESIZE会扩展为BUFSIZE，不会检查是否有更多的扩展。只有当使用TABLESIZE的时候，宏才会扩展更多的名字。如果在某个时间点修改了BUFSIZE的值，那么TABLESIZE的值也会随之改变。

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

【译注：此时使用TABLESIZE值是1020】

```
#undef  BUFSIZE
#define BUFSIZE 37
```

【译注：此时使用TABLESIZE值是37】

如果一个宏的扩展包含自己的名字，不管直接或者间接的，则不会发生再次的扩展，以防止无限递归的扩展。查看Section 3.10.5 [self-referential Macros]来了解更精确的细节。

3.2 Function-like Macros

也可以定义一个类似函数的宏，这类的宏称之为function-like宏。当定义function-like宏，需要使用#define，同时需要紧跟着一对括号。

```
#define lang_init() c_init()
lang_init()
    ->c_init()
```

funcation-like macro只有在宏名字后边有括号的时候才会扩展。如果仅仅写宏的名字，则不会进行扩展。在函数和宏同名的时候，遵从这个原则。

```
extern void foo(void);
#define foo() /*optimized inline version*/
    ....
foo();          【此时是宏扩展】
funcptr = foo;  【此时是函数】
```

如果在宏名字和括号之间加了空格，这时候不是定义function-like宏，而是object-like宏，在扩展时会是括号和之后的宏体

```
#define lang_init () c_init()
lang_init()
    ->() c_init()()
```

3.3 Macro Arguments

Function-like宏可以有参数，就像真正的函数一样。当定义一个带参数的宏，要把参数写到括号中间，参数必须是合法的C标示符，通过逗号和可选的空白分割【译注：空格，tab】。

调用有参数的宏时，在宏的名字的括号中，紧跟着真实的参数，通过逗号分隔。调用宏时不用限制只能一行，可以跨行写。【译注：定义宏时，扩行写需要用反斜杠续行，宏调用时，可以跨行写，而不需要用反斜杠续航】参数的数量必须和定义的数量一样。当宏扩展时，宏体中使用的参数会用对应的真实的参数代替（参数数量可以不用一样）。

下边是一个求较小值的例子：

```
#define min(x, y) ((x) < (y) ? (x) : (y) )
x = min(a, b)
     -> x = ((a) < (b) ? (a) : (b))
```

在参数的开头和结尾的空白会被丢掉，所有参数中间的空白会被替换为一个空格。参数的括号要成对，括号中的逗号不会结束参数列表。但是没有要求[],{}成对，逗号此时还是用来分隔参数。比如：

macro(array[x = y, x + 1])

传递了两个参数给宏 array[x = y 和 x + 1。【此时，逗号分隔了两个参数，分别是array[x=y和 x + 1，而不是一个参数】。如果要将arry[x = y, x + 1]作为一个参数，就需要写成array[(x = y, x + 10)]，和C语言一样。

在宏参数替换到宏体以前，宏参数会完成扩展。在宏参数带入之后，扩展后的代码会再次被检查，如果还有宏，会继续扩展。这个规则看上去比较奇怪，但是是经过仔细的设计，这样你就不用担心任何函数是一个宏调用了。查看3.10.6[argument prescan]以了解更多细节。

比如min(min(a, b), c)首先会被扩展为

```
min(((a) < (b)?(a):(b)),c)    【原手册有误，c不会有括号】
((((a) < (b)?(a):(b))) < (c)
?(((a) < (b)?(a):(b)))
:(c))
```

参数可以传空的，预处理器不会报错（但是会扩展出错误的代码）。不能够参数全部都是空的，如果一个宏有两个参数。则至少要有一个逗号。空白不是预处理器的token，因此如果一个宏比如foo()有一个参数，foo()和foo( )都会是空参数。之前的GNU预处理器的实现和文档在这一点上都有错，需要一个参数的funcion-like macro必须有 一个参数，如果传递空白就是空参数。

在双引号里的宏参数不会被扩展。

```
#define foo(x) x,"x"
foo(bar)       -> bar, "x"
```

3.4 Stringigication

有时候需要把一个宏参数转换为一个字符串常量。参数在字符串里不会被扩展，这时候可以用预处理符号#来达到目的。当一个宏参数以#开始时，预处理器按照字面字串产生字符串。和其他的参数替代不同，#开始的参数不会首先进行宏扩展。这种行为叫做Stringification.

没有办法可以同时做到既stringify一个参数，又有字符串包含这个宏参数【比如 hello#finlworld，想在helloworld之间加上字符串】。要想达到这种目的，可以用连续的字符串和stringification拼接达到目的。预处理器会将stringifed参数替换为字符串。C编译器再将字符串拼接成一个长的字符串。

```
#define WARN_IF(EXP) \
        do { if (EXP) \
            fprintf(stderr, "Warning:" #EXP "\n"\);}\
         while(0)

WARN_IF(x == 0)
```

EXP会替代在if语句中替换，然后在#EXP产生"x == 0"的字符串。如果x是个宏，则只在if语句中expand，而不会在#EXP时扩展。do...while(0)可以保证WARN_IF(arg)的正确，否则可能会出错，查看3.10.3 [swallowing the semicolon]。stringification 不只是简单的在参数外加上双引号。预处理器会按照backslash-escapes的方式转义包含在字符串中的双引号，和字符串中的backslash。这样当stringify p="foo\n";，预处理器会扩展出"p = \"foo\\n\";"【译注：对其中的"和\都进行了反向转义，保证是合法的C语句】。

所有的stringification字段前后的空白都会被忽略。在stringification中的任意的空白都会被转为一个空格。没有办法来将一个宏参数转换为character constant。【译注：比如无法stringification得到'd'】

如果你要想将宏参数也是宏的情况，转换为字符串，需要使用二级宏。

```
#define STR(str) #str
#define XSTR(str) STR(str)
```

```
#define foo 4
STR(foo)
    -> "foo"
XSTR(foo)
    -> 4
```

原因是stringification的参数不会进行扩展。通过二级宏，先扩展参数，在扩展宏体，即可以达到扩展参数的目的。

3.5 Concatenation

【译注：以下是Kernel中的一个例子】

```
#define MODULE_FUNCS_DEFINE(dev_name)                                   \
static void dev_name##_init(void){                                      \
        common_block_set(dev_name##_BLOCK_NAME, NORMAL);                \
}
MODULE_FUNCS_DEFINE(UART1)

#define XX_UART_PLAT_DATA(dev_name, flag)                               \
        {                                                               \
                .dma_filter = dev_name##_dma_filter,                    \
                .dma_rx_param = &XX_DMA_PARAM_NAME(dev_name##_RX),      \
                .dma_tx_param = &XX_DMA_PARAM_NAME(dev_name##_TX),      \
                .irq_flags = flag,                                      \
                .init = dev_name##_init,                                \
                .exit = dev_name##_exit,                                \
        }

static struct amba_pl011_data uart_plat_data[] = {
        XX_UART_PLAT_DATA(UART0, 0),
        XX_UART_PLAT_DATA(UART1, 0),
        XX_UART_PLAT_DATA(UART2, 0),
        XX_UART_PLAT_DATA(UART3, 0),
        XX_UART_PLAT_DATA(UART4, 0),
};
```

【例子完】

宏扩展时，将两个token合并成一个token是一个比较有用的技巧。这种方法称之为token pasting或者token concatenation。##可以达到这个目的。当一个宏扩展时，##两边的token会合并为一个token，新产生的token会代替##以及原先的两个字符串。【也可以连续的##，比如test##test2##test3】通常要合并的token都可以是标示符，或者一个是标示符，一个数preprocess number。当pasted，生成一个长的标示符。 也可以拼接两个数字，或者一个数字一个名字，同时也可以产生一些运算符，比如+=。

如果两个token不能组合成一个合法token，那么这两个token就不能past。比如说不能把x和+进行past。如果这么做，预处理会发出一个warning，并且保留这两个taoken。是否在两个token之间增加空格是未定义的。通常用这种方法发现复杂的宏中不必要的##。如果你遇到这样的warning，可以简单的移除##（或者做相应的修改）。

##来连接的token可以都来自宏体，你也可以在一开始出现的地方就写成一个token。Past最大的作用是一个或者

两个token都是来自宏参数。如果任意一个##连接的token是参数名字，那么会在带入##前进行参数替换【译注：注意不是宏扩展，这里的意思是paset的token是调用时候的参数，而不是定义时候的参数】。和stringify一样，并不会进行宏扩展。如果参数为空，则##没有效果。

需要考虑的是，在宏扩展以前，注释会被预处理器转换为空格。因此上你不能通过/和*来past一个注释，如果这样，注释就会被保留下来。

3.6 Variadic Macros

宏可以声明为接受可变参数，就和函数一样。声明的语法：

```
#define eprintf(...) fprintf(stderr, __VA_ARGS__)
```

这种宏叫variadic。当宏调用时，所有在最后一个有名字的参数(这个例子没有)后的tokens都是variable arguments，这些tokens会替代__VA_ARGS__。

```
eprintf("%s:%d:", input_file, lineno)
    -> fprintf(stderr, "%s:%d:", input_file, lineno)
```

参数会在宏体扩展前，完全替换成参数。可以用#和##来对可变的参数进行stringify或者paste.比如#__VA_ARGS__，或者##__VA_ARGS__。

如果宏比较复杂，可以增加描述性的名字，这个是CPP的一个扩展。先起一个名字，然后紧跟着...。

```
#define eprintf(args...) fprintf(stderr, args)
```

这种情况下，就不能再使用__VA_ARGS__了。也可以同时使用有名字的参数和可变参数。

```
#define eprintf(format, ... ) fprintf(stderr, format, __VA_ARGS)
```

这种宏就看上去更有描述性，但是有点不方便。这种类型，在format参数之后，至少要再指定一个参数。在标准C中，你不能忽略逗号分开的有名字的参数（例子里没有这种情况）如果你不填写可变的参数，会有一个错误。标准C会扩展出一个逗号。

```
eprintf("success!\n")
    ->fprintf(stderr, "success!\n",); 【手册有误】
```

GNU CPP有两种方法方法来处理这种情况。

第一种方法，你可以完全不写可变参数：

```
 erpintf("success!\n")
    ->fprintf(stderr, "sucess!\n", );【！！需要在新版本上验证】
```

第二种方法，在逗号和可变参数之间用##。

```
#define eprintf(format, ...) fprintf(stderr, format, ##__VA_ARGS__)
```

当逗号后的可变参数忽略时，##前的逗号会被删除。但是当你整个都不传参数，或者##前不是个逗号时，就会不后起作用。

当宏参数只有一个可变参数，而没有其他的命名的参数时，就没有必要区分是没有传递参数还是没有参数。所以，C99规定，逗号必须保留，GNU CPP扩展允许没有逗号。因此上当指定特定的C标准是，CPP会保留逗号，其他情况会丢掉逗号。【！！需要在新版本上验证】

```
#include <stdio.h>

#define eprintf(format,...) fprintf(stdout, format, __VA_ARGS__)
#define Oprintf(format,...) fprintf(stdout, format, ##__VA_ARGS__)

int main(){
```

```
        eprintf("eprintf %d\n", __LINE__);
        Oprintf("Oprintf \n");

        return 0;
}

    ----------------------
fprintf(stdout, "eprintf %d\n", 10);
fprintf(stdout, "eprintf \n");
```

可变参数是C99出现的新功能。GNU CPP很早以前就支持了，不过只支持命名的可变参数（支持arg...而不是..和__VAR_ARGS__).如果考虑CNU上的移植，可以只用arg...，如果考虑C99的移植性，则可以只用__AV_ARGS__。

之前的CPP实现逗号移除更加随意，在这个版本上，GUN CPP进行了修改，以减少和C99的差异。为何和之前的GCC兼容，##前必须是逗号，逗号和逗号前的token之间要有空格。

```
#define eprintf(format, args...) fprintf(stderr, format , ##args)
```

查看11.4了解更多。

3.7 Predefined Macros

预定义宏有三种，标准的，通用的和系统特定的。

C++中有第四种，命名操作符，作用和预定义的宏一样，但是不能undefine。

3.7.1 Standard Predefined Macros

标准预定义宏是语言的标准定义的，因此所有实现了标准的编译器都会提供。有些老的编译器可能只实现了部分。标准预处理宏以两个下划线开始。

__FILE__ 这个宏会扩展为当前的输入文件，返回C字符串常量。该值会包含路径，路径从预处理器打开的文件开始计算。比如可能会扩展为/usr/local/include/myheader.h。

__LINE__这个宏会扩展为当前的输入行号，返回一个十进制的整数。虽然我们称这个宏是一个预定义的宏，比较奇怪的是，这个宏的"定义"也会逐行改变。

__FILE__和__LINE__在打印错误消息时，非常有用。比如：

```
fprintf(stderr, "Inernal error: "
          "negative string length"
          "%d at %s, line %d.",
           length, __FILE__, __LINE__);
```

#include语句会改变__FILE__和__LINE__的值，在#include语句时，__FILE__和__LINE__会改变为#include中对应的值。在#include文件结束后，他们的值又会恢复为原先的值。不过__LINE__会加一，因为#include本身也是一行。

#line会改变__LINE__和__FILE__的值。

C99提供了__func__， GCC很早之前提供了__FUNCTION__。这两个宏都会以字符串形式返回当前的函数。这两个都不是宏，预处理器不知道当前函数的名字。将这个宏和__FILE__, __LINE__结合起来，比较常用。

__DATE__, 返回一个字符串，标示编译时候的日期，字符串格式类似"Feb 12 1996". 如果日期小于10，则会在左边扩展一个空格。如果GCC不能判断当前的日期，就会产生一个warning, __DATE__会扩展为"??? ?? ????"。

__TIME__, 返回一个字符串，标示编译时的时间，字符串格式类似"23:59:01"。如果GCC不能判断当前的时间，

就会产生一个warning, __TIME__会扩展为"??:??:??"。

__STDC__,在正常模式下， 这个宏会扩展为常量1，标示当前编译器遵从ISO Standard C。如果GNU CPP使用GCC以外的编译器，该值可以不为真。预处理器总是遵从标准，除非使用-traditional-cpp。如果使用-traditional-cpp，那么这宏不会定义。

如果用其他的hosts，系统编译器使用另外的规则，__STDC__通常是0，如果用户指定严格遵从C标准，这个值就会是1。CPP按照当前host的配置，但是__STDC__总是1。这个有时候会引起问题，比如一些版本的Solaris提供的X windows的头文件，__STDC__要不是未定义，要么是1.

__STDC__VERSION__，扩展为标准C的版本，一个长整数常量，格式是yyyymmL，yyyy和mm是标准的年和月。标示当前编译器和那个标准兼容。和__STDC__一样，这个值大多数编译器都不需要精确，除非GNU CPP和GCC一起使用。

199409L表示1989 C标准，该标准在1994年修改过，这是当前的默认值。199901L标示1999版本的C标准。对1999标准的支持现在还没有完成。

如果定义了-traditional-cpp，那么这个宏不会定义，当编译C++或者Objective-C的时候，这个值也是未定义的。

__STDC_HOSTED__，如果编译器的目标是hosted environment（当前环境），这个值会扩展为1。一个hosted evironment有standard C完全的库和环境。

__cplusplus，在C++编译器使用时，这个宏会定义（defined）。可以使用__cplusplus来测试一个头文件是通过C还是C++编译的。这个宏和__STDC_VERSION__类似，会扩展为一个版本数字。完全按照1998 C++标准的话，这个宏会扩展为199711L。GUN C++还没有完全实现，所以这个值现在是1.希望近期我们可以完成标准C++的实现。

__OBJC__,当使用objective-C使用时，这个宏会被扩展为1。

__ASSEMBLER__，当处理汇编语言的时候，这个值是1.

3.7.2 通用预定义宏

通用预定义宏是GNU C的扩展，这些宏和你使用的机器和操作系统无关，只要GNU C或者GNU Fortan就可以。名字以两个下划线开始。

__COUNTER__,扩展为从0开始计数的整数。和##一起，可以扩展出唯一的标示符。使用时候要注意，预处理器会先包含已经编译的头文件，然后再扩展__COUNTER__。【？？】

__GFORTRAN__ GUN Fortran编译器定义了这个。

__GNUC__

__GNUC_MINOR__

__GNUC_PATCHLEVEL__

在所有使用C预处理器的GNU编译器中都定义了这个宏。这三个宏会被扩展为整数，他们的值分别是主版本号，次版本号，以及patch level。比如说，GCC 3.2.1会定义为__GNUC__ = 3， __GNUC_MINOR__ = 2, __GNUC_PATCHLEVEL__ = 1。当只调用GCC 预处理器的时候，这三个宏也会定义。

__GNUC_PATCHLEVEL__是GCC 3.0中增加的，这个宏在开发时候广泛的用于保留版本快照。如果只是需要知道当前程序是不是通过GCC编译，可以简单的来判断__GNUC__是否定义。如果写的代码依赖于某个特定的版本，那么需要特别小心。

__GNUG__，GNU C++定义了这个宏，这个宏和(__GNUC__&__cplusplus)的作用一致。

__STRICT_ANSI__，当定义了-ansi或者-std选项是，这个宏会被定义为1。这个宏的主要作用是让GNU libc的头文件是1989 C 的一个最小集。

__BASE_FILE__，返回C字符串常量，扩展为用于编译的主文件。这个文件名就是在编译时命令行上指定的文件名。

__INCLUDE_LEVEL__，反馈一个整数，标示当前嵌套包含文件的深度。每一级#include会增加1，退出include文件会减1.从0开始计数，主文件的值在命令行上指定。

__ELF__，如果使用ELF文件格式，这个宏会定义。

__VERSION__，返回一个字符串常量，描述了当前编译器的版本。不能依赖于该宏返回的格式。【可能是4.4.3这

样的格式，如果需要某一位的值，需要使用__GNUC__, __GNUC_MINOR__】

__OPTIMIZE__

__OPTIMIZE_SIZE__

__NO_INLINE__

这三个宏描述了编译模式。如果定义，值是1。__OPTIMIZE__会在所有的优化编译时定义，__OPTIMIZE_SIZE__在编译优化大小时定义【指定-O参数会定义该宏】。__NO_INLINE__是，如果没有函数inline，则会定义该宏(当没有进行优化，或者-fno-inline屏蔽)。

这些宏可以让一些GNU头文件提供对系统库函数优化后的定义，比如使用宏或者inline函数。这些宏要小心使用，在不确定他们用途的情况下，不要使用在其他地方。

__GNUC_GNU_INLINE__

如果一个定义为inline的函数作为GCC传统的gnu90模式处理，那么这个宏会被定义。目标文件会包含所有声明为inline但是不声明为static或者extern的函数的外部定义。不会包含任意声明为extern inline的函数定义。【？？】

__GNUC_STDC_INLINE

如果inline按照ISO C99标准处理inline。目标文件会包含所有声明为extern inline函数的外部定义。不会包含任意声明不包含extern的inline的函数定义。【？？】

如果这个宏被定义，GCC就支持以gnu_inline函数属性来使用gnu90的行为模式。对这个属性的支持和__GNUC_GNU_INLINE__从GCC 4.1.3中开始支持。如果这两个宏都没有被定义，那就是使用了一个较老的GCC版本，inline会按照gnu90的模式来进行， gnu_inline不会被识别。

__CHAR_UNSIGNED__

当且仅当当目标机器定义char是unsigned的时候，这个宏会被定义。这个宏可以保证limits.h正确的工作。你不能直接调用这个宏，应该使用limits.h中的定义。

_WCHAR_UNSIGNED__

和__CHAR_UNSIGNED__一样，当且仅当wchar_t定义为unsigned，并且front-end是C++模式的时候。

__REGISTER_PREFIX__

扩展为一个单独的token（不是一个字符串常量），这个token添加在目标机器所用的汇编语言的CPU寄存器的名字前边。多目标机环境的汇编语言编程时有用。比如，在m86k-aout环境里，这个宏扩展为空，但是在m86k-conff环境中，这个宏会扩展为%。

__USER_LABEL_PREFIX__

扩展为一个单独的token，这个token添加在汇编语言的用户标签前边（用户标签，user label：symbols visible to C code)。比如，在m86k-aout环境里，这个宏扩展为_，但是在m86k-conff环境中，这个宏扩展为空。

这个宏在使用时，即便有-f(no-)underscores选项，也是能正确的扩展，但是在目标机器有调整这个宏的选项在使用时，这个宏不生效（比如OSF/rose '-mno-underscores'选项）。

__SIZE_TYPE__

__PTRDIFF_TYPE__

__WCHAR_TYPE__

__WINT_TYPE__

__INTMAX_TYPE__

__UINTMAX_TYPE__

__SIG_ATOMIX_TYPE__

__INT8_TYPE__

__INT16_TYPE__

__INT32_TYPE__

__INT64_TYPE__

__UINT8_TYPE__

__UINT16_TYPE__

__UINT32_TYPE__

__UINT64_TYPE__

__INT_LEAST8_TYPE__

__INT_LEAST16_TYPE__

__INT_LEAST32_TYPE__

__INT_LEAST64_TYPE__

__UINT_LEAST8_TYPE__

__UINT_LEAST16_TYPE__

__UINT_LEAST32_TYPE__

__UINT_LEAST64_TYPE__

__INT_FAST8_TYPE__

__INT_FAST16_TYPE__

__INT_FAST32_TYPE__

__INT_FAST64_TYPE__

__UINT_FAST8_TYPE__

__UINT_FAST16_TYPE__

__UINT_FAST32_TYPE__

__UINT_FAST64_TYPE__

__INTPTR_TYPE__

__UINTPTR_TYPE__

这些宏用于正确的定义size_t, ptrdiff_t,wchar_t, wint_t, intmax_t, uintmax_t, sig_atomic_t, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, int_least8_t, int_least16_t, int_least32_t, int_least64_t, uint_least8_t, uint_least16_t.....，这些宏保证stddef.h stdint.h, wchar.t可以正确工作。不应该直接只用这些宏，而应该用头文件中的typedef。在有些系统中，由于GCC没有提供stdint.h，这些宏可能不能正确的工作。

__CHAR_BIT__

定义了char所用的位数。不应该直接只用这些宏，而应该用头文件中的typedef。

3.7.3 System-specific Predefined Macros

C预处理器定义了一些宏，可以用来表示当前所用的系统和机器。这些值在GCC支持的系统上不同。这个手册没有写出这些值，不过可以通过cpp -dM来查看。阅读Chapter 12[invocation]来了解更多。所有这些宏扩展为常量，因此可以通过#ifdef或者#if来查看。

C标准要求所有系统特定的宏作为保留命名空间（reserved namespace）的一部分。所有的宏以两个下划线，或者一个下划线和一个大写字母开始。之前一些老版本的系统特定的宏没有前缀的下划线，比如unix宏。所有这些历史遗留的宏，GCC提供了以双划线开始和结束的对应版本。如果unix定义了，那么就有__unix__定义。不会有超过两个下划线的命名，比如_mips对应的版本是__mips__。

当使用-ansi或者任意的-std选项，所有在保留命名空间以外的系统 特定预置宏会被忽略。对应的在保留命名空间的宏会被定义。

当前正在逐步淘汰不在预置命名空间的宏。不应该再在新写的代码中使用这些宏，同时鼓励将老代码中的宏改成

对应的新的宏。

3.7.4 C++ named operators

在C++中，有11个有命名的标点操作符。这些命名在预处理器中也同样适用。这些名字会被扩展为对应的操作符。

Named Operator	Punctuator
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	!=
xor	^
xor_eq	^=

在用gcc编译时在，这些操作符不会被识别，用g++时，这些宏可以识别。

3.8 Undefining and Redefining Macros

当一个宏不再有用，可以通过#undef来取消定义。#undef需要宏的名字作为参数来取消宏定义。只需要使用宏的名字，即便宏是function-like。当宏的名字后有任何的token都会报错（空白不算token【！！在4.9上验证】），如果undef后边的参数不是宏，那么#undef没有任何效果。

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

当宏被取消后，可以通过#define重新定义，新定义的宏和老的宏不需要有没有任何相似的地方。

【！！没有看明白】当一个宏被重新定义，新的定义应该和老的有同样的效果。两个宏定义满足以下条件时，他们的效果一样：

1）同样的类型（object-like or function-like）

2）所有宏体中的token一样

3）参数一样

4）空白出现在同样的地方，不过数量不需要一样。需要注意的是注释会作为空白。

这些宏是一样的

```
#define FOUR (2 + 2)
#define FOUR        (2    +    2)
#define FOUR         (2/*two*/+ 2)
```

这些宏不一样：

```
#define FOUR (2 + 2)
#define FOUR ( 2+2 )
#define FOUR (2 * 2)
#define FOUR (scour, and seven, years, ago) (2+2)
```

如果一个宏重新定以后，老的值和新的效果是行不一样，预处理器会发出warning，使用新的定义。如果新的定义和老的定义一样，那么新的定义会被忽略。这种方法允许两个不同的头文件，定义同一个宏，当这两个宏不一样的时候，预处理器会发出警告。【？？？】

3.9 Directive Within Macro Arguments【！！】

有时使用带参数的预处理指令会比较方便。C和C++的标准中，这种行为时未定义的。

CPP 3.2以前，带参数的预处理指令会报错，但是和普通的函数或者function-like宏相比较，带参数的预处理指令仅仅是语法上的差异，所有这个取消这个限制是有吸引力的，有时人们会觉着没有办法这么使用宏。有时候人们在参数列表中使用条件语句，比如printf，仅仅是为了找到在某一次库将printf升级为function-like的宏，并且这些代码不会被编译。所以从CPP 3.2开始，CPP可以处理有参数的预处理指令，这些指令和function-like宏一样处理。

如果，在一个宏被重新定义了这是新的定义会在参数扩展时使用，而老的定义依然会进行参数替代。这里有个病态的例子：

```
#define f(x) x x
f (1
#undef f
#define f 2
f)
```

这个会被扩展为

```
1 2 1 2
```

3.10 Macro Pitfalls

这一节，我们会描述一些特殊的规则，并且会指出这些和违反直觉的地方。

3.10.1 Misnesting (嵌套错误）

当一个宏通过参数被调用，参数（【也就是通过参数调用的宏】）会先进行扩展，再代入宏体，扩展后的结果被检查，然后一起和剩下的部分，进行更多的宏调用（When a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls）。一个宏调用可以一部分来自宏体，一部分来自宏参数。比如：

```
#define twice(x) (2*(x))
#define call_with_1(x) x(1)
call_with_1 (twice)          【译注：这里的twice还不是宏】
        ->twice(1)           【此时twice才是宏】
        ->(2*(1))
```

宏定义里括号不用匹配。在一个宏体中写开括号，就可以通过在一个通过这个宏体开始宏调用，然后在宏体外结束调用。比如：

```
#define strage(file) fprintf(file, "%s %d",
.....
strage(stderr) p, 35)
        ->fprintf(stderr, "%s %d", p, 35)
```

拼接宏体是有用的，但是不匹配的括号可读性不好，不建议使用。

3.10.2 Operator Precedence Problems

在之前的例子中，每个宏参数都有一个括号。在整个宏体又会有一个括号。这一节会解释为什么这么做是宏定义的最好的方法。

比如你定义这么一个宏，

```
#define ceil_div(x, y) (x + y - 1) / y
```

这么使用：

```
a = ceil_div (b & c, sizeof(int));
    ->a = (b & c + sizeof(int) - 1) / sizeof(int);
```

这个和设想的不同，由于优先级的关系，扩展后是：

```
a = (b & (c + sizeof (int) - 1)) / sizeof(int);
```

事实上我们期待的是：

```
a = ((b & c) + sizeof(int) -1)) / sizeof(int);
```

正确的写法应该是：

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

3.10.3 swallowing the semicolon(吞食分号)

通常宏会在组合的语句中扩展。考虑下边的宏，会对指针进行运算，其中夹杂着一些空格：

```
#define SKIP_SPACES(p, limit) \
    { char *lim = (limit);              \
        while (p < lim) {               \
            if (*p++ != ' '){           \
                p--;                    \
                break; }}}          \
```

【注：这个宏查找一个字符串中的空格，空格的地址】

调用这个宏，可能类似SKIP_SPACES(p, lim)。严格来说，调用会扩展为组合语句，这个语句组合不需要分号。不过，这个调用像很函数，为了比满误解，可以加上SKIP_SPACE(p, lim);。

不过这个调用在else语句时出错，应为分号是空语句，比如：

```
if (*p != 0)
        SKIP_SPACES(p, lim);
else .....
```

在if和else之间会有两个语句，这块代码就是错误的。

可以用do...while(0)来解决这个问题。

```
#define SKIP_SPACES(p, limit) \
    do{ char *lim = (limit);            \
        while (p < lim) {               \
            if (*p++ != ' '){           \
```

```
p--;                    \
break; }}}              \
while(0)            【注意，这里没有分号】
```

这样调用就会扩展为do...while(0);不会有问题。

3.10.4 Duplication of Side Effects

许多C程序定义min宏，比如：

```
#define min(x, y) ((x) < (y) ? (x) : (y))
```

当使用这个宏时，会有边际效果，比如：

```
next = min (x + y, foo(z));
```

这个宏会扩展为：

```
next = ((x+y) < (foo(z)) ? (x + y) : (foo(z)));
```

这里foo(z)从字面上看只写了一次，但是由于运算关系，就被调用了两次，可能不是你期望的这样。我们说这个min是不安全的宏。

最好的方法是只调用foo(z)一次。C语言没有提供标准的做法，不过GNU扩展可以做到这一点：

```
#define min(x, y)       \
({ typeof (x) _x = (x);   \
    typeof(y) _y = (y); \
    (x_ < y_)? x_ : y_;})
```

({...})把一个组合的语句，看作是一个语句。他的值是最后一句的值。这样允许我们定义局部变量，把每个参数赋给其中一个。本地变量以_开始是为了和参数的区分。这样每个参数都只会计算一次。

如果你不愿意使用GNU C扩展，那只能是小心的使用min。比如你可以先计算foo(z)的值，将这个值保存在本地变量里，然后再使用min.

```
#define min(x,y) ((x)<(y)?(x):(y))
    ....
int tem = foo(z);
next = min(x + y, tem);
```

3.10.5 Self-referential Macros

self-referential macro是宏的名字出现在宏的定义之中。宏定义会产生更多的宏替代。如果考虑使用自引用宏，就会产生一个无限递归的扩展。为了预防这种情况，子引用的宏不会认为是宏调用。传递到预处理器中后，输出不会改变。比如：

```
#define foo (4 + foo)
```

foo同时 也是个变量。

按照通常的规则，每个foo都会扩展为(4+foo),然后接下来是(4 + (4 + foo))直到耗光内存为止。

自引用的规则限定扩展只进行一次。因此上，这个宏会foo的值加4。

在大多数情况下，自引用宏弊大于利。看代码是，如果看到foo是个变量，就很难想到也是个宏扩展。

有个常用的情况是，定义一个子扩展的宏。

```
#define EPERM EPERM
```

【？？】

如果x扩展需要y,而y需要x扩展，这种情况是间接自我引用。扩展也遵从只扩展一次的规则。

```
#define x (4 + y)
```

```
#define y (2 + x)
x -> (4 + y)
    -> ( 4 + (2 + x))
y -> ( 2 + x)
    -> (x + (y + 4))
```

3.10.6 Argument Prescan

除了stringify或者pasted，其他情况下，宏参数会在宏体扩展前进行宏扩展。在宏参数扩展后，整个宏体，包括带入的参数会进行第二次的宏扩展。这样的结果就是宏体参数会进行两次扩展。

大多数情况下，这样没有什么用。如果参数有任何宏调用，会在第一次scan时候被扩展。这样结果就不会有宏函数调用，这样第二次就会在有什么改变。如果替换时前没有扩展，第二次就会扩展宏，和之前的结果一样。

你可能期望两次检查来改变自引用的问题，进行两次扩展，事实上这不会发生。

你可能想知道，为什么要提到预扫描参数，这里是否有什么不同？为什么不跳过预扫描，这样可以更快？原因是预扫描在以下三个特例中会有不同：

1.嵌套宏调用

当一个宏参数包含自身的调用时，我们称之为嵌套宏调用。比如，如果f是一个需要一个参数的宏，f(f(1))就是一个嵌套的f调用。期望的扩展方式是，先扩展f(1)，然后将值带入f进行扩展。预扫描可以保证结果和预期的一样。如果没有预扫描，f(1)就会作为参数带入，宏体扩展时就会认为是自引用。

2.一个宏，调用stringify和pasted的宏。

如果一个参数是stringify或者pasted，那么预扫描就不会发生。如果你想先扩展一个宏，然后再进行stringify或者pasted，那么你需要先一个宏，调用另外一个定义了stringify或者pasted的宏。

```
#define AFTER(x) X_ ## x
#define XAFTER(x) AFTER(x)
#define TABLESIZE 1024
#define BUFSIZE TABLESIZE
```

AFTER(BUFSIZE) 会扩展为X_BUFERSIZE， XAFTER(BUFSIZE)会扩展为X_1024.

3.将宏作为参数，宏体里有非正常C语句的逗号

这个会在宏体第二次扩展时引起错误的参数。

```
#define foo a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))
```

我们希望bar(foo)扩展为(1+(foo))，进而扩展为(1+(a,b))。而事实上，bar(foo)会扩展为bar(a,b), lose(a,b)，而lose只需要一个参数，这是个错误。要解决这个问题，只需要增加一个括号即可：

```
#define for (a, b)
```

或者

```
#define bar(x) lose((x))
```

3.10.7 newlines in Arguments

function-like宏可以扩行写很多内容。在当前的编译器实现上，在扩展时会扩展成一行。这样，编译器或者debugger生成的行号就只有一行，查看行号时，就会有不准确的问题。

```
1  #define ignore_second_arg(a, b, c) a;c
2
3  ignore_second_arg(foo(),
4                                    ignored(),
5                                   syntax error);
```

这个语句在syntax error的语句会错误，错误显示的行号是3，也就是ignore_second_arg这一行，而事实上错误是第五行。

我们认为这是个错误，会在以后修改这个问题。