**预处理器**

ANSI C标准不仅描述了C语言，还描述了C预处理器的工作机制。预处理器可以根据您的请求包含其他文件，还可以选择让编译器处理哪些代码。预处理器不能理解C，它一般是接受一些文本并将其转换成其他文本。

**一、翻译程序的第一步**

在程序作预处理之前，代码会先被进行几次翻译处理：

1. 编译器首先把源代码中出现的字符映射到源字符集。该过程处理多字节字符和三元符（trigraph）扩展。

2. 编译器查找反斜线后紧跟换行的实例并删除这些实例。也就是说，将类似下面的两个物理行：

printf ("That's wond\

erful!\n");

转换成一个逻辑行（logical line）：

printf ("That's wonderful\n");

3. 编译器将文本划分成预处理的语言符号（token）序列和空白字符及注释序列。应注意的是编译器用一个空格字符代替每一个注释。

4. C实现可能还会选用单个空格代替每一个空白字符（不包括换行符）序列。

最后，程序进入预处理阶段。

**二、明显常量：#define**

1. 预处理器指令从#开始，到其后第一个换行符为止。也就是说，指令的长度限于一行代码。但正如前文提到的，在预处理开始前，系统会删除反斜线和换行符的组合。因此可以把指令扩展到一个物理行，由这些物理行组成单个逻辑行。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\* preproc.c - 简单的预处理器的例子 \*/  #include <stdio.h>  #define TWO 2  #define OW "Consistency is the last refuge of the unimagia\  tive. - Oscar Wilde" /\* 反斜线把这个定义延续到下一行 \*/  #define FOUR TWO\*TWO  #define PX printf("X is %d.\n", x)  #define FMT "X is %d.\n"    int main (void)  {      int x = TWO;        PX;      x = FOUR;      printf(FMT, x);      printf("%s\n", OW);      printf("TWO: OW\n");        return 0;  } |

每个#define行由三部分组成。第一部分为指令#define自身。第二部分为所选择的缩略语，称为**宏（macro）**。宏的名字中不允许有空格，而且必须遵循C变量命名规则。从宏变成最终的替换文本的过程称为**宏展开（macro expansion）**。

2. 宏展开的过程是依此进行的，所以

x = FOUR;

变成了：

x = TWO\*TWO

然后又变成：

x = 2\*2;

3. 示例程序中，我们在一行结尾加反斜线符号以使该行扩展至下一行：

#define OW "Consistency is the last refuge of the unimagia\

tive. - Oscar Wilde" /\* 反斜线把这个定义延续到下一行 \*/

注意第二行要左对齐。相反，假设这样做：

#define OW "Consistency is the last refuge of the unimagia\

tive. - Oscar Wilde" /\* 反斜线把这个定义延续到下一行 \*/

则输出将会是：

"Consistency is the last refuge of the unimagia    tive. - Oscar Wilde"

4. 注意双引号中的宏不会被替换。

5. 注意，const关键字已经得到了C的支持，这为创建常量提供了更灵活的方法。但是，宏常量仍可以用来制定标准数组的大小并作为const值的初始化值。

[复制代码](javascript:void(0);)

#define LIMIT 20

const int LIM = 50;

static int data1[LIMIT]; // 合法

// static int data2[LIM]; // 无效

const int LIM2 = 2 \* LIMIT; // 合法

// const int LIM3 = 2 \* LIM; // 无效

[复制代码](javascript:void(0);)

6. 语言符号（token）

从技术方面看，系统把宏的主体当作**语言符号（token）**类型字符串，而不是字符型字符串。C预处理器中的语言符号是宏定义主体中的单独的“词（word）”。用空白字符把这些词分开。比如：

#define FOUR 2\*2

 这个定义中有一个语言符号：即序列2\*2。但是：

#define SIX 2 \* 3

这个定义中有三个语言符号：2、\*和3。

注意，作为在token类型字符串，空格不是主体的一部分，只是作为分隔符号。

7. 重定义变量

对于宏的重新定义，ANSI C标准只允许新定义与旧定义完全相同（语言符号也要相同）。

所以，下面两个定义被认为是相同的，因为它们都有三个语言符号：2、\*和3。

#define SIX 2 \* 3

#define SIX 2 \* 3

而下面的定义则被认为于上面两个不同，因为它只有一个语言符号：2\*3。

#define SIX 2\*3

**三、在#define中使用参数**

C有两种宏，一种为“**类对象宏（object-like macro）**”，即上个部分所示那样，进行直接的替换；另一种为“**类函数宏（function-like macro）**”，可以在宏中使用参数。

1. 下面是一个类函数宏定义的示例：

#define SQUARE (X) X \* X

在程序中可以这样使用：

x = SQUARE (2);

将被替换成：

x = 2 \* 2;

2. 类函数宏定义的组成：

|  |  |
| --- | --- |
| #define | 指令 |
| SQUARE | 宏名称 |
| (X) | 宏参数列表 |
| X \* X | 宏替换的内容 |

注意，宏名称与宏参数列表之间尽量不要隔开（不同编译器支持程度不同）。但指令于宏名称、宏参数列表与替换内容之间一定要隔开。

3. 注意类函数宏仍进行的是直接的替换，只是把参数另替换为制定的形式，不会进行其他处理。所以这其中就会引起一些问题。比如，SQUARE(x + 2)将被替换为x + 2 \* x + 2，而不是想要的(x + 2) \* (x + 2)。

类函数宏正确的使用方式是，将替换内容中的参数及替换内容整体放到括号中，以保证结果是原想要的形式，即

#define SQURE (X) ((X) \* (X))

4. 但是，即使这样，也还要在宏参数中避免使用自增和自减运算。比如：

“y = SQURE(x++);”会被替换为“y = ((x++) \* (x++));”

此时x自增的顺序在标准中是未定义的，将会依赖于编译器的实现。

5. 利用宏参数创建字符串：#运算符

加入您希望在字符串中包含宏参数，可以使用**#运算符**，它将把语言符号转化为字符串。代码示例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | /\* subst.c - 在字符串中进行替换 \*/  #include <stdio.h>    #define PSQR(x) printf ("The square of "#x" is %d.\n", ((x) \* (x)))    int main (void)  {      int y = 5;        PSQR (y);      PSQR (2 + 4);        return 0;  } |

输入如下：

The square of y is 25.

The square of 2 + 4 is 36.

第一次调用宏时，用“y”代替#x；第二次调用宏时，用“2 + 4”代替#x。

6. 宏处理的粘合剂：##运算符

**##运算符**可以把两个语言符号组合成单个语言符号。例如，可以定义如下的宏：

#define XNAME(n) x ## n

这样，下面的宏调用：

XNAME (4);

会被展开成下列形式：

x4

代码示例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // glue.c -- 使用##运算符  #include <stdio.h>    #define XNAME(n) x##n  #define PRINT\_XN(n) printf ("x"#n" = %d\n", x##n)    int main (void)  {      int XNAME(1) = 14; // 变为 int x1 = 14;      int XNAME(2) = 20; // 变为 int x2 = 20;      PRINT\_XN(1); // 变为 printf("x1 = %d\n", x1);      PRINT\_XN(2); // 变为 printf("x2 = %d\n", x2);        return 0;  } |

输出：

x1 = 14

x2 = 20

注意宏PRINT\_XN()使用#运算符组合字符串，使用##运算符把两个语言符号组合为一个新的标识符。

7. 可变宏：...和\_\_VA\_ARGS\_\_

有些函数接受可变数量的参数。C99对宏也可以有同样的工作，实现思想就是宏定义中参数列表的最后一个参数为省略号（也就是三个句号）。这样，预定义宏**\_\_VA\_ARGS\_\_**就可以被用在替换部分中，以表明省略号代表什么。例如，考虑下面的定义：

#define PR(...) printf (\_\_VA\_ARGS\_\_)

假设稍后用下面的方式调用该宏：

PR("Howdy");

PR("weight = %d, shipping = $.2f\n", wt, sp);

第一次调用中，\_\_VA\_ARGS\_\_展开为1个参数：

"Howdy"

第二次调用中，它展开为3个参数：

"weight = %d, shipping = $%.2f\n", wt, sp

因此，展开后的代码为：

printf ("Howdy");

printf ("weight = %d, shipping = $%.2f\n", wt, sp);

代码示例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // variadic.c -- 可变宏  #include <stdio.h>  #include <math.h>    #define PR(X, ...) printf("Message "#X": ", \_\_VA\_ARGS\_\_)    int main (void)  {      double x = 48;      double y;        y = sqrt (x);      PR (1, "x = %g\n", x);      PR (2, "x = %.2f, y = %.4f\n", x, y);        return 0;  } |

注意，可变宏不限定宏参数中必须有普通参数，但省略号只能代替最后的宏参数。

**四、文件包括：#include**

预处理器发现**#include**指令后，就会寻找后跟的文件名并把这个文件的内容包括到当前文件中。被包括文件中的文本将替换源代码文件中的#include指令，就像您把被包含文件中的全部内容键入到源文件中的这个特定位置一样。

1. #include指令有两种使用形式：

|  |  |
| --- | --- |
| #include <stdio.h> | 文件名放在尖括号中 |
| #include "mystuff.h" | 文件名放在双引号中 |

在UNIX系统中，尖括号告诉预处理器在一个或多个标准系统目录中寻找文件。双引号告诉预处理器先在当前目录（或文件名中指定的其他目录）中寻找文件，然后在标准位置寻找文件。

|  |  |
| --- | --- |
| #include <stdio.h> | 搜索系统目录 |
| #include "hot.h" | 搜索当前工作目录 |
| #include "/usr/biff/p.h" | 搜索/usr/biff目录 |

2. 使用头文件

习惯上使用后缀.h表示头文件（header file），这类文件包含置于程序头部的信息。头文件内容的最常见形式包括：

* **明显常量**——例如，典型的stdio.h文件定义EOF、NULL和BUFSIZE（标准I/O缓冲区的大小）。
* **宏函数**——例如，getchar( )通常被定义为getc (stdin)，getc()通常被定义为较复杂的宏，而头文件ctype.h通常包含ctype函数的宏定义。
* **函数声明**——例如，头文件string.h包含字符串函数系列的函数声明。在ANSI C中，声明采用函数原型形式。
* **结构模版定义**——标准I/O函数使用FILE结构，该结构包含文件及文件相关缓冲区的信息。头文件stdio.h中存放FILE结构的声明。
* **类型定义**——可以使用只想FILE的指针作为参数调用标准I/O函数。通常，stdio.h用#define或typedef使得FILE代表指向FILE结构的指针。与之类似。size\_t和time\_t类型也在头文件中定义。

**五、其他指令**

1. #undef指令

**#undef指令**取消定义一个给定的#define。取消一个未定义的宏也是合法的，这会被自动忽略掉。

宏的已定义：#define宏的作用域从文件中地定义点开始，直到用#undef指令取消宏为止，或直到文件尾为止。

注意：C中有几个预定义的宏，如\_\_DATA\_\_、\_\_FILE\_\_和\_\_VA\_ARGS\_\_，这些宏总被认为是已定义的，而且不能被取消定义。

2. 条件编译

条件编译告诉编译器，根据编译时的条件接受或忽略代码块。

**#ifdef、#else和#endif指令**

#ifdef——如果已经定义宏……

#else——否则

#endif——结束if判断

一个简短的示例：

[复制代码](javascript:void(0);)

#ifdef MAVIS

#include "horse.h"

#define STABLES 5

#else

#include "cow.h"

#define STABLES 15

#endif

[复制代码](javascript:void(0);)

**#ifndef指令**

#ifndef——如果未定义宏……

类似#ifdef指令，可以与#else指令、#endif指令一起使用

#ifndef SIZE

#define SIZE 100

#endif

 #ifndef指令通常用于防止多次包含同一文件。也就是说，头文件可采用类似下面几行的设置：

#ifndef THINGS\_H

#define THINGS\_H

/\* 头文件其他部分 \*/

#endif

**#if和#elif指令**

#if就像常规的if：#if后跟常量整数表达式。如果表达式为非零值，则表达式为真。在该表达式中可以使用关系运算符和逻辑运算符。

#if SYS == 1

#include "ibm.h"

#endif

可以使用#elif指令扩展if-else序列

[复制代码](javascript:void(0);)

#if SYS == 1

#include "imb.h"

#elif SYS == 2

#include "vax.h"

#elif SYS == 3

#include "mac.h"

#else

#include "general.h"

#endif

[复制代码](javascript:void(0);)

**defined预处理运算符**

#if可以和defined组合使用实现#ifdef指令的功能。以下两个语句的作用是相同的：

#ifdef VAX

#if defined VAX

3. 预定义宏

如前面所说，预定义宏是无法被取消定义的。

C标准指定的一些预定义宏如下表所示：

|  |  |
| --- | --- |
| 预定义宏 | |
| 宏 | 意义 |
| **\_\_DATA\_\_** | 进行预处理的日期（"Mmm dd yyyy"形式的字符串文字） |
| **\_\_FILE\_\_** | 代表当前源代码文件名的字符串文字 |
| **\_\_LINE\_\_** | 代表当前源代码文件中的行号的整数常量 |
| **\_\_STDC\_\_** | 设置为1时，表示该实现遵循C标准 |
| **\_\_STDC\_HOSTED\_\_** | 为本机环境设置为1，否则为0 |
| **\_\_STDC\_VERSION\_\_** | 为C99时设置为19901L |
| **\_\_TIME\_\_** | 源文件编译时间，格式为"hh: mm: ss" |

C99标准还提供一个名为**\_\_func\_\_**的预定义标识符。\_\_func\_\_展开为一个代表函数名的字符串。该标识符具有函数作用域，因而不是一个预定义宏，而是一个预定以标识符。

4. #line和#error

**#line指令**用于重置由\_\_LINE\_\_和\_\_FILE\_\_宏报告的行号和文件名。可以这样使用#line：

#line 1000 // 把当前行号重置为1000

#line 10 "cool.c" // 把行号重置为10，文件名重置为cool.c

**#error指令**使预处理器发出一条错误信息，该信息包含指令中的文本。可能的话，编译过程应该中断。可以这样使用#error：

#if \_\_STDC\_VERSION\_\_ != 199901L

#error Not C99

#endif

5. #pragma

#pragma用于修改编译器的某些设置。

例如，在开发C99时，用C9X代表C99。编译器可以使用下面的编译指示（pragma）来启用对C9X的支持：

#pragma c9x on

C99还提供了\_Pragma预处理器运算符，可将字符串转换成常规的编译器指示。例如：

\_Pragma("nonstandardtreatmenttypeB on")

等价于下面的指令：

#pragma nonstandardtreatmenttypeB on

\_Pragma运算符还会完成字符串析构（destringizing）工作；也就是说，将字符串中的转义序列转换成它所代表的字符。因而：

\_Pragma("use\_bool \"true\" \"false\"")

变成：

#pragma use\_bool "true" "false"