

# C 语言预处理指令详解

## 一 前言

预处理(或称预编译)是指在进行编译的第一遍扫描(词法扫描和语法分析)之前所作的工作。预处理指令指示在程序正式编译前就由编译器进行的操作,可放在程序中任何位置。

预处理是 C 语言的一个重要功能,它由预处理程序负责完成。当对一个源文件进行编译时,系统将自动引用预处理程序对源程序中的预处理部分作处理,处理完毕自动进入对源程序的编译。

C 语言提供多种预处理功能,主要处理 # 开始的预编译指令,如宏定义(`#define`)、文件包含(`#include`)、条件编译(`#ifdef`)等。合理使用预处理功能编写的程序便于阅读、修改、移植和调试,也有利于模块化程序设计。

本文参考诸多资料,详细介绍常用的几种预处理功能。因成文较早,资料来源大多已不可考,敬请谅解。

## 二 宏定义

C 语言源程序中允许用一个标识符来表示一个字符串,称为“宏”。被定义为宏的标识符称为“宏名”。在编译预处理时,对程序中所有出现的宏名,都用宏定义中的字符串去代换,这称为宏替换或宏展开。

宏定义是由源程序中的宏定义命令完成的。宏替换是由预处理程序自动完成的。

在 C 语言中,宏定义分为有参数和无参数两种。下面分别讨论这两种宏的定义和调用。

### 2.1 无参宏定义

无参宏的宏名后不带参数。其定义的一般形式为:

```
#define 标识符 字符串
```

其中,“#”表示这是一条预处理命令(以 # 开头的均为预处理命令)。“define”为宏定义命令。“标识符”为符号常量,即宏名。“字符串”可以是常数、表达式、格式串等。

宏定义用宏名来表示一个字符串,在宏展开时又以该字符串取代宏名。这只是一种简单的文本替换,预处理程序对它不作任何检查。如有错误,只能在编译已被宏展开后的源程序时发现。

注意理解宏替换中“换”的概念,即在对相关命令或语句的含义和功能作具体分析之前就要进行文本替换。

【例 1】定义常量:

```
1 #define MAX_TIME 1000
```

若在程序里面写 `if(time < MAX_TIME){.....}`,则编译器在处理该代码前会将 `MAX_TIME` 替换为 `1000`。

注意,这种情况下使用 `const` 定义常量可能更好,如 `const int MAX_TIME = 1000;`。因为 `const` 常量有数据类型,而宏常量没有数据类型。编译器可以对前者进行类型安全检查,而对后者只进行简单的字符文本替换,没有类型安全检查,并且在字符替换时可能会产生意料不到的错误。

【例 2】反例:

```
1 #define pint (int*)
2 pint pa, pb;
```

本意是定义 `pa` 和 `pb` 均为 `int` 型指针,但实际上变成 `int* pa,pb;`。`pa` 是 `int` 型指针,而 `pb` 是 `int` 型变量。本例中可用 `typedef` 来代替 `define`,这样 `pa` 和 `pb` 就都是 `int` 型指针了。因为宏定义只是简单的字符串代换,在预处理阶

段完成，而 **typedef** 是在编译时处理的，它不是作简单的代换，而是对类型说明符重新命名，被命名的标识符具有类型定义说明的功能。**typedef** 的具体说明见附录 6.4。

无参宏注意事项：

- 宏名一般用大写字母表示，以便于与变量区别。
- 宏定义末尾不必加分号，否则连分号一并替换。
- 宏定义可以嵌套。
- 可用 **#undef** 命令终止宏定义的作用域。
- 使用宏可提高程序通用性和易读性，减少不一致性，减少输入错误和便于修改。如数组大小常用宏定义。
- 预处理是在编译之前的处理，而编译工作的任务之一就是语法检查，预处理不做语法检查。
- 宏定义写在函数的花括号外边，作用域为其后的程序，通常在文件的最开头。
- 字符串 " " 中永远不包含宏，否则该宏名当字符串处理。
- 宏定义不分配内存，变量定义分配内存。

## 2.2 带参宏定义

C 语言允许宏带有参数。在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。

对带参数的宏，在调用中，不仅要宏展开，而且要用实参去代换形参。

带参宏定义的一般形式为：

```
#define 宏名(形参表) 字符串
```

在字符串中含有各个形参。

带参宏调用的一般形式为：

```
宏名(实参表);
```

在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。

在带参宏定义中，形参不分配内存单元，因此不必作类型定义。而宏调用中的实参有具体的值，要用它们去代换形参，因此必须作类型说明，这点与函数不同。函数中形参和实参是两个不同的量，各有自己的作用域，调用时要把实参值赋予形参，进行“值传递”。而在带参宏中只是符号代换，不存在值传递问题。

【例 3】

```
1 #define INC(x) x+1 //宏定义
2 y = INC(5);        //宏调用
```

在宏调用时，用实参 5 去代替形参 x，经预处理宏展开后的语句为 **y=5+1**。

【例 4】反例：

```
1 #define SQ(r) r*r
```

上述这种实参为表达式的宏定义，在一般使用时没有问题；但遇到如 `area=SQ(a+b);` 时就会出现问题，宏展开后变为 `area=a+b*a+b;`，显然违背本意。

相比之下，函数调用时会先把实参表达式的值(`a+b`)求出来再赋予形参 `r`；而宏替换对实参表达式不作计算直接地照原样代换。因此在宏定义中，字符串内的形参通常要用括号括起来以避免出错。

进一步地，考虑到运算符优先级和结合性，遇到 `area=10/SQ(a+b);` 时即使形参加括号仍会出错。因此，还应在宏定义中的整个字符串外加括号，

综上，正确的宏定义是 `#define SQ(r) ((r)*(r))`，即宏定义时建议所有的层次都要加括号。

【例 5】带参函数和带参宏的区别：

```
1 #define SQUARE(x) ((x)*(x))
2 int Square(int x) {
3     return (x * x); //未考虑溢出保护
4 }
5
6 int main(void) {
7     int i = 1;
8     while(i <= 5)
9         printf("i = %d, Square = %d\n", i, Square(i++));
10
11     int j = 1;
12     while(j <= 5)
13         printf("j = %d, SQUARE = %d\n", j, SQUARE(j++));
14
15     return 0;
16 }
```

执行后输出如下：

```
1 i = 2, Square = 1
2 i = 3, Square = 4
3 i = 4, Square = 9
4 i = 5, Square = 16
5 i = 6, Square = 25
6 j = 3, SQUARE = 1
7 j = 5, SQUARE = 9
8 j = 7, SQUARE = 25
```

本例意在说明，把同一表达式用函数处理与用宏处理两者的结果有可能是不同的。

调用 `Square` 函数时，把实参 `i` 值传给形参 `x` 后自增 1，再输出函数值。因此循环 5 次，输出 1~5 的平方值。

调用 `SQUARE` 宏时，`SQUARE(j++)` 被代换为 `((j++)*(j++))`。在第一次循环时，表达式中 `j` 初值为 1，两者相乘的结果为 1。相乘后 `j` 自增两次变为 3，因此表达式中第二次相乘时结果为 `3*3=9`。同理，第三次相乘时结果为 `5*5=25`，并在此次循环后 `j` 值变为 7，不再满足循环条件，停止循环。

从以上分析可以看出函数调用和宏调用二者在形式上相似，在本质上是完全不同的。

带参宏注意事项：

- 宏名和形参表的括号间不能有空格。
- 宏替换只作替换，不做计算，不做表达式求解。
- 函数调用在编译后程序运行时进行，并且分配内存。宏替换在编译前进行，不分配内存。
- 宏的哑实结合不存在类型，也没有类型转换。
- 函数只有一个返回值，利用宏则可以设法得到多个值。
- 宏展开使源程序变长，函数调用不会。
- 宏展开不占用运行时间，只占编译时间，函数调用占运行时间(分配内存、保留现场、值传递、返回值)。
- 为防止无限制递归展开，当宏调用自身时，不再继续展开。如：`#define TEST(x) (x + TEST(x))`被展开为 `1 + TEST(1)`。

## 2.3 实践用例

包括基本用法(及技巧)和特殊用法(`#`和`##`等)。

`#define` 可以定义多条语句，以替代多行的代码，但应注意替换后的形式，避免出错。宏定义在换行时要加上一个反斜杠“`\`”，而且反斜杠后面直接回车，不能有空格。

### 2.3.1 基本用法

1. 定义常量：

```
1 #define PI    3.1415926
```

将程序中出现的 `PI` 全部换成 `3.1415926`。

2. 定义表达式：

```
1 #define M    (y*y+3*y)
```

编码时所有的表达式(`y*y+3*y`)都可由 `M` 代替，而编译时先由预处理程序进行宏替换，即用(`y*y+3*y`)表达式去置换所有的宏名 `M`，然后再进行编译。

注意，在宏定义中表达式(`y*y+3*y`)两边的括号不能少，否则可能会发生错误。如 `s=3*M+4*M` 在预处理时经宏展开变为 `s=3*(y*y+3*y)+4*(y*y+3*y)`，如果宏定义时不加括号就展开为 `s=3*y*y+3*y+4*y*y+3*y`，显然不符合原意。因此在作宏定义时必须十分注意。应保证在宏替换之后不发生错误。

3. 得到指定地址上的一个字节或字：

```
1 #define MEM_B(x)    (*(char *) (x))
2 #define MEM_W(x)    (*(short *) (x))
```

4. 求最大值和最小值：

```
1 #define MAX(x, y)    (((x) > (y)) ? (x) : (y))
2 #define MIN(x, y)    (((x) < (y)) ? (x) : (y))
```

以后使用 **MAX (x,y)**或 **MIN (x,y)**，就可分别得到 **x** 和 **y** 中较大或较小的数。

但这种方法存在弊病，例如执行 **MAX(x++, y)**时，**x++**被执行多少次取决于 **x** 和 **y** 的大小；当宏参数为函数也会存在类似的风险。所以建议用内联函数而不是这种方法提高速度。不过，虽然存在这样的弊病，但宏定义非常灵活，因为 **x** 和 **y** 可以是各种数据类型。

以下给出 **MAX** 宏的两个安全版本(源自 **linux/kernel.h**):

```
1 #define MAX_S(x, y) ({ \
2     const typeof(x) _x = (x); \
3     const typeof(y) _y = (y); \
4     (void)(&_x == &_y); \
5     _x > _y ? _x : _y; })
6
7 #define TMAX_S(type, x, y) ({ \
8     type _x = (x); \
9     type _y = (y); \
10    _x > _y ? _x: _y; })
```

Gcc 编译器将包含在圆括号和大括号双层括号内的复合语句看作是一个表达式，它可出现在任何允许表达式的地方；复合语句中可声明局部变量，判断循环条件等复杂处理。而表达式的最后一条语句必须是一个表达式，它的计算结果作为返回值。**MAX\_S** 和 **TMAX\_S** 宏内就定义局部变量以消除参数副作用。

**MAX\_S** 宏内 **(void)(&\_x == &\_y)** 语句用于检查参数类型一致性。当参数 **x** 和 **y** 类型不同时，会产生“**comparison of distinct pointer types lacks a cast**”的编译警告。

注意，**MAX\_S** 和 **TMAX\_S** 宏虽可避免参数副作用，但会增加内存开销并降低执行效率。若使用者能保证宏参数不存在副作用，则可选用普通定义(即 **MAX** 宏)。

5. 得到一个成员在结构体中的偏移量(lint 545 告警表示“&用法值得怀疑”，此处抑制该警告):

```
1 #define FPOS(type, field) \
2 /*lint -e545 */ (((int)&((type *)0)-> field) /*lint +e545 */
```

6. 得到一个结构体中某成员所占用的字节数:

```
1 #define FSIZ(type, field)    sizeof(((type *)0)->field)
```

7. 按照 **LSB** 格式把两个字节转化为一个字(**word**):

```
1 #define FLIPW(arr)          (((short)(arr)[0]) * 256) + (arr)[1])
```

8. 按照 **LSB** 格式把一个字(**word**)转化为两个字节:

```
1 #define FLOPW(arr, val) \
2     (arr)[0] = ((val) / 256); \
3     (arr)[1] = ((val) & 0xFF)
```

9. 得到一个变量的地址:

```
1 #define B_PTR(var)          ((char *) (void *)&(var))
2 #define W_PTR(var)          ((short *) (void *)&(var))
```

10. 得到一个字(word)的高位和低位字节:

```
1 #define WORD_LO(x)      ((char)((short)(x)&0xFF))
2 #define WORD_HI(x)      ((char)((short)(x)>>0x8))
```

11. 返回一个比 X 大的最接近的 8 的倍数:

```
1 #define RND8(x)          (((x) + 7) / 8) * 8)
```

12. 将一个字母转换为大写或小写:

```
1 #define UPCASE(c)        (((c) >= 'a' && (c) <= 'z') ? ((c) + 'A' - 'a') : (c))
2 #define LOCASE(c)        (((c) >= 'A' && (c) <= 'Z') ? ((c) + 'a' - 'A') : (c))
```

注意, **UPCASE** 和 **LOCASE** 宏仅适用于 **ASCII** 编码(依赖于码字顺序和连续性), 而不适用于 **EBCDIC** 编码。

13. 判断字符是不是 10 进值的数字:

```
1 #define ISDEC(c)         ((c) >= '0' && (c) <= '9')
```

14. 判断字符是不是 16 进值的数字:

```
1 #define ISHEX(c)         (((c) >= '0' && (c) <= '9') || \
2      ((c) >= 'A' && (c) <= 'F') || \
3      ((c) >= 'a' && (c) <= 'f'))
```

15. 防止溢出的一个方法:

```
1 #define INC_SAT(val)      (val = ((val)+1 > (val)) ? (val)+1 : (val))
```

16. 返回数组元素的个数:

```
1 #define ARR_SIZE(arr)     (sizeof((arr)) / sizeof((arr[0])))
```

17. 对于 IO 空间映射在存储空间的结构, 输入输出处理:

```
1 #define INP(port)         (*((volatile char *) (port)))
2 #define INPW(port)        (*((volatile short *) (port)))
3 #define INPDW(port)       (*((volatile int *) (port)))
4 #define OUTP(port, val)   (*((volatile char *) (port)) = ((char)(val)))
5 #define OUTPW(port, val)  (*((volatile short *) (port)) = ((short)(val)))
6 #define OUTPDW(port, val) (*((volatile int *) (port)) = ((int)(val)))
```

18. 使用一些宏跟踪调试:

**ANSI** 标准说明了五个预定义的宏名(注意双下划线), 即: **\_\_LINE\_\_**、**\_\_FILE\_\_**、**\_\_DATE\_\_**、**\_\_TIME\_\_**、**\_\_STDC\_\_**。

若编译器未遵循 **ANSI** 标准, 则可能仅支持以上宏名中的几个, 或根本不支持。此外, 编译程序可能还提供其它预定义的宏名(如 **\_\_FUCTION\_\_**)。

**\_\_DATE\_\_** 宏指令含有形式为月/日/年的串, 表示源文件被翻译到代码时的日期; 源代码翻译到目标代码的时间作为串包含在 **\_\_TIME\_\_** 中。串形式为时:分:秒。

如果实现是标准的, 则宏 **\_\_STDC\_\_** 含有十进制常量 **1**。如果它含有任何其它数, 则实现是非标准的。

可以借助上面的宏来定义调试宏, 输出数据信息和所在文件所在行。如下所示:

```
1 #define MSG(msg, date)
printf(msg);printf( "[%d] [%d] [%s]" , date, __LINE__, __FILE__)
```

19. 用 `do{...}while(0)` 语句包含多语句防止错误:

```
1 #define D0(a, b) do{\
2     a+b;\
3     a++;\
4 }while(0)
```

20. 实现类似“重载”功能

C 语言中没有 `swap` 函数, 而且不支持重载, 也没有模板概念, 所以对于每种数据类型都要写出相应的 `swap` 函数, 如:

```
1 IntSwap(int *, int *);
2 LongSwap(long *, long *);
3 StringSwap(char *, char *);
```

可采用宏定义 `TSWAP (t,x,y)` 或 `SWAP(x, y)` 交换两个整型或浮点参数:

```
1 #define TSWAP(type, x, y) do{ \
2     type _y = y; \
3     y = x; \
4     x = _y; \
5 }while(0)
6 #define SWAP(x, y) do{ \
7     x = x + y; \
8     y = x - y; \
9     x = x - y; \
10 }while(0)
11
12 int main(void) {
13     int a = 10, b = 5;
14     TSWAP(int, a, b);
15     printf( "a=%d, b=%d\n" , a, b);
16     return 0;
17 }
```

21. 1 年中有多少秒(忽略闰年问题) :

```
1 #define SECONDS_PER_YEAR (60UL * 60 * 24 * 365)
```

该表达式将使一个 16 位机的整型数溢出, 因此用长整型符号 `L` 告诉编译器该常数为长整型数。

注意, 不可定义为 `#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL`, 否则将产生 `(31536000)UL` 而非 `31536000UL`, 这会导致编译报错。

以下几种写法也正确:

```
1 #define SECONDS_PER_YEAR 60 * 60 * 24 * 365UL
```

```
2 #define SECONDS_PER_YEAR (60UL * 60UL * 24UL * 365UL)
3 #define SECONDS_PER_YEAR ((unsigned long)(60 * 60 * 24 * 365))
```

22. 取消宏定义:

```
#define [MacroName]
[MacroValue]           //定义宏
#undef
[MacroName]           //
取消宏
```

宏定义必须写在函数外，其作用域为宏定义起到源程序结束。如要终止其作用域可使用 `#undef` 命令:

```
1 #define PI 3.14159
2 int main(void) {
3     //.....
4 }
5 #undef PI
6 int func(void) {
7     //.....
8 }
```

表示 `PI` 只在 `main` 函数中有效，在 `func1` 中无效。

### 2.3.2 特殊用法

主要涉及 C 语言宏里 `#` 和 `##` 的用法，以及可变参数宏。

#### 2.3.2.1 字符串化操作符 `#`

在 C 语言的宏中，`#` 的功能是将其后面的宏参数进行字符串化操作 (**Stringfication**)，简单说就是将宏定义中的传入参数名转换成用一对双引号括起来参数名字符串。`#` 只能用于有传入参数的宏定义中，且必须置于宏定义体中的参数名前。例如:

```
1 #define EXAMPLE(instr) printf("The input string is:\t%s\n", #instr)
2 #define EXAMPLE1(instr) #instr
```

当使用该宏定义时，`example(abc)` 在编译时将会展开成 `printf("the input string is:\t%s\n","abc");`；`string str=example1(abc)` 将会展成 `string str="abc"`。

又如下面代码中的宏:

```
1 define WARN_IF(exp) do{ \
2     if(exp) \
3         fprintf(stderr, "Warning: " #exp "\n"); \
4 }while(0)
```

则代码 `WARN_IF (divider == 0)` 会被替换为:

```
1 do{
2     if(divider == 0)
3         fprintf(stderr, "Warning" "divider == 0" "\n");
```



```
4 }while(0)
```

这样，每次 **divider**(除数)为 0 时便会在标准错误流上输出一个提示信息。

注意**#**宏对空格的处理：

- 忽略传入参数名前面和后面的空格。如 `str= example1( abc )` 会被扩展成 `str="abc"`。
- 当传入参数名间存在空格时，编译器会自动连接各个子字符串，每个子字符串间只以一个空格连接。如 `str= example1( abc def)` 会被扩展成 `str="abc def"`。

### 2.3.2.2 符号连接操作符##

**##**称为连接符(concatenator 或 token-pasting)，用来将两个 Token 连接为一个 Token。注意这里连接的对象是 Token 就行，而不一定是宏的变量。例如：

```
1 #define PASTER(n)      printf( "token" #n " = %d", token##n)
2 int token9 = 9;
```

则运行 **PASTER(9)**后输出结果为 `token9 = 9`。

又如要做一个菜单项命令名和函数指针组成的结构体数组，并希望在函数名和菜单项命令名之间有直观的、名字上的关系。那么下面的代码就非常实用：

```
1 struct command{
2     char * name;
3     void (*function)(void);
4 };
5 #define COMMAND(NAME)    {NAME, NAME##_command}
```

然后，就可用一些预先定义好的命令来方便地初始化一个 **command** 结构的数组：

```
1 struct command commands[] = {
2     COMMAND(quit),
3     COMMAND(help),
4     //...
5 }
```

**COMMAND** 宏在此充当一个代码生成器的作用，这样可在一定程度上减少代码密度，间接地也可减少不留心所造成的错误。

还可以用 **n** 个**##**符号连接 **n+1** 个 Token，这个特性是**#**符号所不具备的。如：

```
1 #define LINK_MULTIPLE(a, b, c, d)    a##_##b##_##c##_##d
2 typedef struct record_type LINK_MULTIPLE(name, company, position, salary);
```

这里这个语句将展开为 `typedef struct record_type name_company_position_salary`。

注意：

- 当用**##**连接形参时，**##**前后的空格可有可无。
- 连接后的实际参数名，必须为实际存在的参数名或是编译器已知的宏定义。
- 凡是宏定义里有用'**#**'或'**##**'的地方，宏参数是不会再展开。如：

```
1 #define STR(s)      #s
2 #define CONS(a, b)  int(a##e##b)
```

则 `printf("int max: %s\n", STR(INT_MAX))` 会被展开为 `printf("int max: %s\n", "INT_MAX")`。其中，变量 `INT_MAX` 为 `int` 型的最大值，其值定义在 `<limits.h>` 中。`printf("%s\n", CONS(A, A))` 会被展开为 `printf("%s\n", int(AeA))`，从而编译报错。

`INT_MAX` 和 `A` 都不会再被展开，多加一层中间转换宏即可解决这个问题。加这层宏是为了把所有宏的参数在这层里全部展开，那么在转换宏里的那一个宏(如 `_STR`)就能得到正确的宏参数。

```
1 #define _STR(s)      #s
2 #define STR(s)      _STR(s)      // 转换宏
3 #define _CONS(a, b)  int(a##e##b)
4 #define CONS(a, b)  _CONS(a, b)  // 转换宏
```

则 `printf("int max: %s\n", STR(INT_MAX))` 输出为 `int max: 0x7fffffff`；而 `printf("%d\n", CONS(A, A))` 输出为 `200`。

这种分层展开的技术称为宏的 **Argument Prescan**，参见附录 6.1。

### 【'#'和'##'的一些应用特例】

#### 1. 合并匿名变量名

```
1 #define __ANONYMOUS1(type, var, line)  type  var##line
2 #define __ANONYMOUS0(type, line)      __ANONYMOUS1(type, _anonymous, line)
3 #define ANONYMOUS(type)                __ANONYMOUS0(type, __LINE__)
```

例：`ANONYMOUS(static int)` 即 `static int _anonymous70`，70 表示该行行号。

第一层：`ANONYMOUS(static int) → __ANONYMOUS0(static int, __LINE__)`

第二层：`→ __ANONYMOUS1(static int, _anonymous, 70)`

第三层：`→ static int _anonymous70`

即每次只能解开当前层的宏，所以 `__LINE__` 在第二层才能被解开。

#### 2. 填充结构

```
1 #define FILL(a)      {a, #a}
2
3 enum IDD{OPEN, CLOSE};
4 typedef struct{
5     IDD id;
6     const char * msg;
7 } T_MSG;
```

则 `T_MSG tMsg[ ] = {FILL(OPEN), FILL(CLOSE)}` 相当于：

```
1 T_MSG tMsg[] = {{OPEN, "OPEN"},
2                {CLOSE, "CLOSE"}};
```

3. 记录文件名

```
1 #define _GET_FILE_NAME(f)      #f
2 #define GET_FILE_NAME(f)      _GET_FILE_NAME(f)
3 static char FILE_NAME[] = GET_FILE_NAME(__FILE__);
```

4. 得到一个数值类型所对应的字符串缓冲大小

```
1 #define _TYPE_BUF_SIZE(type)   sizeof #type
2 #define TYPE_BUF_SIZE(type)   _TYPE_BUF_SIZE(type)
3 char buf[TYPE_BUF_SIZE(INT_MAX)];
4     //--> char buf[_TYPE_BUF_SIZE(0xffffffff)];
5     //--> char buf[sizeof "0xffffffff"];
```

这里相当于: `char buf[11];`

### 2.3.2.3 字符化操作符@#

@#称为字符化操作符(charizing)，只能用于有传入参数的宏定义中，且必须置于宏定义体的参数名前。作用是将传入的单字符参数名转换成字符，以一对单引号括起来。

```
1 #define makechar(x)    #@x
2 a = makechar(b);
```

展开后变成 `a= 'b'`。

### 2.3.2.4 可变参数宏

...在 C 语言宏中称为 Variadic Macro，即变参宏。C99 编译器标准允许定义可变参数宏(Macros with a Variable Number of Arguments)，这样就可以使用拥有可变参数表的宏。

可变参数宏的一般形式为：

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

省略号代表一个可以变化的参数表，变参必须作为参数表的最右一项出现。使用保留名\_\_VA\_ARGS\_\_ 把参数传递给宏。在调用宏时，省略号被表示成零个或多个符号(包括里面的逗号)，一直到到右括号结束为止。当被调用时，在宏体(macro body)中，那些符号序列集合将代替里面的\_\_VA\_ARGS\_\_标识符。当宏的调用展开时，实际的参数就传递给 fprintf ()。

注意：可变参数宏不被 ANSI/ISO C++ 所正式支持。因此，应当检查编译器是否支持这项技术。

在标准 C 里，不能省略可变参数，但却可以给它传递一个空的参数，这会导致编译出错。因为宏展开后，里面的字符串后面会有个多余的逗号。为了解决这个问题，GNU CPP 中做了如下扩展定义：

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

若可变参数被忽略或为空，## 操作将使编译器删除它前面多余的逗号(否则会编译出错)。若宏调用时提供了可变参数，编译器会把这些可变参数放到逗号的后面。

同时，GCC 还支持显式地命名变参为 args，如同其它参数一样。如下格式的宏扩展：

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

这样写可读性更强，并且更容易进行描述。

用 GCC 和 C99 的可变参数宏，可以更方便地打印调试信息，如：

```
1 #ifdef DEBUG  
2     #define DBGPRINT(format, args...) \  
3         fprintf(stderr, format, ##args)  
4 #else  
5     #define DBGPRINT(format, args...) \  
6 #endif
```

这样定义之后，代码中就可以用 `dbgprint` 了，例如 `dbgprint("aaa [%s]", __FILE__)`。

结合第 4 节的“条件编译”功能，可以构造出如下调试打印宏：



```
1 #ifdef LOG_TEST_DEBUG  
2     /* OMCI 调试日志宏 */  
3     //以 10 进制格式日志整型变量  
4     #define PRINT_DEC(x)          printf("#x" = %d\n", x)  
5     #define PRINT_DEC2(x, y)      printf("#x" = %d\n", y)  
6     //以 16 进制格式日志整型变量  
7     #define PRINT_HEX(x)          printf("#x" = 0x%-X\n", x)  
8     #define PRINT_HEX2(x, y)      printf("#x" = 0x%-X\n", y)  
9     //以字符串格式日志字符串变量  
10    #define PRINT_STR(x)           printf("#x" = %s\n", x)  
11    #define PRINT_STR2(x, y)        printf("#x" = %s\n", y)  
12  
13    //日志提示信息  
14    #define PROMPT(info)            printf("%s\n", info)  
15  
16    //调试定位信息打印宏  
17    #define TP                      printf("%-4u - [%s<%s>]\n", __LINE__, __FILE__,  
__FUNCTION__);  
18  
19    //调试跟踪宏，在待日志信息前附加日志文件名、行数、函数名等信息  
20    #define TRACE(fmt, args...) \  
21    do{\  
22        printf("[%s (%d)<%s>]", __FILE__, __LINE__, __FUNCTION__);\  
23        printf((fmt), ##args);\  
24    }while(0)  
25 #else  
26     #define PRINT_DEC(x) \  
27     #define PRINT_DEC2(x, y) \  
28  
29     #define PRINT_HEX(x) \  

```

```
30     #define PRINT_HEX2(x, y)
31
32     #define PRINT_STR(x)
33     #define PRINT_STR2(x, y)
34
35     #define PROMPT(info)
36
37     #define TP
38
39     #define TRACE(fmt, args...)
40 #endif
```



### 三 文件包含

文件包含命令的一般形式为：

```
#include "文件名"
```

通常，该文件是后缀名为"h"或"hpp"的头文件。文件包含命令把指定头文件插入该命令行位置取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。

在程序设计中，文件包含是很有用的。一个大程序可以分为多个模块，由多个程序员分别编程。有些公用的符号常量或宏定义等可单独组成一个文件，在其它文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都去书写那些公用量，从而节省时间，并减少出错。

对文件包含命令要说明以下几点：

- 包含命令中的文件名可用双引号括起来，也可用尖括号括起来，如**#include "common.h"**和**#include <math.h>**。但这两种形式是有区别的：使用尖括号表示在包含文件目录中去查找(包含目录是由用户在设置环境时设置的**include**目录)，而不在当前源文件目录去查找；使用双引号则表示首先在当前源文件目录中查找，若未找到才到包含目录中去查找。用户编程时可根据自己文件所在的目录来选择某一种命令形式。
- 一个**include**命令只能指定一个被包含文件，若有多个文件要包含，则需用多个**include**命令。
- 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

### 四 条件编译

一般情况下，源程序中所有的行都参加编译。但有时希望对其中一部分内容只在满足一定条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

条件编译功能可按不同的条件去编译不同的程序部分，从而产生不同的目标代码文件。这对于程序的移植和调试是很有用的。

条件编译有三种形式，下面分别介绍。

## 4.1 #ifdef 形式


```
#ifdef 标识符 (或#if defined 标识符)
    程序段 1
#else
    程序段 2
#endif
```

如果标识符已被 **#define** 命令定义过，则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2(它为空)，**#else** 可以没有，即可以写为：


```
#ifdef 标识符 (或#if defined 标识符)
    程序段
#endif
```

这里的“程序段”可以是语句组，也可以是命令行。这种条件编译可以提高 C 源程序的通用性。

### 【例 6】



```
1 #define NUM OK
2 int main(void) {
3     struct stu {
4         int num;
5         char *name;
6         char sex;
7         float score;
8     } *ps;
9     ps = (struct stu *) malloc(sizeof(struct stu));
10    ps->num = 102;
11    ps->name = "Zhang ping";
12    ps->sex = 'M';
13    ps->score = 62.5;
14    #ifdef NUM
15        printf("Number=%d\nScore=%f\n", ps->num, ps->score); /*--Execute--*/
16    #else
17        printf("Name=%s\nSex=%c\n", ps->name, ps->sex);
18    #endif
19    free(ps);
20    return 0;
21 }
```



由于在程序中插入了条件编译预处理命令，因此要根据 **NUM** 是否被定义过来决定编译哪个 **printf** 语句。而程序首行已对 **NUM** 作过宏定义，因此应对第一个 **printf** 语句作编译，故运行结果是输出了学号和成绩。

程序首行定义 **NUM** 为字符串“OK”，其实可为任何字符串，甚至不给出任何字符串，即 **#define NUM** 也具有同样的意义。只有取消程序首行宏定义才会去编译第二个 **printf** 语句。

## 4.2 #ifndef 形式

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

如果标识符未被 `#define` 命令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译。这与 `#ifdef` 形式的功能正相反。


“`#ifndef 标识符`”也可写为“`#if !(defined 标识符)`”。

## 4.3 #if 形式


```
#if 常量表达式
    程序段 1
#else
    程序段 2
#endif
```

如果常量表达式的值为真(非 0)，则对程序段 1 进行编译，否则对程序段 2 进行编译。因此可使程序在不同条件下，完成不同的功能。

**【例 7】**输入一行字母字符，根据需要设置条件编译，使之能将字母全改为大写或小写字母输出。



```
1 #define CAPITAL_LETTER 1
2 int main(void) {
3     char szOrig[] = "C Language", cChar;
4     int dwIdx = 0;
5     while((cChar = szOrig[dwIdx++]) != '\0')
6     {
7     #if CAPITAL_LETTER
8         if((cChar >= 'a') && (cChar <= 'z')) cChar = cChar - 0x20;
9     #else
10        if((cChar >= 'A') && (cChar <= 'Z')) cChar = cChar + 0x20;
11    #endif
12        printf("%c", cChar);
13    }
14    return 0;
15 }
```



在程序第一行定义宏 `CAPITAL_LETTER` 为 1，因此在条件编译时常量表达式 `CAPITAL_LETTER` 的值为真(非零)，故运行后使小写字母变成大写(C LANGUAGE)。

本例的条件编译当然也可以用 `if` 条件语句来实现。但是用条件语句将会对整个源程序进行编译，生成的目标代码程序很长；而采用条件编译，则根据条件只编译其中的程序段 1 或程序段 2，生成的目标程序较短。如果条件编译的程序段很长，采用条件编译的方法是十分必要的。

## 4.4 实践用例

## 1. 屏蔽跨平台差异

在大规模开发过程中，特别是跨平台和系统的软件里，可以在编译时通过条件编译设置编译环境。

例如，有一个数据类型，在 **Windows** 平台中应使用 **long** 类型表示，而在其他平台应使用 **float** 表示。这样往往需要对源程序作必要的修改，这就降低了程序的通用性。可以用以下的条件编译：

```
1 #ifdef WINDOWS
2     #define MYTYPE long
3 #else
4     #define MYTYPE float
5 #endif
```

如果在 **Windows** 上编译程序，则可以在程序的开始加上 **#define WINDOWS**，这样就编译命令行 **#define MYTYPE long**；如果在这组条件编译命令前曾出现命令行 **#define WINDOWS 0**，则预编译后程序中的 **MYTYPE** 都用 **float** 代替。这样，源程序可以不必作任何修改就可以用于不同类型的计算机系统。

## 2. 包含程序功能模块

例如，在程序首部定义 **#ifdef FLV**：

```
1 #ifdef FLV
2     include "fastleave.c"
3 #endif
```

如果不许向别的用户提供该功能，则在编译之前将首部的 **FLV** 加一下划线即可。

## 3. 开关调试信息

调试程序时，常常希望输出一些所需的信息以便追踪程序的运行。而在调试完成后不再输出这些信息。可以在源程序中插入以下的条件编译段：

```
1 #ifdef DEBUG
2     printf("device_open(%p)\n", file);
3 #endif
```

如果在它的前面有以下命令行 **#define DEBUG**，则在程序运行时输出 **file** 指针的值，以便调试分析。调试完成后只需将这个 **define** 命令行删除即可，这时所有使用 **DEBUG** 作标识符的条件编译段中的 **printf** 语句不起作用，即起到“开关”一样统一控制的作用。

## 4. 避开硬件的限制。

有时一些具体应用环境的硬件不同，但限于条件本地缺乏这种设备，可绕过硬件直接写出预期结果：

```
1 #ifndef TEST
2     i = dial(); //程序调试运行时绕过此语句
3 #else
4     i = 0;
5 #endif
```

调试通过后，再屏蔽 **TEST** 的定义并重新编译即可。

## 5. 防止头文件重复包含

头文件(.h)可以被头文件或 C 文件包含。由于头文件包含可以嵌套，C 文件就有可能多次包含同一个头文件；或者不同的 C 文件都包含同一个头文件，编译时就可能出现重复包含(重复定义)的问题。



在头文件中为了避免重复调用(如两个头文件互相包含对方)，常采用这样的结构：

```
1 #ifndef <标识符>
2     #define <标识符>
3     //真正的内容，如函数声明之类
4 #endif
```

<标识符>可以自由命名，但一般形如\_\_HEADER\_H，且每个头文件标识都应该是唯一的。

事实上，不管头文件会不会被多个文件引用，都要加上条件编译开关来避免重复包含。

6. 在#ifndef 中定义变量出现的问题(一般不定义在#ifndef 中)。

```
1 #ifndef PRECMPL
2     #define PRECMPL
3     int var;
4 #endif
```

其中有个变量定义，在 VC 中链接时会出现变量 var 重复定义的错误，而在 C 中成功编译。

(1) 当第一个使用这个头文件的.cpp 文件生成.obj 时，var 在里面定义；当另一个使用该头文件的.cpp 文件再次(单独)生成.obj 时，var 又被定义；然后两个 obj 被第三个包含该头文件.cpp 连接在一起，会出现重复定义。


(2) 把源程序文件扩展名改成.c 后，VC 按照 C 语言语法对源程序进行编译。在 C 语言中，遇到多个 int var 则自动认为其中一个是定义，其他的是声明。

(3) C 语言和 C++ 语言连接结果不同，可能是在进行编译时，C++ 语言将全局变量默认为强符号，所以连接出错。C 语言则依照是否初始化进行强弱的判断的(仅供参考)。


解决方法：

(1) 把源程序文件扩展名改成.c。

(2) .h 中只声明 extern int var;，在.cpp 中定义(推荐)



```
1 //<x.h>
2 #ifndef __X_H
3     #define __X_H
4     extern int var;
5 #endif
6 <x.c>
7 int var = 0;
```



综上，变量一般不要定义在.h 文件中。

## 五 小结

1. 预处理功能是 C 语言特有的功能，它是在对源程序正式编译前由预处理程序完成的。程序员在程序中用预处理命令来调用这些功能。

- 宏定义是用一个标识符来表示一个字符串，这个字符串可以是常量、变量或表达式。在宏调用中将用该字符串代换宏名。
- 宏定义可以带有参数，宏调用时是以实参代换形参。而不是“值传递”。
- 为了避免宏替换时发生错误，宏定义中的字符串应加括号，字符串中出现的形式参数两边也应加括号。
- 文件包含是预处理的一个重要功能，它可用来把多个源文件连接成一个源文件进行编译，结果将生成一个目标文件。
- 条件编译允许只编译源程序中满足条件的程序段，使生成的目标程序较短，从而减少了内存的开销并提高了程序的效率。
- 使用预处理功能便于程序的修改、阅读、移植和调试，也便于实现模块化程序设计。

## 六 附录

### 6.1 Argument Prescan

(摘自 <http://gcc.gnu.org/onlinedocs/cpp/Argument-Prescan.html>)

Macro arguments are completely macro-expanded before they are substituted into a macro body, unless they are stringified or pasted with other tokens. After substitution, the entire macro body, including the substituted arguments, is scanned again for macros to be expanded. The result is that the arguments are scanned *twice* to expand macro calls in them.

宏参数被完全展开后再替换入宏体，但当宏参数被字符串化(`#`)或与其它子串连接(`##`)时不予展开。在替换之后，再次扫描整个宏体(包括已替换宏参数)以进一步展开宏。结果是宏参数被扫描两次以展开参数所(嵌套)调用的宏。

若带参数宏定义中的参数称为形参，调用宏时的实际参数称为实参，则宏的展开可用以下三步来简单描述(该步骤与 gcc 摘录稍有不同，但更易操作)：

- 1) 用实参替换形参，将实参代入宏文本中；
- 2) 若实参也是宏，则展开实参；
- 3) 继续处理宏替换后的宏文本，若宏文本也包含宏则继续展开，否则完成展开。

其中第一步将实参代入宏文本后，若实参前遇到字符`"#"`或`"##"`，即使实参是宏也不再展开实参，而当作文本处理。

上述展开步骤示例如下：

```
1 #define TO_STRING(x)    _TO_STRING(x)
2 #define _TO_STRING(x)  #x
3 #define FOO            4
```

则`_TO_STRING(FOO)`展开为`"FOO"`；`TO_STRING(FOO)`展开为`_TO_STRING(4)`，进而展开为`"4"`。相当于借助`_TO_STRING`这样的中间宏，先展开宏参数，延迟其字符化。

### 6.2 宏的其他注意事项

1. 避免在无作用域限定(未用`{}`括起)的宏内定义数组、结构、字符串等变量，否则函数中对宏的多次引用会导致实际局部变量空间成倍放大。

2. 按照宏的功能、模块进行集中定义。即在一处将常量数值定义为宏，其他地方通过引用该宏，生成自己模块的宏。严禁相同含义的常量数值，在不同地方定义为不同的宏，即使数值相同也不允许(维护修改后极易遗漏，造成代码隐患)。

3. 用只读变量适当替代(类似功能的)宏，例如将 `#define PIE 3.14` 改为 `const float PIE = 3.14`。这样做的好处如下：

- 1) 预编译时用宏定义值替换宏名，编译时报错不易理解；
- 2) 跟踪调试时显示宏值，而不是宏名；
- 3) 宏没有类型，不能做类型检查，不安全；
- 4) 宏自身没有作用域；
- 5) 只读变量和宏的效率同样高。

注意，C 语言中只读变量不可用于数组大小、变量(包括数组元素)初始化值以及 `case` 表达式。

4. 用 `inline` 函数代替(类似功能的)宏函数。好处如下：

- 1) 宏函数在预编译时处理，编译出错信息不易理解；
- 2) 宏函数本身无法单步跟踪调试，因此也不要再在宏内调用函数。但某些编译器(为了调试需要)可将 `inline` 函数转成普通函数；

- 3) 宏函数的入参没有类型，不安全；

- 5) `inline` 函数会在目标代码中展开，和宏的效率一样高；

注意，某些宏函数用法独特，不能用 `inline` 函数取代。当不想或不能指明参数类型时，宏函数更合适。

5. 不带参数的宏函数也要定义成函数形式，如 `#define HELLO( ) printf("Hello.")`。

括号会暗示阅读代码者该宏是一个函数。

6. 带参宏内定义变量时，应注意避免内外部变量重名的问题：

```
1 typedef struct{
2     int d;
3 }T_TEST;
4 T_TEST gtTest = {0};
5 #define ASSIGN1(_d) do{ \
6     T_TEST t = {0}; \
7     t.d = _d; \
8     gtTest = t; \
9 }while(0)
10
11 #define ASSIGN2(_p) do{ \
12     int _d; \
13     _d = 5; \
14     (_p) = _d; \
15 }while(0)
```

若宏参数名或宏内变量名不加前缀下划线，则 `ASSIGN1(c)`将会导致编译报错(`t.d` 被替换为 `t.c`)，`ASSIGN2(d)` 会因宏内作用域而导致外部的变量 `d` 值保持不变(而非改为 `5`)。

7. 不要用宏改写语言。例如：

```
1 #define FOREVER    for ( ; ; )
2 #define BEGIN      {
3 #define END        }
```

C 语言有完善且众所周知的语法。试图将其改变成类似于其他语言的形式，会使读者混淆，难于理解。

### 6.3 do{...}while(0)妙用

1. 函数中使用 `do{...}while(0)`可替代 `goto` 语句。例如：

goto 写法	替代写法
<pre>bOk = func1(); if(!bOk) goto errorhandle; bOk = func2(); if(!bOk) goto errorhandle; bOk = func3(); if(!bOk) goto errorhandle;  //... ... //执行成功，释放资源并返回 delete p; p = NULL; return true;  errorhandle: delete p; p = NULL; return false;</pre>	<pre>do{     //执行并进行错误处理     bOk = func1();     if(!bOk) break;     bOk = func2();     if(!bOk) break;     bOk = func3();     if(!bOk) break;      // ..... }while(0);  //释放资源 delete p; p = NULL; return bOk;</pre>

2. 宏定义中使用 `do{...}while(0)`的原因及好处：

- 1) 避免空的宏定义产生 `warning`，如`#define DUMMY( ) do{}while(0)`。
- 2) 存在一个独立的代码块，可进行变量定义，实现比较复杂的逻辑处理。

注意，该代码块内(即`{...}`内)定义的变量其作用域仅限于该块。此外，为避免宏的实参与其内部定义的变量同名而造成覆盖，最好在变量名前加上`_`(基于如下编程惯例：除非是库，否则不应定义以`_`开始的变量)。

3) 若宏出现在判断语句之后，可保证作为一个整体来实现。

如 `#define SAFE_DELETE(p) delete p; p = NULL;`，则以下代码

```
1 if(NULL != p)
2     SAFE_DELETE(p)
3 else
4     DUMMY( );
```

就有两个问题：

a) 因为 `if` 分支后有两条语句，`else` 分支没有对应的 `if`，编译失败；

b) 假设没有 `else`，则 `SAFE_DELETE` 中第二条语句无论 `if` 判断是否成立均会执行，这显然违背程序设计的原始目的。

那么，为了避免这两个问题，将宏直接用 `{}` 括起来是否可以？如：

`#define SAFE_DELETE(p) {delete p; p = NULL;}`

的确，上述问题不复存在。但 C/C++ 编程中，在每条语句后加分号是约定俗成的习惯，此时以下代码

```
1 if(NULL != p)
2     SAFE_DELETE(p);
3 else
4     DUMMY( );
```

其 `else` 分支就无法通过编译(多出一个分号)，而采用 `do{...}while(0)` 则毫无问题。

使用 `do{...} while(0)` 将宏包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数编译器都能够识别 `do{...}while(0)` 这种无用的循环并优化，所以该法不会导致程序的性能降低。

## 6.4 类型定义符 typedef

C 语言不仅提供了丰富的数据类型，而且还允许由用户自己定义类型说明符，也就是说允许由用户为数据类型取“别名”。类型定义符 `typedef` 即可用来完成此功能。

`typedef` 定义的一般形式为：

<code>typedef</code>	原类型
名	新类型名

其中原类型名中含有定义部分，新类型名一般用大写表示，以便于区别。

例如，有整型量 `int a,b`。其中 `int` 是整型变量的类型说明符。`int` 的完整写法为 `integer`，为增加程序的可读性，可把整型说明符用 `typedef` 定义为 `typedef int INTEGER`。此后就可用 `INTEGER` 来代替 `int` 作整型变量的类型说明，如 `INTEGER a,b` 等效于 `int a,b`。

用 `typedef` 定义数组、指针、结构等类型将带来很大的方便，不仅使程序书写简单而且意义更为明确，因而增强了可读性。

例如，`typedef char NAME[20]` 表示 `NAME` 是字符数组类型，数组长度为 20。然后可用 `NAME` 说明变量，如 `NAME a1,a2,s1,s2` 完全等效于：`char a1[20],a2[20],s1[20],s2[20]`。

又如：

```
1 typedef struct {
```

```

2     char name[20];
3     int age;
4     char sex;
5 }STU;

```

然后可用 **STU** 来定义结构变量: **STU body1,body2;**

有时也可用宏定义来代替 **typedef** 的功能, 但是宏定义是由预处理完成的, 而 **typedef** 则是在编译时完成的, 后者更为灵活方便。

此外, 采用 **typedef** 重新定义一些类型, 可防止因平台和编译器不同而产生的类型字节数差异, 方便移植。如:



```

1 typedef unsigned char boolean;          /* Boolean value type. */
2 typedef unsigned long int uint32;       /* Unsigned 32 bit value */
3 typedef unsigned short uint16;          /* Unsigned 16 bit value */
4 typedef unsigned char uint8;            /* Unsigned 8 bit value */
5 typedef signed long int int32;           /* Signed 32 bit value */
6 typedef signed short int16;             /* Signed 16 bit value */
7 typedef signed char int8;               /* Signed 8 bit value */
8 //下面的不建议使用
9 typedef unsigned char byte;             /* Unsigned 8 bit value type. */
10 typedef unsigned short word;            /* Unsigned 16 bit value type. */
11 typedef unsigned long dword;            /* Unsigned 32 bit value type. */
12 typedef unsigned char uint1;            /* Unsigned 8 bit value type. */
13 typedef unsigned short uint2;           /* Unsigned 16 bit value type. */
14 typedef unsigned long uint4;            /* Unsigned 32 bit value type. */
15 typedef signed char int1;               /* Signed 8 bit value type. */
16 typedef signed short int2;              /* Signed 16 bit value type. */
17 typedef long int int4;                  /* Signed 32 bit value type. */
18 typedef signed long sint31;              /* Signed 32 bit value */
19 typedef signed short sint15;             /* Signed 16 bit value */
20 typedef signed char sint7;              /* Signed 8 bit value */

```