预处理器

编译一个 C 程序设计很多步骤。其中第 1 个步骤被称为预处理阶段。C 预处理器在源代码编译之前对其进行一些文本性质的操作。他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。

预定义符号

1.预处理器定义了一些符号，他们的值或者是字符串常量，或者是十进制数字常量，**FILE** 的含义是进行编译的源文件名。

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

#define

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）是一个由都好分隔的值的列表，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。

这里由一个宏。它接受一个参数：

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

如果在上述声明之后把 `SQUARE(5)`至于程序中，预处理器就会用下面的这个表达式替换上面的表达式：

```
((5)*(5))
```

这里添加括号的原因是，为了避免在使用宏时，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。

#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，他们首先被替换。
2. 替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被他们的值所替代。
3. 最后，再次对结果文本进行扫描，看看他是否包含了任何由#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。

#argument 替换符号

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。#argument 这种结构被预处理器翻译为 "argument"，可以把一个宏参数转化为字符串。如果想把宏参数插入到字符串常量中，可以使用下面的技巧：

技巧一

首先，临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。例子如下：

```
#define PRINT(FORMAT,VALUE)  \
printf( "The value is "#FORMAT"", VALUE)
...
int x = 6;
PRINT_FOR( %d, x + 3);
```

输出结果：

```
The value is 9
```

技巧二

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

输出结果：

```
the value of x + 3 is 9
```

书上的技巧一和技巧二并不能通过编译，我经过修改的到上面的两个例子，我感觉是用了同一个技巧就是将字符串中的 "#VALUE" 替换成传入的字符，如技巧二中介

绍的将 #VALUE 替换成了字符串中的 x+3，将 #FORMAT 替换成 %d，最后一个 VALUE 作为实际的参数只是进行了简单的文本替换，不涉及字符串修改。

技巧三

```
#define PRINT(VALUE)  printf(#VALUE)
...
PRINT(hello SummerGift);
```

输出结果：

```
hello SummerGift
```

技巧三展示了基本的 #argument 功能，即将 #VALUE 替换成 "hello SummerGift"，也就是将 hello SummerGift 简单替换成了 "hello SummerGift"。

##连接符号

结构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。下面的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。

宏与函数

1. 用宏来完成比较简单的计算，比如比较两个表达式的大小的好处是，如果使用函数来实现，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任务的代码更大，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。
   `#define MAX(A, B) ((A)>(B) ? (A):(B))`
2. 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形、长整形、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。
3. 与函数相比，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。
4. 还有一些任务根本无法用函数实现。比如下面的宏定义，这个宏的第二个参数是一种类型，它无法作为函数参数进行传递。

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

实际转换过程如下：

转换前：pi = MALLOC(25, int);
转换后：pi = ((int *)malloc((25) * sizeof(int)))

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结果，例如：

```
x+1
```

这个表达式可以重复执行几百次，他每次获得的值结果都是一样的。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，他将产生一个不同的结果。MAX 宏可以证明具有副作用的参数引起的问题。

```
#define MAX(A, B) ((A)>(B) ? (A):(B))
...
x = 5;
y = 8;
z = MAX(x, y);
printf("x = %d, y = %d, z = %d", x, y, z);
```

这个宏的结果是 x = 6, y = 10, z = 9。

检查替换后的结果： z = (x++) > (y++) ? (x++) : (y++);
那个较小的值只增值了一次，而那个较大的值却增值了两次，第一次是在比较的时候，第二次是在执行 ？ 后面的表达式的时候。

命名约定

为宏采纳一种命名约定是很重要的，用来区分一个名称是一个函数还是一个宏，避免发生混淆。

#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，那么他的旧定义首先必须用#undef 移除。
如果一个宏不想让后面的程序用了，那么就将其 undef，如果后面使用了这个宏就会报错。

命令行定义

许多编译器提供了一种能力，允许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。例如有如下数组：

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

那么在 UNIX 系统中，编译这个程序的命令行很可能是下面的样子：

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

条件编译

常见的条件编译：

```
#if constant-expression
    statments
#endif
```

其中，constant-expression(常量表达式)由预处理器进行求值，如果其值为真，则 statements 部分就被正常编译，否则预处理器就安静地删除他们。

是否被定义

测试一个符号是否被定义也是可能的。在条件编译中完成这个任务往往更方便，因为程序如果并不需要控制编译的符号所控制的特性，他就不需要被定义。这个测试可以用下面的方式来进行：

```
#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(ABD) && defined(BCD)
```

嵌套指令

上面提到的这些指令可以嵌套于另一个指令内部。

```
#if   xxx
    statments
#elif xxx
    statments
#endif
```

文件包含

使用 #include 指令的替换执行方式很简单，预处理删除这条指令，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，它实际被编译了 10 次。

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。实际上，他们之间的区别很小。

函数库文件包含

函数库头文件包含使用下面的语法：

```
#include <filename>
```

这种情况的包含会在系统目录里寻找函数库头文件。也可以使用编译器选项添加自己的文件函数库。

本地文件包含

本地头文件包含使用下面的语法：

```
#include "filename"
```

这种情况的包含会先在当前目录进行查找，如果未找到再像查找函数库头文件一样在标准位置查找本地头文件。

嵌套文件包含

使用条件编译来避免重复包含头文件出现的错误：

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

其他指令

#error 指令允许生成错误信息

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

使用方法：

```
#error no option selsected
```

#line 修改下一行输入的行号

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

它通知预处理器 number 使下一行输入的行号。如果给出了可选部分 string，预处理器就把他当作当前文件的名字。值得注意的是，这条指令修改 __LINE__符号的值，如果加上可选部分，他还将修改__FILE__符号的值。

#progma 用于支持因编译器而异的特性

#progma 指令是另一种机制，用于支持因编译器而异的机制。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者其他任何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者关闭清单显示，或者把会变代码插入到 C 程序中。从本质上说，#progma 是不可移植的。预处理器将忽略他们不认识的#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。

#(null directive) 无效指令

无效指令就是一个#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。下面地例子中无效指令通过把 #include 与周围的代码分隔开来，凸显它的存在。

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

插入空行也可以取得相同的效果。

总结

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，但忘了在他们周围加上括号。
3. 忘了在整个宏定义的两边加上括号。

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 宏创建一种新的语言。
4. 采用采用命名约定，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 文件是我们很难判断源文件之间的依赖关系。