表（3.1） 预处理指令

预处理名称	意　义
#define	宏定义
#undef	撤销已定义过的宏名
#include	使编译程序将另一源文件嵌入到带有#include 的源文件中
#if	#if 的一般含义是如果#if 后面的常量表达式为 true，则编译它与#endif 之间的代码，否则跳过这些代码。命令#endif 标识一个#if 块的结束。#else 命令的功能有点象 C 语言中的 else ，#else 建立另一选择（在# if 失败的情况下）。#elif 命令意义与 else if 相同，它形成一个 if else-if 阶梯状语句，可进行多种编译选择。
#else	
#elif	
#endif	
#ifdef	用#ifdef 与#ifndef 命令分别表示“如果有定义”及“如果无定义”，是条件编译的另一种方法。
#ifndef	
#line	改变当前行数和文件名称，它们是在编译程序中预先定义的标识符 命令的基本形式如下： #line number["filename"]
#error	编译程序时，只要遇到 #error 就会生成一个编译错误提示消息，并停止编译
#pragma	为实现时定义的命令，它允许向编译程序传送各种指令例如，编译程序可能有一种选择，它支持对程序执行的跟踪。可用#pragma 语句指定一个跟踪选择。

另外 ANSI 标准 C 还定义了如下几个宏：

LINE 表示正在编译的文件的行号

FILE 表示正在编译的文件的名字

DATE 表示编译时刻的日期字符串，例如： "25 Dec 2007"

TIME 表示编译时刻的时间字符串，例如： "12:30:55"

STDC 判断该文件是不是定义成标准 C 程序

如果编译器不是标准的，则可能仅支持以上宏的一部分，或根本不支持。当然编译器也有可能还提供其它预定义的宏名。注意：宏名的书写由标识符与两边各二条下划线构成。

相信很多初学者，甚至一些有经验的程序员都没有完全掌握这些内容，下面就一一详细讨论这些预处理指令。

3.1，宏定义

3.1.1，数值宏常量

#define 宏定义是个演技非常高超的替身演员，但也会经常耍大牌的，所以我们用它要慎之又慎。它可以出现在代码的任何地方，**从本行宏定义开始**，以后的代码就都认识这个宏了；也可以把任何东西定义成宏。因为编译器会在预编译的时候用真身替换替身，而在我们的代码里面却又用常常用替身来帮忙。看例子：

#define PI 3.141592654

在此后的代码中你尽可以使用 PI 来代替 3.141592654，而且你最好就这么做。不然的话，如果我要把 PI 的精度再提高一些，你是否愿意一个一个的去修改这串数呢？你能保证不漏不出错？而使用 PI 的话，我们却只需要修改一次。这种情况还不是最要命的，我们再看一个例子：

#define ERROR_POWEROFF -1

如果你在代码里不用 ERROR_POWEROFF 这个宏而用-1，尤其在函数返回错误代码的时候（往往一个开发一个系统需要定义很多错误代码）。肯怕上帝都无法知道-1 表示的是什么意思吧。这个-1，我们一般称为“魔鬼数”，上帝遇到它也会发狂的。所以，我奉劝你代码里一定不要出现“魔鬼数”。

第一章我们详细讨论了 const 这个关键字，我们知道 const 修饰的数据是有类型的，而 define 宏定义的数据没有类型。为了安全，我建议你以后在定义一些宏常数的时候用 const 代替，编译器会给 const 修饰的只读变量做类型校验，减少错误的可能。但一定要注意 const 修饰的不是常量而是 readonly 的变量，**const 修饰的只读变量不能用来作为定义数组的维数，也不能放在 case 关键字后面。**

3.1.2，字符串宏常量

除了定义宏常数之外，经常还用来定义字符串，尤其是路径：

A),#define ENG_PATH_1 E:\English\listen_to_this\listen_to_this_3

B),#define ENG_PATH_2 “E:\English\listen_to_this\listen_to_this_3”

噢，到底哪一个正确呢？如果路径太长，一行写下来比较别扭怎么办？用反斜杠接续符啊：

C), #define ENG_PATH_3 E:\English\listen_to_this\listen\

_to_this_3

还没发现问题？这里用了 4 个反斜杠，到底哪个是接续符？回去看看接续符反斜杠。反斜杠作为接续符时，在本行其后面不能再有任何字符，空格都不行。所以，只有最后一个反斜杠才是接续符。至于 A)和 B)，那要看你怎么用了，既然 define 宏只是简单的替换，那给 ENG_PATH_1 加上双引号不就成了：“ENG_PATH_1”。

但是请注意：有的系统里规定路径的要用双反斜杠“\\”,比如：

#define ENG_PATH_4 E:\\English\\listen_to_this\\listen_to_this_3

3.1.3，用 define 宏定义注释符号？

上面对 define 的使用都很简单，再看看下面的例子：

#define BSC //

#define BMC /*

#define EMC */

D),BSC my single-line comment

E),BMC my multi-line comment EMC

D)和 E)都错误，为什么呢？因为注释先于预处理指令被处理,当这两行被展开成//…或/*…*/时,注释已处理完毕,此时再出现//…或/*…*/自然错误.因此,试图用宏开始或结束一段注释是不行的。

3.1.4，用 define 宏定义表达式

这些都好理解，下面来点有“技术含量”的：

定义一年有多少秒：

#define SEC_A_YEAR 60*60*24*365

这个定义没错吧？很遗憾，很有可能错了，至少不可靠。你有没有考虑在 16 位系统下把这样一个数赋给整型变量的时候可能会发生溢出？一年有多少秒也不可能是负数吧。修改一下：

#define SEC_A_YEAR （60*60*24*365）UL

又出现一个问题，这里的括号到底需不需要呢？继续看一个例子：

定义一个宏函数，求 x 的平方：

#define SQR (x) x * x

对不对？试试：假设 x 的值为 10，SQR (x)被替换后变成 10*10。没有问题。

再试试：假设 x 的值是个表达式 10+1，SQR (x)被替换后变成 10+1*10+1。问题来了，这并不是我想要得到的。怎么办？括号括起来不就完了？

#define SQR (x) ((x) * (x))

最外层的括号最好也别省了，看例子：

求两个数的和：

#define SUM (x) (x) + (x)

如果 x 的值是个表达式 5*3,而代码又写成这样：SUM (x)* SUM (x)。替换后变成：(5*3)+(5*3) * (5*3) + (5*3)。又错了！所以最外层的括号最好也别省了。我说过 define 是个演技高超的替身演员，但也经常耍大牌。**要搞定它其实很简单，别吝啬括号就行了**。

注意这一点：宏函数被调用时是以实参代换形参。而不是“值传送”。

3.1.5，宏定义中的空格

另外还有一个问题需要引起注意，看下面例子：

#define SUM （x） （x）+（x）

这还是定义的宏函数 SUM（x）吗？显然不是。编译器认为这是定义了一个宏：SUM，其代表的是（x） （x）+（x）。为什么会这样呢？其关键问题还是在于 SUM 后面的这个空格。所以在定义宏的时候一定要注意什么时候该用空格，什么时候不该用空格。这个空格仅仅在定义的时候有效，在使用这个宏函数的时候，空格会被编译器忽略掉。也就是说，上一节定义好的宏函数 SUM（x）在使用的时候在 SUM 和（x）之间留有空格是没问题的。比如：SUM（3）和 SUM （3）的意思是一样的。

3.1.6，#undef

#undef 是用来撤销宏定义的，用法如下：

```
#define PI 3.141592654
…
// code


#undef PI
//下面的代码就不能用 PI 了，它已经被撤销了宏定义。
```

也就是说宏的生命周期从#define 开始到#undef 结束。很简单，但是请思考一下这个问题：

```
#define X 3
#define Y X*2
#undef X
#define X 2
int z=Y;
```

z 的值为多少？

3.2，条件编译

条件编译的功能使得我们可以按不同的条件去编译不同的程序部分，因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：

第一种形式：

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

它的功能是，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)，本格式中的#else 可以没有，即可以写为：

```
#ifdef 标识符
程序段

#endif
```

第二种形式：

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

与第一种形式的区别是将“ifdef”改为“ifndef”。它的功能是，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。这与第一种形式的功能正相反。

第三种形式：

```
#if 常量表达式
程序段 1



#else
程序段 2
#endif
```

它的功能是，如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。因此可以使程序在不同条件下，完成不同的功能。

至于#elif 命令意义与 else if 相同，它形成一个 if else-if 阶梯状语句，可进行多种编译选择。

3.3，文件包含

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。C 语言提供#include 命令来实现文件包含的操作，它实际是宏替换的延伸，有两种格式：

格式 1：

```
#include <filename>
```

其中，filename 为要包含的文件名称，用尖括号括起来，也称为头文件，表示预处理到系统规定的路径中去获得这个文件（即 C 编译系统所提供的并存放在指定的子目录下的头文件）。找到文件后，用文件内容替换该语句。

格式 2：

```
#include “filename”
```

其中，filename 为要包含的文件名称。双引号表示预处理应在当前目录中查找文件名为 filename 的文件，若没有找到，则按系统指定的路径信息，搜索其他目录。找到文件后，用文件内容替换该语句。

需要强调的一点是：#include 是将已存在文件的内容嵌入到当前文件中。

另外关于#include 的路径也有点要说明：include 支持相对路径，格式如 trackant(蚁迹寻踪)所写：

.代表当前目录，..代表上层目录。

3.4，#error 预处理

#error 预处理指令的作用是，编译程序时，只要遇到 #error 就会生成一个编译错误提示消息，并停止编译。其语法格式为：

```
#error error-message
```

注意，宏串 error-message 不用双引号包围。遇到#error 指令时，错误信息被显示，可能同时还显示编译程序作者预先定义的其他内容。关于系统所支持的 error-message 信息，请查找相关资料，这里不浪费篇幅来做讨论。

3.5，#line 预处理

#line 的作用是改变当前行数和文件名称，它们是在编译程序中预先定义的标识符

命令的基本形式如下：

#line number["filename"]

其中[]内的文件名可以省略。

例如：

#line 30 a.h

其中，文件名 a.h 可以省略不写。
这条指令可以改变当前的行号和文件名，例如上面的这条预处理指令就可以改变当前的行号为 30，文件名是 a.h。初看起来似乎没有什么用，不过，他还是有点用的，那就是用在编译器的编写中，我们知道编译器对 C 源码编译过程中会产生一些中间文件，通过这条指令，可以保证文件名是固定的，不会被这些中间文件代替，有利于进行分析。

3.6，#pragma 预处理

在所有的预处理指令中，#pragma 指令可能是最复杂的了，它的作用是设定编译器的状态或者是指示编译器完成一些特定的动作。#pragma 指令对每个编译器给出了一个方法,在保持与 C 和 C ++语言完全兼容的情况下,给出主机或操作系统专有的特征。依据定义,编译指示是机器或操作系统专有的,且对于每个编译器都是不同的。
其格式一般为:

#pragma para

其中 para 为参数，下面来看一些常用的参数。

3.6.1，#pragma message

message 参数：Message 参数是我最喜欢的一个参数，它能够在编译信息输出窗口中输出相应的信息，这对于源代码信息的控制是非常重要的。其使用方法为：

#pragma message("消息文本")

当编译器遇到这条指令时就在编译输出窗口中将消息文本打印出来。
当我们在程序中定义了许多宏来控制源代码版本的时候，我们自己有可能都会忘记有没有正确的设置这些宏，此时我们可以用这条指令在编译的时候就进行检查。假设我们希望判断自己有没有在源代码的什么地方定义了_X86 这个宏可以用下面的方法

```
#ifdef _X86
#Pragma message("_X86 macro activated!")
#endif
```

当我们定义了_X86 这个宏以后，应用程序在编译时就会在编译输出窗口里显示"_X86 macro activated!"。我们就不会因为不记得自己定义的一些特定的宏而抓耳挠腮了

3.6.2，#pragma code_seg

另一个使用得比较多的 pragma 参数是 code_seg。格式如：

#pragma code_seg(["section-name"[,"section-class"]])

它能够设置程序中函数代码存放的代码段，当我们开发驱动程序的时候就会使用到它。

3.6.3，#pragma once

#pragma once (比较常用）

只要在头文件的最开始加入这条指令就能够保证头文件被编译一次，这条指令实际上在 Visual C++6.0 中就已经有了，但是考虑到兼容性并没有太多的使用它。

3.6.4，#pragma hdrstop

#pragma hdrstop 表示预编译头文件到此为止，后面的头文件不进行预编译。BCB 可以预编译头文件以加快链接的速度，但如果所有头文件都进行预编译又可能占太多磁盘空间，所以使用这个选项排除一些头文件。

有时单元之间有依赖关系，比如单元 A 依赖单元 B，所以单元 B 要先于单元 A 编译。你可以用#pragma startup 指定编译优先级，如果使用了#pragma package(smart_init) ，BCB 就会根据优先级的大小先后编译。

3.6.5，#pragma resource

#pragma resource "*.dfm"表示把*.dfm 文件中的资源加入工程。*.dfm 中包括窗体外观的定义。

3.6.6，#pragma warning

#pragma warning(disable : 4507 34; once : 4385; error : 164)

等价于：

#pragma warning(disable:4507 34) // 不显示 4507 和 34 号警告信息

#pragma warning(once:4385) // 4385 号警告信息仅报告一次

#pragma warning(error:164) // 把 164 号警告信息作为一个错误。

同时这个 pragma warning 也支持如下格式：

#pragma warning(push [,n])

#pragma warning(pop)

这里 n 代表一个警告等级(1---4)。

#pragma warning(push)保存所有警告信息的现有的警告状态。

#pragma warning(push, n)保存所有警告信息的现有的警告状态，并且把全局警告等级设定为 n。

#pragma warning(pop)向栈中弹出最后一个警告信息，在入栈和出栈之间所作的一切改动取消。例如：

#pragma warning(push)

#pragma warning(disable : 4705)

```
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
//.....
#pragma warning( pop )
```

在这段代码的最后，重新保存所有的警告信息(包括 4705，4706 和 4707)。

3.6.7，#pragma comment

```
#pragma comment(...)
```

该指令将一个注释记录放入一个对象文件或可执行文件中。

常用的 lib 关键字，可以帮我们连入一个库文件。 比如：

```
#pragma comment(lib, "user32.lib")
```

该指令用来将 user32.lib 库文件加入到本工程中。

linker:将一个链接选项放入目标文件中,你可以使用这个指令来代替由命令行传入的或者在开发环境中设置的链接选项,你可以指定/include 选项来强制包含某个对象,例如:

```
#pragma comment(linker, "/include:__mySymbol")
```

3.6.8，#pragma pack

这里重点讨论内存对齐的问题和#pragma pack（）的使用方法。

什么是内存对齐？

先看下面的结构：

```
struct TestStruct1
{
    char c1;
    short s;
    char c2;
    int i;
};
```

假设这个结构的成员在内存中是紧凑排列的，假设 c1 的地址是 0，那么 s 的地址就应该是 1，c2 的地址就是 3，i 的地址就是 4。也就是 c1 地址为 00000000, s 地址为 00000001, c2 地址为 00000003, i 地址为 00000004。

可是，我们在 Visual C++6.0 中写一个简单的程序：

```
struct TestStruct1 a;
printf("c1 %p, s %p, c2 %p, i %p\n",
   (unsigned int)(void*)&a.c1 - (unsigned int)(void*)&a,
   (unsigned int)(void*)&a.s - (unsigned int)(void*)&a,
   (unsigned int)(void*)&a.c2 - (unsigned int)(void*)&a,
   (unsigned int)(void*)&a.i - (unsigned int)(void*)&a);
```

运行，输出：

c1 00000000, s 00000002, c2 00000004, i 00000008。

为什么会这样？这就是内存对齐而导致的问题。

3.6.8.1，为什么会有内存对齐？

字，双字，和四字在自然边界上不需要在内存中对齐。（对字，双字，和四字来说，自然边界分别是偶数地址，可以被 4 整除的地址，和可以被 8 整除的地址。）无论如何，为了提高程序的性能，数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；然而，对齐的内存访问仅需要一次访问。

一个字或双字操作数跨越了 4 字节边界，或者一个四字操作数跨越了 8 字节边界，被认为是未对齐的，从而需要两次总线周期来访问内存。一个字起始地址是奇数但却没有跨越字边界被认为是对齐的，能够在一个总线周期中被访问。某些操作双四字的指令需要内存操作数在自然边界上对齐。如果操作数没有对齐，这些指令将会产生一个通用保护异常。双四字的自然边界是能够被 16 整除的地址。其他的操作双四字的指令允许未对齐的访问（不会产生通用保护异常），然而，需要额外的内存总线周期来访问内存中未对齐的数据。

缺省情况下，编译器默认将结构、栈中的成员数据进行内存对齐。因此，上面的程序输出就变成了：c1 00000000, s 00000002, c2 00000004, i 00000008。编译器将未对齐的成员向后移，将每一个都成员对齐到自然边界上，从而也导致了整个结构的尺寸变大。尽管会牺牲一点空间（成员之间有部分内存空闲），但提高了性能。也正是这个原因，我们不可以断言 sizeof(TestStruct1)的结果为 8。在这个例子中，sizeof(TestStruct1)的结果为 12。

3.7，#运算符

#也是预处理？是的，你可以这么认为。那怎么用它呢？别急，先看下面例子：

```
#define SQR(x)    printf("The square of    x    is %d.\n", ((x)*(x)));
```

如果这样使用宏：

```
SQR(8);
```

则输出为：

```
The square of    x    is 64.
```

注意到没有，引号中的字符 x 被当作普通文本来处理，而不是被当作一个可以被替换的语言符号。

假如你确实希望在字符串中包含宏参数，那我们就可以使用“#”，它可以把语言符号转化为字符串。上面的例子改一改：

```
#define SQR(x)    printf("The square of    "#x"    is %d.\n", ((x)*(x)));
```

再使用：

```
SQR(8);
```

则输出的是：

```
The square of    8    is 64.
```

很简单吧？相信你现在已经明白#号的使用方法了。

3.8，##预算符

和#运算符一样，##运算符可以用于宏函数的替换部分。这个运算符把两个语言符号组

合成单个语言符号。看例子：

#define XNAME(n) x ## n

如果这样使用宏：

XNAME(8)

则会被展开成这样：

x8

看明白了没？##就是个粘合剂，将前后两部分粘合起来。