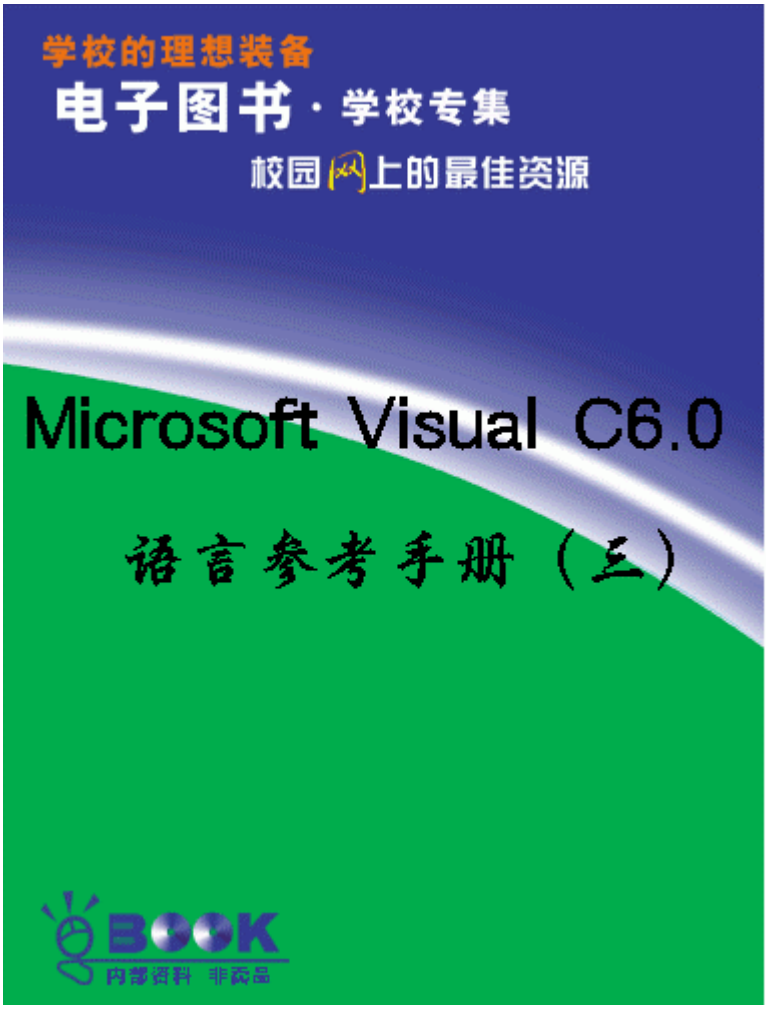
学校的理想装备
电子图书·学校专集
校园网上的最佳资源
Microsoft Visual C6.0
语言参考手册（三）
BOOK
内部资料 非卖品

返回总目录

Microsoft Visual C++ 6.0
预处理器参考手册

目　录

引　　言

本书用于介绍 Microsoft Visual C++的预处理器,预处理器是 C 和 C++文件送入编译器之前对其进行预处理的一种工具,它的功能如下:

- 定义和反定义宏
- 扩展宏
- 条件编译代码
- 插入指定的文件
- 指示编译时产生的错误信息
- 将特定机器的规则用于代码的指定部分

特殊术语

在本书中,名词“参量”指的是传送给一个函数的实体。有时候,它用“actual”或“formal”修饰,它们分别用于表示函数调用时的参量表达式和在函数定义时的参量说明。

名词“变量”指的是一种简单的 C 类型数据对象,名词“对象”指的是 C++对象和变量;它是一个含义广泛的名词。

第 1 章　预 处 理 器

预处理器是一种处理源文件文本的文本处理器，它是翻译起始阶段的一个组成部分。
预处理器并不在语法上分析处理源文本，但出于定位宏调用的目的，它将源文本分开语言符号。虽然编译器一般在初次编译时启动预处理器，但预处理器也可以不经编译，单独地处理文本。
Microsoft 特殊处→
用/E 或/EP 编译器选项进行预处理之后，你可以得到一个源代码的列表。在多数情况下，启动预处理器和输出结果文本到输出设备，这两种选项都是控制台指令，这两种选项的区别在于/E 包括了#line 指令，/EP 没有这些指令。
Microsoft 特殊处结束

特殊术语

在本书中，名词“参量”指的是传送给一个函数的实体。有时候，它用“actual”或“formal”修饰，它们分别用于表示函数调用时的参量表达式和在函数定义时的参量说明。名词“变量”指的是一种简单的 C 类型数据对象，名词“对象”指的是 C++对象和变量；它是一个含义广泛的名词。

翻译阶段

C和C++程序由一个或多个源文件组成,它们都包含了程序的某些文本,一个不包含代码部分的源文件和它的包含文件(用#indude 预处理器指令包含的文件),若被条件编译指令(比如#if)调用,则称其为一个“转换单元”。
源文件可被翻译多次,翻译过去的文件事实上是很正常的。已经翻译了的翻译单元可保存在单独的对象文件或对象代码库里,这些单个的转换单元可被连接形成一个可执行文件或动态链接库(DLL)。
转换单元可采用下列形式通信:

- 调用具有外部连接的函数。
- 调用具有外部连接的类成员函数。
- 直接更改具有外部连接的对象。
- 文件的直接更改。
- 内部外理通信(仅限于基于Microsoft Windows的应用程序)。

以下是编译器翻译文件的各个阶段:
字符映射
源文件中的字符被映射为内部源代码的形式。 此阶段三字母序列被转换为单字符的内部表现形式。
行拼接
在此阶段,源文件中所有以反斜杠(\)结尾且其后紧跟一换行符的行, 将与下一行连接,从而由物理行生成逻辑行。所有非空源文件结束于一个前面没有反斜杠的换行符。

语言符号化
此阶段源文件被分为预处理语言符号和空白字符。源文件中每个注释被用一个空白字符代替。换行符被保留。
预处理
此阶段执行预处理指令并将宏扩展至源文件,#include 语句调用对所有包括文本启动前面三个翻译步骤开头的翻译过程。
字符集映射
所有的源字符集成员和转义序列将转换为执行字符集中的等价形式,对于 Microsoft C 和 C++来说,源字符集和执行字符集都是 ASCII 码。
字符串合并
所有相邻的字符串和宽字符文字都将被合并。例如:“String”“concatenation”合并为“String concatenation”。
翻译
所有的语言符号将按语法和语义规则进行分析;这些语言符号被转换为目标代码。
链接
此阶段所有的外部引用被分解以生成一个可执行程序或一个动态链接库。
编译器在翻译过程中遇到语法错误时,将发出一个警告或错误信息。
链接器分解所有的外部引用,并把一个或多个分开处理的转换单元和标准库联接起来,以生成一个可执行程序或动态链接库(DLL)。

预处理器指令

预处理器指令如#define 和#ifdef，一般被用在不同的运行环境下，使源程序易于更改和编译。源文件中的指令指示预处理器执行特有的行为。例如,预处理器可替换文本中的语言符号,将其它的文件内容插入源文件中,或移走文本的一部分以抑制文件中某部分的编译。预处理器行在宏扩展之前被识别且执行。不过,如果宏扩展看起来象一个预处理器指令,该命令将不能被预处理器识别。
除转义序列之外,预处理器语句采用与源文件语句相同的字符集。在预处理器语句中的字符集和可执行程序的字符集是一样的。预处理器也可识别负字符值。
预处理器可识别如下指令:

#define	#error	#import	#undef
#elif	#if	#include	
#else	#ifdef	#line	
#endif	#ifndef	#pragma	

数字符号(#)是包含预处理器指令的行中的第一个非空白字符。 空白字符可出现在数字符号和指令的第一个字母之间。某些指令包含参量和值。指令之后的任何文本(除作为指令一部分的参量或值之外)必须放在单行注释分界符(//)之后或注释分界符(/* */)之间。
预处理器指令可出现在源文件的任何地方,但它们仅用于源文件的剩余部分。

#define 指令

可以用#define 指令给程序中的常量取一个有意义的名称,其语法的两种形式如下:

语法

#define 标识符 语言符号字符串$_{opt}$

#define 标识符[(标识符$_{opt}$,...,标识符$_{opt}$)] 语言符号字符串$_{opt}$

#define 指令用语言符号字符串替换源文件中一个标识符的所有出现,标识符仅在它形成一个语言符号时被替换(参见“Microsoft Visual C++ 6.0 参考库”的“Microsoft Visual C++ 6.0 语言参考手册”卷的第 1 章“词法规定”中的“语言符号”)。例如,若标识符出现在一个注释、一个字符串或作为一个长标识符的一部分之中,它就不被替换。

一个不带语言符号字符串的#define 指令将移走源文件中每次标识符的出现。标识符保留其定义且能用#defined 和#ifdef 测试。

语言符号字符串参量由一系列语言符号组成,如关键字、常量或完整的语句。一个或多个空白字符可将语言符号字符串和标识符分开。空白字符不会被认为是被替换文本的一部分,文本最后语言符号之后的空白也不会认为是替换文本的一部分。

形式参数名称出现在语言符号字符串中以标志出实际值被替换的位置,每个参数名称可在语言符号字符串中出现多次,且可以以任何次序出现。调用时参量的数目必须与宏定义的参数数目相匹配。圆括号的自由运用可确保正确地说明复杂的实际参量。

用第二种语法形式可创建类似函数的宏。这种形式接受一个用圆括号括起的可选参数表。在最初定义之后引用该标识符,可以使用实际参量替代形式参数的语言符号字符串参量的形式,来替换标识符(标识符 $_{opt}$,...,标识符 $_{opt}$)的每次出现。
表中的形式参数必须用逗号隔开。该表中的每个名称都必须是唯一的,且此参量表必须包括在圆括号中,标识符和左边的圆括号之间不能有空格。对于占用多行的长指令可使用行连接,把反斜杠(\)放在换行符前。 形式参数名称的范围延伸到结束语言符号字符串的换行符。
当一个宏以第二种语法形式定义时,参量表后的文本实例就构成一个宏调用。在源文件中,一个标识符实例后的实际参量必须与宏定义的相应形式参数匹配。每个语言符号字符串之前无字符串化(#)、字符化(#@)或语言符号粘贴(##)操作符,或其后无##操作符的形式参量,都被相应的实际参量所替换。在指令替换形式参数之前,实际参量中的任何宏都将被扩展(本章之后的“预处理器操作符”中将介绍这些操作符)。
以下带参量宏的例子说明了#define语法的第二种形式:

```
//定义光标行的宏
#define CURSOR(top,bottom)  ((top) << 8) | bottom))

//获取指定范围中的一个随机整数的宏
#define getrandom(min,max) \
  ((rand()%(int)(((max)+1)-(min)))+(min))
```

有副作用的参量有时会导致宏产生不希望的结果。一个给定的形式参量在语言

符号字符串中可能出现多次。如果该形式参数被一个有副作用的表达式所替换，则该表达式及其副作用，可能被求值多次(参见本章后面“语言符号粘贴操作符##”中的例子)。
#undef 指令可使一个标识符的预处理器定义失效。有关的更多信息参见#undef指令。若一个被定义的宏名称出现在语言符号字符串中(即使是另一个宏扩展的结果)，它将不被扩展。
除非第二次定义（#define）宏与原定义完全相同，否则重定义一个已定义过的宏将产生一个错误信息。
Microsoft 特殊处→
Microsoft C/C++允许一个宏的重定义，但会产生一个警告信息，说明新的定义与原定义相同。ANSI C 认为宏的重定义是错误的。例如，下面的宏对 C/C++是相同的，但会产生一个警告信息：

```
#define test(f1,f2) (f1*f2)
#define test(a1,a2) (a1*a2)
```

Microsoft 特殊处结束
这个例子用于说明#define指令：

```
#define WIDTH  80
#define LENGTH (WIDTH+10)
```

第一个说明定义标识符 WIDTH 为整形常量 80，且用 WIDTH 和整形常量 10 定义 LENGTH。LENGTH 的每次出现都用(WIDTH+10)所替换，接着，WIDTH+10 的每次出现都用表达式(80+10)替换。WIDTH+10 的圆括号非常重要，因为它们决定着如下语句的解释。

var=LENGTH*20；
经预处理后该语句变为：
var=(80+10)*20；
求得值为 1800,若无括号,结果为：
var=80+10*20
其值为 280。
Microsoft 特殊处→
在文件开头用/D 编译器选项定义宏和常量,和用一个#define 预处理指令效果是一样的。能用/D 选项定义的宏可达 30 个。
Microsoft 特殊处结束

#error 指令

采用 error 指令可产生编译错误信息。
语法
#error 语言符号字符串
错误消息包括语言符号字符串参量,且从属于宏扩展。这些指令对于检测程序的前后矛盾和预处理时的违犯约束是非常有用的,以下例子说明了预处理时的出错处理：

```
#if !defined(__cplusplus)
#error C++ complier required.
#endif
```

当遇到#error 指令时,编译终止。

#if, #elif, #else 和#endif 指令

#if、#elif、#else 和#endif 指令控制源文件中某部分的编译。如果表达式(#if 之后)有一个非 0 值,则紧跟在#if 指令之后的行组将保留在转换单元中。

语法

条件的：

if 部分 elif 部分 $_{opt}$ else 部分 $_{opt}$ endif 行

if 部分：

if 行 文本

if 行：

#if 常量表达式

#ifdef 标识符

#ifndef 标识符

elif 部分：

elif 行 文本

elif 部分 elif 行 文本

elif 行：

#elif 常量表达式

else 部分：

else 行 文本

else 行：

#else

endif 行：

#endif

源文件中每个#if 指令都必须与最近的一个#endif 相匹配。在#if 和#endif 指令之前的#elif 指令的数目是不限的,但最多只能有一个#else 指令。#else 必须是#endif 之前的最后一个指令。

#if、#elif、#else 和#endif 指令可嵌套在其它#if 指令的文本部分。每个嵌套的#else、#elif 或#endif 指令应属于前面最近的一个#if 指令。

所有的条件编译指令,如#if 和#ifdef,必须与文件结束前最近的#endif 指令匹配;否则,将产生一个错误消息。当条件编译指令包括在包含文件中时,他们必须满足相同的条件:在包含文件结尾没有不匹配的条件编译指令。

宏替换应在#elif 命令后的命令行部分内进行,因此一个宏调用可用在常量表达式中。

预处理器选择一个给定文本的出现之一作进一步的处理。文本中指定的一个块可以是文本的任何序列。它可能占用一行以上。通常该文本是对于编译和预处理器有意义的程序文本。

预处理器处理选择文本并将其传送给编译器。若该文本包含预处理器指令,预处理器将执行这些指令。编译器只编译预处理器选定的文本块。

预处理器通过求值每个#if 或#elif 指令之后的常量表达式直到找到一个为真(非 0)的常量表达式来选择单个文本项。预处理器选择所有的文本(包括以#开头的其它预处理器指令)直到它关联的#elif、#else 或#endif。

如果常量表达式的所有出现都为假,或者如果没有#elif 指令,预处理器将选择#else 后的文本块。如果#else 被忽略，且所有#if 块中的常量表达式都为假，

则不选择任何文本块。
常量表达式是一个有以下额外限制的整型常量表达式：

- 表达式必须是整型且可以包括整型常量,字符常量和 defined 操作符。
- 表达式不能使用 sizeof 或一个类型造型操作符。
- 目标环境不能表示整数的所有范围。
- 在翻译表示中,int 类型和 long 类型以及 unsigned int 类型和 unsigned long 类型是相同的。
- 翻译器可将字符常量翻译成一组不同于目标环境的代码值。为了确定目标环境的属性，应在为目标环境建立的应用程序中检测 LIMITS.H 的宏值。
- 表达式不需执行所有的环境查询,但需与目标计算机的执行过程细节隔离开。

预处理器操作符 defined 可用于特殊的常量表达式,语法如下：
语法
defined(标识符)
defined　标识符
若此标识符当前已定义,则该常量表达式被认为是真(非 0);否则,条件为假(0)。一个定义为空文本的标识符可认为已定义。defined 指令只能用于#if 和#endif 指令。
在如下例子中,#if 和#endif 指令控制着三个函数调用中某一个的编译：

```
#if defined (CREDIT)
```

```
  credit();
#elif defined(DEBIT)
  debit();
#else
  printerror();
#endif
```

若标识符 CREDIT 已定义,则对于 credit 的函数调用被编译。若标识符 DEBIT 被定义,则对于 debit 的函数调用被编译。若未定义任何标识符,将编译对于 printerror 的函数调用。

注意,在 C 和 C++中,CREDIT 和 credit 是不同的标识符,因为它们的大小写不一样。

如下例子中的条件编译语句给出了一个名称为 DLEVEL 的已定义的符号常量:

```
#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
       #define STACK 200
    #else
      #define STACK 100
    #endif
#else
    #define  SIGNAL  0
    #if STACKUSE==1
```

```
      #define STACK 100
    #else
      #define STACK 50
#endif
#endif
#if DLEVEL==0
    #define STACK  0
#elif DLEVEL==1
    #define STACK 100
#elif DLEVEL > 5
    display(debugptr);
#else
    #define STACK 200
#endif
```

第一个#if 块中有两组嵌套的#if、#else 和#endif 指令。第一组指令仅当DLEVELl>5 为真时执行;否则,执行#else 之后的语句。

第二组中的#elif 和#else 指令选择基于 DLEVEL 值的四个选项之一。常量 STACK 依据 DLEVEL 定义为 0，100 或 200。若 DLEVEL 大于 5,则编译语句：

```
#elif DLEVEL > 5
display(debugptr);
```

且此时不定义 STACK。

条件编译一般用于防止同一头文件的多重包含。C++ 中在头文件内经常定义类

的位置,可使用如下结构来防止多次定义。

```
//EXAMPLE.H 例子头文件

#if !defined(EXAMPLE_H)
#define ExampleE_H

class Example
{
...
};

#endif  //!defined(EXAMPLE_H)
```

上面的代码用于检查符号常量 EXAMPLE_H 是否已定义。若已定义,该文件就已被包括且不需再处理;如果未定义,常量 EXAMPLE_H 将被定义,以标记 EXAMPLE.H 为已经处理。

Microsoft 特殊处→

条件编译表达式被看作为 signed long 值,且这些表达式与 C++中的表达式采用相同的规则求值。例如,表达式:

```
#if 0xFFFFFFFFL > 1UL
```

为真。

Microsoft 特殊处结束

#ifdef 和 ifndef 指令

#ifdef 和#ifndef 指令与使用 defined(标识符)操作符的作用是一样的。
语法
#ifdef 标识符
#ifndef 标识符
等同于
#if defined 标识符
#if !defined 标识符
#if 指令能用的任何地方都可以用#ifdef 和#ifndef 指令。当标识符已被定义时,#ifdef 标识符语句等同于#if 1;而当标识符未定义或用#undef 指令对其反定义时,该语句等同于#if 0。这些指令仅用于检查 C 或 C++源代码中是否出现该标识符,而不是用于检查 C 或 C++源程序中该标识符的说明。
提供这几个指令只为了与该语言的老版本兼容。目前的趋势是偏向于采用 defined(标识符)定义常量表达式的#if 指令。
#ifndef 指令测试与#ifdef 相反的条件。若标识符未定义(或已用#undef 反定义),其条件为真(非 0);反之,条件为假(0)。
Microsoft 特殊处→
可以使用/D 选项从命令行传送标识符,采用/D 选项至多可以指定 30 个宏。
检查一个定义是否存在是非常有用的,因为定义可从命令行传送。例如:
//prog.cpp
#ifndef test //这三个语句放在你的代码中

```
#define final
#endif

CL /Dtest prog.cpp  //这是编译的命令
Microsoft 特殊处结束

#import 指令

C++特殊处→
#import 指令用于从一个类型库中结合信息。该类型库的内容被转换为 C++类，
主要用于描述 COM 界面。
语法
#import "文件名" [属性]
#import <文件名> [属性]
属性：
    属性 1,属性 2,...
    属性 1 属性 2 ...
文件名是一个包含类型库信息的文件的名称。一个文件可为如下类型之一：
        ● 一个类型库(.TLB 或.ODL)文件。
        ● 一个可执行(.EXE)文件。
        ●  一个包含类型库资源(如.OCX)的库文件(.DLL)。
        ● 一个包含类型库的复合文档。
        ● 其它可被 LoadTypeLib API 支持的文件格式。
```

文件名之前可以有一个目录规格。文件名必须是一个已存在文件的名称。 两种格式的区别是当路径未完全说明时,预处理器检索类型库文件的顺序不同。

语法格式	动作
引号格式	这种格式让预处理器首先搜索与包含#import 语句的文件同一目录的类型库文件,然后在所有包括(#include)该文件的目录中搜索,最后在如下路径中搜索
尖括号格式	这种格式指示预处理器沿以下路径搜索类型库文件

编译器在以下目录中搜索已命名的文件:

1. PATH 环境变量路径表。
2. LIB 环境变量路径表。
3. 用/I(额外的包括目录)编译器选项指定的路径。

#import 可以任选地包含一个或多个属性。这些属性使编译器改变类型库头文件的内容。一个反斜杠(\)符可用在一个单一的#import 语句中包含额外的行,例如:

```
#import "test.lib"  no_namespace \
  rename("OldName","NewName")
```

#import 属性列出如下:

```
        exclude                     high_method_prefix
        high_property_prefixes      implementation_only
        include(...)                inject_statement
        named_guids                 no_auto_exclude
```

续表

no_implementation	no_namespace
raw_dispinterfaces	raw_interfaces_only
raw_method_prefix	raw_native_types
raw_property_prefixes	rename
rename_namespace	

#import 指令可创建两个在 C++源代码中重构类型库内容的头文件,第一个头文件和用 Microsoft 接口定义语言(MIDL)编译器生成的头文件类似,但有额外的编译器生成代码和数据。第一个头文件与类型库有相同的基本名,其扩展名为.TLH。第二个头文件也有与类型库相同的基本名,其扩展名为.TLI。它包括了编译器生成成员函数的实现,且被包含在(#include)的第一个头文件内。

两个头文件都在用/Fo(命名对象文件)选项指定的输出目录中。随后它们被读出和编译,就像第一个头文件被#include 指令命名一样。

以下是伴随#import 指令的编译器优化:

- 头文件被创建时,将被分配与类库相同的时间标志。
- 处理#import 时,编译器首先测试头文件是否存在,是否过期。若条件为真,就不需重新创建。
- 编译器延迟对于 OLE 子系统的初始化,直到碰到第一个#import 命令。

#import 指令也可参与最小重建且可被置于一个预编译头文件中。

基本类型库头文件

基本类型库头文件由七个部分组成：
1. 头部固定正文：由注释、COMDEF.H(定义用在头部的一些标准宏)的#include语句和其它繁杂的安装信息组成。
2.前向引用和类型定义：由象 struct IMyinterface 之类的结构说明和用于一些TKIND_ALIAS 项的类型定义组成。
3.灵敏指针说明：模块类_com_ptr_t 是一个封装接口指针和消除调用 AddRef、Release 和 QueryInterface 函数需求的灵敏指针实现。此外，它隐藏了创建一个新 COM 对象中的 CoCreateInstance 调用。此部分采用宏语句_COM_SMARTPTR_TYPEDEF 将 COM 接口的类型定义创建为_com_ptr_t 模板类的模板特例化。例如，对于界面 IFoo，.TLH 文件包含有：

```
_COM_SMARTPTR_TYPEDEF(IFoo, __uuidof(IFoo));
```

编译器将其扩展为：

```
type def _com_ptr_t<_com_IIID<IFoo,_ _uuidof(IFoo) >> IFooPtr;
```

类型 IFooPtr 可以用在原始的界面指针 IFoo*的地方。结果，就不需调用各种IUnknown 成员函数。
4. 类型信息(typeinfo)说明：主要由类定义和其它项组成，这些项说明由ITyptLib:GetTypeInfo 返回的单个的信息类型项目。在这部分，每个来自于类型库的信息类型都以一种依赖于 TYPEKIND 信息的格式反映在该头部。
5. 任选旧式 GUID 定义：包含命名的 GUID 常量的初始化过程，这些定义是格式CLSID_CoClass 和 IID_Interface 的名称，与那些由 MIDL 编译器产生的类似。

6. 用于第二个类型库头部的#include 语句。
7. 结尾固定正文:目前包括#pragma pack(pop)。
以上这些部分除头部固定正文和结尾固定正文部分之外,都被包括在原来的 IDL 文件中以 library 语句指定其名称的名称空间中。你可以通过用名称空间显式限定或包括如下语句从类型库头部使用该名称。
using namespace MyLib
在源代码的#import 语句之后立即
名称空间可用#import 指令的 no_namespace 属性来阻止。但阻止的名称空间可能导致名称冲突。名称空间也可用 rename_namespace 属性重新换名。
编译器提供完全路径给需要依赖当前正在处理的类型库的任何类型库。路径以注释格式写入到由编译器为每个处理的类型库生成的类型库头部(.TLH)。
如果一个类型库包含了对其它类型库定义的类型引用,.TLH 文件将包括以下注释:

```
//
//Cross-referenced type libraries:
//
//#import "c:\path\typelib0.tlb"
//
```

在#import 注释中的实际文件名是存储在寄存器中交叉引用的类型库全路径。
如果你遇到由于省略类型定义的错误时,检查.TLH 头部的注释,看哪一种依赖类型库需要先输入。在编译该.TLI 文件时可能的错误有语法错误(例如 C2143,C2146,C2321)、C2501(缺少说明指示符)或 C2433(在数据说明中禁止'

inline')。
你必须确定哪些依赖注释是不被系统头部给出的,而是在依赖类型库的#import 指令前的某处给出一个#import 指令以消除这些错误。

exclude 属性

exclude(“ 称 1”[,“ 名称 2”,...])
名称 1
    被排斥的第一个项
名称 2
    被排斥的第二个项(如有必要)
类型库可能包含在系统头部或其它类型库内定义的项的定义。该属性可用于从生成的类型库头文件中排斥这些项。这个属性可带任意数目的参量,每个参量是一个被排斥的高级类型库项目:

high_method_prefix 属性

high_method_prefix("*Prefix*")
Prefix
被使用的前缀
在缺省的情况下,高级错误处理属性和方法用一个无前缀命名的成员函数来展示。这个名称来自于类型库。high_method_prefix 属性说明一个前缀以用于命名这些高级属性和方法。

high_property_prefixes 属性

high_property_prefixes("*GetPrefix*, ""*PutPrefix*, ""*PutRefPrefix*")

GetPrefix
用于 propget 方法的前缀
PutPrefix
用于 propput 方法的前缀
PutRefPrefix
用于 propputref 方法的前缀
在缺省情况下,高级错误处理方法,如 propget、propput 和 propputref,分别采用以前缀 Get、Put 和 PutRef 命名的成员函数来说明。high_property_prefixes 属性用于分别说明这三种属性方法的前缀。

implementation_only 属性

implementation_only 属性禁止.TLH 头文件(基本头文件)的生成。 这个文件包括了所有用于展示类型库内容的说明。该.TLI 头文件和 wrapper 成员函数的实现,将被生成且包含在编译过程中。
当指定该属性时,该.TLI 头部的内容将和用于存放普通.TLH 头部的内容放在相同的名称空间。此外,该成员函数不会作为联编说明。implementation_only 属性一般希望与 no_implementation 属性配对使用,以跟踪预编译头文件(PCH)之外的实现。一个有 no_implementation 属性的#import 语句被置于用来创建 pch 的源区域中,结果 PCH 将被一些源文件所用。一个带 implementation_only 属性

的#import 语句随后被用在 PCH 区域之外。在一个源文件里只需用一次这种语句。这将生成不需对每个源文件进行额外重编译的所有必要的 wrapper 成员函数。
注意:一个#import 语句中的 implementation_ only属性必须和相同类型库中no_implementation 属性的另一个#import 语句配套使用。否则,将产生编译错误。这是因为带 no_implementation 属性的#import 语句生成的 wrapper 类定义需要编译 implementation_only 属性生成的语句实现。

include(...)属性

include(名称 1[,名称 2,...])
名称 1
第一个被强制包含的项
名称 2
第二个被强制包含的项(如果必要)
类型库可能包含在系统头部或其它类型库中定义的项的定义。#import 指令试图用自动排斥这些项来避免多重定义错误。若这些项已经被排斥,象警告 C4192 所指出的那样,且它们不应该被排斥,则这个属性可用于禁止自动排斥。该属性可带任意数目的参量,每个参量应是被包括的类型库项的名称。

inject_statement 属性

inject_statement("*source_text*")

source_text
被插入到类型库头文件的源文本。
inject_statement 属性将其参量作为源文本插入类型库头部。此文本被置于包括头文件中类型库内容的名称空间说明的起始处。

named_guids 属性

named_guids 属性让编译器定义和初始化模板 LIBID_MyLib、CLSID_MyCoClass、IID_MyInterface 和 DIID_MyDispInterface 的旧式格式的 GUID 变量。

no_implementation 属性

该属性阻止.TLI 头文件的生成,这个文件包含 wrapper 成员函数的实现。如果指定这个属性,则展示类型库项说明的.TLH 头将生成没有一个#include 语句包括该.TLI 头文件。
该属性与 implementation_only 属性配套使用。

no_auto_exclude 属性

类型库可能包括在系统头部或其它类型库中定义的项的定义。#import 试图通过自动排斥这些项来避免多重定义错误。当这样做时,每个被排斥的项都将生成一个 C4192 警告信息。你可禁止这个属性使用自动排斥。

no_namespace 属性

#import 头文件中的类型库内容一般定义在一个名称空间里。名称空间的名称在原来 IDL 文件的 library 语句中指定。如果指定 no_namespace 属性,编译器就不会生成这个名称空间。
如果你想使用一个不同的名称空间,应代替使用 rename_namespace 属性。

raw_dispinterfaces 属性

raw_dispinterfaces 属性让编译器生成一个低级 wrapper 函数。该函数用于调用 IDispatch::Invoke 和返回 HRESULT 错误代码的 dispinterface 方法和属性。如果未指定此属性,则只生成高级 wrapper,它在失败时丢弃该 C++异常。

raw_interfaces_only 属性

raw_interfaces_only 属性禁止生成错误处理 wrapper 函数以及使用这些 wrapper 函数的__declspec(属性)说明。
raw_interfaces_only 属性也导致删除在命名 non_property 函数中的缺省前缀。通常该前缀是 raw_。若指定此属性,函数名称将直接从类型库中生成。
该属性只允许展示类型库的低级内容。

raw_method_prefix 属性

raw_method_prefix("*Prefix*")

Prefix
被使用的前缀
用 raw_作为缺省前缀的成员函数展示低层属性和方法,以避免与高级错误处理成员函数的名称冲突。raw_method_prefix 属性用于指定一个不同的前缀。
注意: raw_method_prefix 属性的效果不会因 raw_method_prefix 属性的存在而改变。在说明一个前缀时,raw_method_prefix 总是优先于 raw_interfaces_only。若两种属性用在同一个 #import 语句中时,则采用 raw_method_prefix 指定的前缀。

raw_native_types 属性

在缺省情况下,高级错误处理方法在 BSTR 和 VARIANT 数据类型和原始 COM 界面指针的地方使用 COM 支持类_bctr_t 和_variant_t。这些类封装了分配和取消分配这些数据类型的存储器存储的细节,并且极大地简化了类型造型和转换操作。raw_native_types 属性在高级 wrapper 函数中禁止使用这些 COM 支持类,并强制替换使用低级数据类型。

raw_property_prefix 属性

raw_property_prefix("*GetPrefix*","*PutPrefix*","*PutRefPrefix*")
GetPrefix
用于 propget 方法的前缀
PutPrefix

用于 propput 方法的前缀
PutRefPrefix
用于 propputref 方法的前缀
在缺省情况下,低级方法 propget、propput 和 propputref 分别用后缀为 get_、put_和 putref_的成员函数来展示。这些前缀与 MIDL 生成的头文件中的名称是兼容的。raw_property_prefixes 属性分别用于说明这三个属性方法的前缀。

rename 属性

rename("*OldName*", ""*NewName*")
OldName
类型库中的旧名
NewName
用于替换旧名的名称
rename 属性用于解决名称冲突的问题。若该属性被指定,编译器将在类型库中的 OldName 的所有出现处用结果头文件中用户提供的 NewName 替换。
此属性用于类型库中的一个名称和系统头文件中的宏定义重合时。若这种情况未被解决,则将产生大量语法错误,如 C2059 和 C2061。
注意:这种替换用于类型库的名称,而不是用于结果头文件中的名称。
这里有一个例子:假设类型库中有一个名称为 MyParent 的属性,且头文件中定义了一个用在#import 之前的宏 GetMyParent。由于 GetMyParent 是用于错误处理属性 get 的一个 wrapper 函数的缺省名称,所以将产生一个名称冲突。为解决这个问题,使用#import 语句中的以下属性:

rename("MyParent","MyParentX")
该语句将重新命名类型库中的名称 MyParent,而试图重新命名 GetMyParent wrapper 名称将会出错:
rename("GetMyParent","GetMyParentX")
这是因为名称 GetMyParent 只出现在结果类型库头文件中。

rename_namespace 属性

rename_namespace("*NewName*")
NewName
名称空间的新名称
rename_namespace 属性用于重新命名包含类型库内容的名称空间。它带有一个指定名称空间新名 newname 的参量。
消除名称空间可以使用 no_namespace 属性。
C++特殊处结束

#include 指令

#include 指令告诉预处理器处理一个指定文件的内容,就象这些内容以前就在这条指令出现的源程序中。你可以把常量和宏定义放在包含文件中,然后用 #include 指令把这些定义加到任何源文件中。包含文件对于外部变量和复杂数据类型结合的说明也是有用的。
你只需在为此目的创建的一个包含文件中定义和命名这些类型一次。

语法

#include "*path-spec*"

#include <path-spec>

path_spec 是一个前面有目录说明的任选文件名。这个文件名必须命名一个现存文件。

path_spec 的语法依赖于编译该程序的操作系统。

这两种语法格式都导致用已说明的包含文件的全部内容来替换该指令。两种格式的区别在于路径未完整指定时预处理器搜索头文件的顺序。

语法格式	动作
引号格式	这种格式指示预处理器先在包含#include 语句的文件的相同目录内搜索,然后在任何包括该文件的目录中搜索。随后预处理器沿着/I 编译器选项指定的路径搜索,最后是在INCLUDE 环境变量说明的路径搜索
尖括号格式	这种格式指示预处理器首先在/I 编译器选项指定的路径中搜索包含文件。然后在 INCLUDE 环境变量说明的路径中搜索

一旦预处理器找到指定文件,它就立即停止搜索。如果用双引号给出一个明确完整的包含文件的路径,预处理器将只搜索该路径规格而忽略标准目录。

如果在双引号间的文件名不是一个完整的路径规格,预处理器将先搜索“父”文件的目录。父文件是一个包含#include 指令的文件。例如,如果你把名称为 file2 的文件包括在一个名称为 file1 的文件中,file1 就是父文件。

包含文件可被嵌套;这指的是一个#include 指令出现在以另一个#include 指令

命名的文件里。例如,以上的文件 file2,可包含文件 file3,在这种情况下, file1是 file2 的父文件,而且是 file3 的祖父文件。
当包含文件嵌套时,目录搜索首先由父文件的目录开始,然后,搜索祖父文件的目录。
因此,搜索从包含当前处理源文件的目录开始,若文件未找到,搜索就转到/I 编译器选项指定的目录,最后搜索 include 环境变量指定的目录。
下面的例子给出使用尖括号的文件包括:

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

这个例子把名称为 STDIO.H 的文件内容加入到源程序中。尖括号指示预处理器在搜索完/I 编译器选项说明的目录之后,搜索 STDIO.H 的环境变量指定的目录。
下面的例子给出用引号格式的文件包括:

```
#include "defs.h"
```

这个例子把 DEFS.H 指定的文件内容加入源程序。双引号标记意味着预处理器首先搜索包含父源文件的目录。
包含文件的嵌套可高达 10 层,只要在处理嵌套的#include 指令时,预处理器就会不断地把包含文件加入到最初的源文件中。
Microsoft 特殊处→
为了定位可包括源文件,预处理器首先搜索/I 编译器选项指定的目录。若/I 选项未给定或已失败,预处理器就用 INCLUDE 环境变量搜索尖括号内的包含文件。INCLUDE 环境变量和/I 编译器选项可包含用分号分开的多个路径。若在/I 选项的部分或在 INCLUDE 环境变量里有多于一个的目录,预处理器将以它们出现的顺序对它们进行搜索。

例如,命令:

```
CL /ID:\MSVC\INCLUDE MYPROG.C
```

导致预处理器在目录 D:\MSVC\INCLUDE 中搜索诸如 STDIO.H 的包含文件。命令:

```
SET INCLUDE=D:\MSVC\INCLUDE
CL MYPROG.C
```

有相同的作用。如果所有搜索都失败了,将产生一个致命编译错误。

如果用包括一个冒号的路径(例如,F:\MSVC\SPECIAL\INCL\TEST.H)来完整地说明一个包含文件的文件名,预处理器将沿此路径搜索。

对于指定为#include "path_spec"的包含文件,目录搜索将从父文件的目录开始,然后搜索祖父文件的目录。因此,搜索将从包含当前处理的#include 指令的源文件的目录开始,如果没有祖父文件或文件未找到,搜索将继续,就像文件名包括在尖括号中一样。

Microsoft 特殊处结束

#line 指令

#line 指令告诉预处理器将编译器内部存储的行号和文件名转变为一个给定的行号和文件名。编译器使用该行号和文件名指出编译过程中发现的错误。行号一般指的是当前输入行,文件名指当前输入文件。每处理一行,行号就增 1。

语法

#line

数字序列 “文件名”$_{opt}$

数字序列的值可以是任何整型常数。宏替换可在预处理语言符号中执行,但结果

必须求值为正确的语法。文件名可以是任意字符的组合,且应括在双引号(“ ”)间。如果省略文件名,则前面的文件名保持不变。
你可以通过编写一个#line 指令来改动源行号和文件名。翻译器使用行号和文件名来确定预定义宏__FILE__和__LINE__的值。你可以使用这些宏把自描述错误消息加入到程序文本中。有关这些宏的更多信息参见预定义的宏。
__FILE__宏扩展成内容为用双引号(“ ”)括起的文件名的一个字符串。
如果你改变行号和文件名,编译器将忽略原有的值,用新值继续处理。#line 指令通常被程序生成器用来生成指向最初源程序的错误消息,而不是生成程序。
下面的例子用于说明#line 以及__LINE__和__FILE__宏。在这个语句中,内部存储的行号设置为 151,文件名改为 copy.c。

```
#line 151 "copy.c"
```

在这个例子中,若一个给定的“断言”(assertion)不为真,则宏 ASSERT 使用预定义宏__LINE__和__FILE__打印出一个关于源文件的错误消息。

```
#define ASSERT(cond)

if( !(cond) ) \
{ printf("assertion error line %d, file(%s)\n",\
 __LINE__,__FILE__); }
```

Null 指令

空预处理器指令是一行中一个单独的数字标号(#),无任何作用。
语法

#

#undef 指令

正如其名所隐含的,#undef 指令取消(反定义)一个原来由#define 指令创建的名称。
语法
#undef
    标识符
#undef 指令取消标识符的当前定义。其结果是,标识符的每次出现都将被预处理器所忽略。为取消一个用#undef 的宏定义,只须给出宏的标识符,不须给出参数表。
你也可以将#undef 指令用于一个原来未定义的标识符。这将确认这个标识符是未定义的。宏替换不能在#undef 语句中执行。
#undef 指令通常和一个#define 指令匹配,以在源程序中创建一个区域,在这个区域中一个标识符有其特定的含义。例如,源程序的一个特有函数可以使用显式常量定义不影响程序余下部分的环境特定值。#undef 指令也可与#if 指令配对以控制源程序的条件编译过程。有关更多信息参见“ #if、#elif、#else 和#endif 指令 ”。
下面的例子中,#undef 指令取消了一个符号常量和一个宏的定义,注意该指令只给出了宏的标识符。

```
#define WIDTH      80
#define ADD(X,Y) (X)+(Y)
```

```
        .
        .
        .
#undef WIDTH
#undef ADD
```

Microsoft 特殊处→

宏可通过采用/U 选项的命令行反定义,此命令行后跟反定义的宏名称。此命令与在文件开头处的#undef 宏名称语句序列的作用是相等的。

Microsoft 特殊处结束

预处理器操作符

#define 指令的文本中有四种预处理器特有的操作符(它们的总结参见下面的表)。

字符化、字符串化和语言符号粘贴操作符将在下面三章中讨论。defined 操作符的信息参见“ #if、#elif、#else 和#endif 指令 ”。

运算符	动作
字符串化操作符(#)	将相应实参置于双引号内
字符化操作符(#@)	将相应的参量置于单引号内,且将其作为字符处理(Microsoft 特殊处)
语言符号粘贴操作符(##)	可将语言符号作为实参使用,且将其合并为其它 的语言符号

续表

定义的操作符	简化在某特定宏指令中复合表达式的写法

字符串化操作符(#)

数字符号或“字符串化”操作符(#)将宏参数(扩展后)转化为字符串常量。它只用于带参量的宏。如果它在宏定义中的一个形式参量之前,宏调用传给的实际参量就被括在双括号中,且被看作为一个字符串文字。然后该字符串文字将替换该宏定义中操作符和形参组合的每次出现。
实参的第一个语言符号之前和最后一个语言符号之后的空白被忽略。实参中语言符号之间的所有空白在结果字符串语义中都被看作为一个空格。因此,若实参中的一个注解出现在两个语言符号之间,它将被看作为一个空格。结果字符串文字自动地与任何仅用空格分开的相邻字符串文字连接。
此外,如果一个包含在参量里的字符在用作一个字符串文字(例如,双引号(")或反斜杠(\)字符)时通常需要一个转义序列,必要的转义反斜杠被自动地插入字符之前。下面的例子给出了一个包含字符串化操作符的宏定义和一个调用该宏的 main 函数:

```
#define stringer(x) printf(#x "\n")

void main( )
{
   stringer(In quotes in the printf function call\n);
   stringer("In quotes when printed to the screen"\n);
```

```
   stringer("This:\" prints an escaped double quote");
}
```

这种调用在预处理时会被扩展,产生如下代码:

```
void main()
{
   printf("In quotes in the printf function call\n" "\n");
   printf("\"In quotes when printed to the screen\"\n" "\n");
   printf("\"This; \\\" prints an escaped double quote \"" "\n");
}
```

当运行该程序时,每行的屏幕输出如下:

```
In quotes in the printf function call

"In quotes when printed to the screen"

"This; \" prints an escaped double quotation mark"
```

Microsoft 特殊处→

Microsoft C(版本 6.0 及更早版本)扩展 ANSI C 的标准,ANSI C 扩展在字符串文字和字符常量中出现的宏形式参量不再被支持。依赖于此扩展的代码应该使用字符串化操作符(#)重写。

Microsoft 特殊处结束

字符化操作符(#@)

Microsoft 特殊处→
字符化操作符只可用于宏参量,若宏定义中#@在一个形参前,则实参应被放在单引号中,在宏扩展时作为一个字符处理。例如:

```
#define makechar(x)  #@x
```

将语句:

```
a=makechar(b);
```

扩展为:

```
a='b';
```

单引号字符不能用于字符化操作符。
Microsoft 特殊处结束

语言符号粘贴操作符(##)

双数字语言符号或“语言符号粘贴”操作符(##),有时称作“合并”操作符,用于类对象宏和类函数宏中。它允许将分开的语言符号加入一个单个语言符号中,因此不能是宏定义的第一个语言符号或最后一个语言符号。
如果一个宏定义中的形参在语言符号粘贴操作符的前后,则形参将立即被未扩展的实参替换。在替换之前不对参量执行宏扩展。
然后,语言符号字符串中语言符号粘贴操作符的每次出现将被删除,其前后的语言符号将被合并。其结果语言符号必须是一个有效的语言符号。若其有效,如果该语言符号代表一个宏名称,则扫描它以发现可能的替换。该标识符表示在替换

前程序中已知合并的语言符号的名称。每个语言符号都代表一个在程序中或在编译器命令行中定义的语言符号。
该操作符前后的空白是任意的。
如下例子说明了程序输出中字符串化操作符和语言符号粘贴操作符的用法：

```
#define paster(n) printf("token" #n "=%d",taken##n)
int token9=9;
```

若一个宏用一个类似于下面的数值参量调用：

```
paster(9);
```

宏将生成：

```
printf("token" "9" "=%d",token9);
```

它变成为：

```
printf("token9 =  %d", token9 );
```

宏

对宏扩展的预处理在所有那些不是预处理指令的行(第一个非空白字符不是#的行),以及其指令并未作为条件编译的一部分而忽略的行中进行。“条件编译”指令允许通过检测一个常量表达式或标识符以决定在预处理过程中哪个文本块送入编译器、哪个文本块从源文件中删除,并以此种方式控制一个源文件中某部分的编译。
#define 指令通常使用有意义的标识符与常量、关键字、常用语句和表达式关联。表示常量的标识符有时被称作“符号常量”或“显式”常量。表示语句或

表达式的常量称为“宏”。在本预处理器文档中,只使用术语“宏”。
当宏的名称在程序源文本或在某些其它预处理器命令的参量中被识别时,它被处理为对该宏的调用。宏名称被宏体的一个拷贝所替换。若该宏接受参量,宏名称后的实参就会替换宏体中的形参。用宏体中处理的拷贝来替换一个宏调用的过程,称为宏调用的“扩展”。
实际的术语中有两种类型的宏。“类对象”宏不带参量,而“类函数”宏可定义为带参量。因此它们的形式和功能都象函数调用,由于宏不生成实际的函数调用,所以有时可用宏替代函数调用使程序运行得更快,(在 C++中,inline 函数通常是一个好方法),然而,如果不小心的定义和使用宏,也可能造成麻烦。在带参量的宏定义时,你必须使用括号以保持一个表达式中正常的优先级,同时宏也不能正确地处理具有副作用的表达式。有关更多的信息参见“#define 指令”中的例子 getrandom。
一旦你定义了一个宏,你不能不经取消该宏原有定义,而重新定义它为一个不同的值。但可用正好相同的定义来重定义该宏,因此,一个程序中宏的相同定义可出现多次。
#undef 指令用于取消宏的定义。一旦取消该宏的定义,就可重新定义该宏为一个不同的值。#define 和#undef 两节分别详细讨论了#define 和#undef 指令。

宏和 C++

C++提供了一些新的功能。其中有些功能替代了原来由 ANSI C 所提供的功能。这些新的功能增强了类型安全性和该语言的可预测性:

- 在 C++中,以 const 说明的对象可用于常量表达式中,这使程序可以

说明有类型和值信息的常量,以及能被调试器逐个字符检查的枚举值的常量。使用预处理器指令#define 定义常量并不精确。除非在程序中找到一个带地址的表达式,否则一个 const 对象将不分配任何存储。

- C++联编函数替代了函数类型宏,相对于宏来说使用联编函数的优势在于:
- 类型安全性。联编函数和一般函数一样需进行相同的类型检测,宏无类型安全性检测。
- 纠正具有副作用的参量处理。联编函数在进入函数体之前对参量的表达式求值。因此,一个有副作用的表达式将是安全的。

对于联编函数的更多信息参见 inline、__inline 节。

为了向下兼容,Microsoft C++保留了所有在 ANSI C 和更早 C++规格中的预处理器功能。

预定义宏

编译器可识别六种预定义的 ANSI C 宏(参见表 1.1),而 Microsoft C++实现提供更多的预定义宏(参见表 1.2)。这些宏不带参量,但不能被重定义。它们的值(除__LINE__和__FILE__外)必须是经过编译的常量。下面列出的一些预定义宏须用多个值来定义,它们的值可在 Visual C++开发环境中选择相应的菜单选项来设置或采用命令行开关。更多的信息参见下表。

表 1.1 ANSI 预定义宏

宏	说明
__DATE__	当前源文件的编译日期。日期是格式为 Mmm dd yyyy 的字符串文字。月份名称 Mmm 与在 TIME.H 中说明的库函数 asctime 产生的日期一样
__FILE__	当前源文件名称。__FILE__扩展为用双引号括起的一个字符串
__LINE__	当前源文件的行号。该行号是一个十进制整型常量。可用一个#line 指令修改
__STDC__	指出与 ANSI C 标准的完全一致性。仅当给出/Za 编译器选项且不编译 C++代码时定义为整型量 1;否则是不确定的
__TIME__	当前文件的最近编译时间。该时间是格式为 hh:mm:ss 的字符串文字
__TIMESTAMP__	当前源文件的最近修改日期。日期是格式为 Ddd Mmm Date hh:mm:ss yyyy 的字符串文字,这里 Ddd 是星期几的简写,Date 是从 1 到 31 的一个整数

表 1.2 Microsoft 特殊预定义的宏

宏	说明
__CHAR_UNSIGNED	缺省 char 类型是无符号的,当指定/J 时定义的
__cplusplus	仅为 C++程序定义

续表

_CPPRTTI	定义为用/GR 编译的代码(允许运行时类型信息)
_CPPUNWIND	定义为用/GX 编译的代码(允许异常处理)
_DLL	指定/MD 或/MDd(多线程 DLL)时定义的
_M_ALPHA	为 DEC ALPHA 平台定义,使用 ALPHA 编译器时定义为 1,若使用另一个编译器时不定义
_M_IX86	为 x86 处理器定义,参见表 1.3
_M_MPPC	为 Power Macintosh 平台定义,缺省为 601(/QP601)参见表 1.4
_M_MRX000	为 MIPS 平台定义, 缺省为 4000(/QMR4000),参见表 1.5
_M_PPC	为 PowerPC 平台定义,缺省为 604(/QP604),参见表 1.6
_MFC_VER	为 MFC 版本定义,为 Microsoft Founndation 类库 4.21 定义为 0x0421,它总是定义的
_MSC_EXTENSIONS	该宏在使用/Ze 编译选项(缺省值)时定义,定义时其值总为 1
_MSC_VER	定义编译器版本,对于 Microsoft Visual C++ 6.0 定义为 1200,它总是定义的
_MT	当指定/MD 或/MDd(多线程 DLL)或/MT 或/MTd(多线程)选项时定义
_WIN32	为 Win32 应用程序而定义。它总是定义的

如下表所示,编译器对反映处理器选项的预处理器标识符产生一个值。

表 1.3 _M_IX86 的值

开发者的选项	命令行选项	返回值
Blend	/GB	_M_IX86=500(缺省值。将来的编译器将给出一个不同的值以影响主处理器)
Pentium	/G5	_M_IX86=500
Pentium Pro	/G6	_M_IX86=600
80386	/G3	_M_IX86=300
80486	/G4	_M_IX86=400

表 1.4 _M_MPPC 的值

开发者的选项	命令行选项	返回值
PowerPC 601	/QP601	_M_MPPC=601(缺省值)
PowerPC 603	/QP603	_M_MPPC=603
PowerPC 604	/QP604	_M_MPPC=604
PowerPC 620	/QP620	_M_MPPC=620

表 1.5 _M_MRX000 的值

开发者选项	命令行选项	返回值
R4000	/QMR4000	_M_MRX000=4000(缺省值)
R4100	/QMR4100	_M_MRX000=4100
R4200	/QMR4200	_M_MRX000=4200

续表

R4400	/QMR4400	_M_MRX000=4400
R4600	/QMR4600	_M_MRX000=4600
R10000	/QMR10000	_M_MRX000=10000

表 1.6 _M_PPC 的值

开发者选项	命令行选项	返回值
PowerPC 601	/QP601	_M_PPC=601
PowerPC 603	/QP603	_M_PPC=603
PowerPC 604	/QP604	_M_PPC=604(缺省值)
PowerPC 620	/QP620	_M_PPC=620

第 2 章　编译指示指令

C 和 C++的每个实现对它的主机或操作系统都支持一些独有的特征。例如，某些程序须对存放数据的存储器区域进行精确的控制,或必须控制特定函数接受参量的方式。#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。

语法

#pragma 语言符号字符串

语言符号字符串是给出特有编译器指令和参量的字符序列。数字符号(#)必须是包含编译指示行中的第一个非空白字符。空白字符可分开数字符号(#)和单词 pragma 。在#pragma 之后,可以编写翻译器作为预处理器语言符号分析的任何文本。#pragma 的参量从属于宏扩展。

如果编译器找到一个不能识别的编译指示,将发出一个警告,但编译将继续。

编译指示可用在条件说明中,以提供新的预处理器功能,或提供定义的实现信息给编译器。C 和 C++编译器可识别下面的编译指示:

alloc_text	comment	init_seg*	optimize
auto_inline	component	inline_depth	pack

续表

bss_seg	data_seg	inline_recursion	pointers_to_members*
check_stack	function	intrinsic	setlocale
code_seg	hdrstop	message	vtordisp*
const_seg	include_alias	once	warning

*仅被 C++编译器支持

C++编译器专有编译指示

以下是 C++编译器专有的编译指示指令：

- init_seg
- pointers_to_members
- vtordisp

init_seg

C++特殊处→
#pragma init_seg({complier/lib/user/ "*section-name*" [,"*func-name*"]})
该指令指定一个影响启动代码执行顺序的关键字或代码段。由于全局静态对象的初始化可能涉及执行代码,因此必须指定创建对象时定义的一个关键字。在动

态连接库(DLL)或需初始化的库中使用 init_seg 编译指示尤其重要。
init_seg 编译指示的选项如下:
complier
该选项保留给 Microsoft C 运行库初始化。这个组中的对象最先被创建。
lib
用于第三方类库供应商的初始化。该组中的对象在 complier 标志之后,其他标记之前创建。
user
用于任何用户。此组对象最后创建。
section_name
允许初始化段的显示规格。在一个用户指定 section-name(段名称)中的对象不能被隐含地创建,但它们的地址可放在以 section_name 命名的段中。
func_name
指定在程序退出时在 exit()地方调用的函数。指定的函数必须与 exit 函数具有相同的特征:
int funcname(void(__cdecl*)(void));
如果你需要延迟初始化过程(例如,在一个 DLL 中),你可以选择显式地指定该段名称。然后必须为每个静态对象调用构造函数。
C++特殊处结束

pointers_to_members

C++特殊处→

#progma pointers_to_members(指针说明,[最一般表示])

该指令指定一个类成员的指针能否在其相关定义之前被说明,且用于控制该指针尺寸和解释该指针需要的代码。你可以把一个 pointers_to_members 编译指示放入你的源文件中替换/vmx 编译器选项。

指针说明参量指定你在一个关联函数定义之前还是之后说明了一个成员的指针。指针说明参量是以下两个符号之一:

参量	说明
full_generality	生成安全、但常常并非最优的代码。如果在关联类定义之前说明任何成员的指针,可使用 full_generality。该参量通常使用最一般表示参量指定的指针表示形式。等同于/vmg 选项
best_case	为所有成员指针使用最佳情况(best__case)表示生成安全的最优代码。使用该参量是需在定义一个类的成员指针说明之前定义此类。其缺省值为 best_case

最一般表示参量说明了在转换单元中,编译器能够安全地引用任何指向类成员的指针的最小指针表示。该参量取如下值之一:

参量	说明
single_inheritance	最一般表示是单继承的,即一个成员函数的指针。对于其中说明了一个指向成员指针的一个类定义,若其继承模式说明为多重的或虚拟的,将导致错误

续表

multiple_inheritance	最一般表示是多重继承的,即一个成员函数的指针。对于其中说明了一个指向成员指针的一个类定义,若其继承模式是虚拟的,将导致错误
vitual_inheritance	最一般表示是虚拟继承,即一个成员函数的指针。该函数不会导致错误。当使用#pragma pointers_to_members(full_generality)时这是个缺省参量

C++特殊处结束

vtordisp

C++特殊处→
#pragma vtordisp({on|off})
该指令允许增加隐含的 vtordisp 构造函数/析构函数替换成员。vtordisp 编译指示只使用虚基类的代码。若一个派生类重迭一个从虚拟基类继承的虚拟函数,且如果派生类的一个构造函数或析构函数调用那个使用该虚拟基类指针的函数,则编译器可能将增加的隐含“vtordisp”域到有虚拟基的类中。
vtordisp 编译指示会影响其后类的分布。/Vd0 或/Vd1 选项指定了对于完全模式的相同动作。指定 off 将抑制隐含的 vtordisp 成员。指定缺省值 on,将在需要的位置打开它们。Vtordisp 指令仅在类的构造/析构函数在用 this 指针指向的对象处不可能调用虚拟函数时关闭。
#pragma vtordisp(off)

```
class GetReal:virtual public{...};
#pragma vtordisp(on)
```
C++特殊处结束

C 和 C++编译器编译指示

以下是为 C 和 C++编译器定义的编译指示：

alloc_text	component	init_seg*	optimize
auto_inline	const_seg	inline_depth	pack
bss_seg	data_seg	inline_recursion	pointers_to_members*
check_stack	function	intrinsic	setlocale
code_seg	hdrstop	message	vtordisp*
comment	include_alias	once	warning

* 仅被 C++编译器支持

alloc_text

```
#pragma alloc_text(“文本段”,函数 1,...)
```
该指令用于命名指定的函数定义将要驻留的代码段。该编译指示对已命名的函数必须出现在一个函数说明符和该函数定义之间。
alloct_text 编译指示并不处理 C++成员函数或重载函数。它仅用于以 C 连接方

式说明的函数,这指的是用 extern“C”连接规格说明的函数。如果你试图将此编译指示用于非 C++连接的函数,将产生一个编译错误。
由于不支持使用 __based 的函数地址,指定段位址需要使用 alloc_text 编译指示,以文本段指定的名称应包括在双引号间。
alloc_text 编译指示必须出现在指定的函数说明之后,这些函数的定义之前。
一个 alloc_text 编译指示中的函数引用必须在此编译指示的同一模块中定义。如果未这样做,且一个未定义的函数随后被编译到一个不同的文本段,则这个错误可能找得到,也可能找不到。虽然该程序一般会正常运行,但该函数不会分配到预期的段中。
alloc_text 的其他限制如下:

- 它不能用在一个函数的内部。
- 它必须在已说明的函数之后和已定义的函数之前使用。

auto_inline

#pragma auto_inline([{on|off}])
排除自动内联扩展的候选者中指定为 off 的区域中定义的函数。为了使用 auto_inline 编译指示,把它放在一个函数定义之前或立即之后(不在该函数定义之内)。在看到该编译指示之后的第一个函数定义处,该编译指示发生作用。
编译指示 auto_inline 不能应用于显式内联函数。

bss_seg

#pragma data_seg(["*section-name*"[,"*section-class*"]])
指定未初始化数据的缺省段。data_seg 编译指示处理初始化或未初始化数据有相同的作用。在某些情况下,你可以使用 bss_seg 通过把所有未初始化数据放在一个段中来加速加载的时间。
#pragma bss_seg("MY_DATA")
导致#pragma 语句后未初始化的数据分配到一个名称为 MY_DATA 的段中。
用 bss_seg 编译指示分配的数据不会保留关于它的位置的任何信息。
第二个参量 section_class 用于与 Visual C++之前的版本兼容,现在已被忽略。

check_stack

#pragma check_stack([{on|off}])
#pragma check_stack{+|-}
该指令在 off(或 -)选项时指示编译器关闭栈搜索。在 on(或+)选项指定时,打开搜索。
若无参量,栈搜索就按缺省情况处理。在看到该编译指示之后第一个定义的函数处发生作用。栈搜索既不是宏的一部分,也不是产生的内联函数的一部分。
如果未赋予一个参量给 check_stack 编译指示,栈检查将还原成在命令行中说明的行为,有关更多的信息参见“编译器参考”。#pragma check_stack 和/Gs 选项的交互关系参见表 2.1。

表 2.1 使用 check_stack 编译指示

语法	是否用/Gs 选项编译	行为
#pragma check_stack()或 #pragma check_stack	是	关闭其后函数的栈检查
#pragma check_stack()或 #pragma check_stack	否	打开其后函数的栈检查
#pragma check_stack(on) 或 #pragma check_stack +	是或否	打开其后函数的栈检查
#pragma check_stack(off) 或 #pragma check_stack -	是或否	关闭其后函数的栈检查

code_seg

#pragma code_seg(["*section-name*"[,"*section-class*"]])
该指令用于指定一个分配函数的代码段。code_seg 编译指示指定了函数的缺省段。
你可以有选择性地指定类和段名。使用没有 section-name 字符串的#pragma code_seg 可在编译开始时将其复位。

const_seg

#pragma const_seg(["*section-name*"[,"*section-class*"]])
该指令用于指定对于常量数据的缺省段。data_seg 编译指示对所有数据具有相同作用。你可以用此指令将你的所有常量数据放入一个只读段中。
#pragma const_seg("MY_DATA")
导致该指令将#pragma 语句后的常量数据放入一个名称为 MY_DATA 的段里。
使用 const_seg 编译指示分配的数据不会保留有关它的位置的任何信息。
第二个参数 section-class 用于与 Visual C++ 2.0 版之前的版本兼容,现在已可忽略。

comment

#pragma comment(*comment-type*,[*commentstring*])
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。comment-type 是下面五种说明的预定义标识符之一,它们指出了注释记录的类型。任选的 commentstring 是给一些注释类型提供额外信息的字符串文字。由于 commentstring 是一个字符串文字,因此它必须遵循对于字符串文字的诸如转义字符、嵌入或引号标记(")以及合并的所有规则。

complier

该选项将编译器的名称和版本号放入对象文件中。这个注释记录被链接器忽略,如果你为这个记录类型给出一个 commentstring 参量,该编译器将产生一个警告

信息。

exestr

该选项将 commentstring 放入对象文件中。在连接时,该字符串被置入可执行文件中。
该字符串并不与可执行文件同时加载到存储器,但它可用在文件中寻找可打印字符串的程序找到。这个注释记录类型的一个用途是把版本号或类似信息嵌入到一个可执行文件中。

lib

该选项将一个库搜索记录放入对象文件。该注释类型必须带有一个 commentstring 参数。这个参数包含你想要的链接器搜索的库的名称(有可能包含路径)。由于在对象文件中该库名称在缺省的库搜索记录之前,所以链接器搜索该库就象你在命令行中命名了它一样。你可以把多个库搜索记录放在同一个源文件中,每个记录在对象文件中都以其在源文件中出现的同样顺序出现。

linker

该选项将一个链接器选项放入对象文件中。可以用该注释类型指定一个链接器选项,用于取代在 Project Setting 对话框中 Link 选项卡上放入该选项。例如,你可以指定/include 选项来强行包括一个符号:

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

user

该选项将一个一般的注释放入对象文件中。commentstring 参量包含了该注释的文本。这个注释记录被链接器忽略。

以下编译指示导致链接器在连接时搜索 EMAPI.LIB 库。 该连接器首先在当前工作目录中搜索,随后在 LIB 环境变量说明的路径中搜索。

```
#pragma comment(lib,"emapi")
```

以下编译指示导致编译器把编译器的名称和版本号放入对象文件:

```
#pragma comment(complier)
```

注意:对于一个带 commentstring 参量的注释, 你可以在任何要使用一个字符串文字的地方使用宏,让这个宏扩展为一个字符串文字。你也可以把任何一个字符串文字的任何组合与扩展为字符串文字的宏合并起来,例如:下面的语句是可以接受的:

```
#pragma comment(user,"Compiled on" __DATA__ "at" __TIME__)
```

component

```
#pragma component(browser,{on|off}[,references[,name]])
#pragma component(minrebuild,on|off)
```

该指令用于控制源文件内的浏览信息或依赖信息的集合。

browser

你可以打开或关闭集合,并可以在收集信息时指定忽略的特定名称。

用 on 或 off 选项控制前面编译指示的浏览信息的集合。例如:
#pragma component(browser,off)
该指令让编译器停止收集浏览信息。
注意:用此指令打开浏览信息的集合,浏览信息必须先用 Project Settings 对话框或命令行打开。
references 选项可带也可不带 name 参量。不带 name 参量的 references 选项用于打开或关闭引用的集合(但此时继续收集其它浏览信息)。例如:
#pragma component(browser,off,references)
该指令使编译器停止收集引用信息。
带 name 和 off 参量的 references 选项,用于防止浏览信息窗口中出现对 name 的引用。使用这种语法可忽略你不感兴趣的名称和类型,并且可缩短浏览信息的尺寸。例如:
#pragma component(browser,off,references,DWORD)
忽略该点之前对于 DWORD 的引用。但你可用 on 选项把对于 DWORD 引用的集合重新打开。
#pragma component(browser,on,references,DWORD)
这是恢复对 name 引用集合唯一的方式;用此方式你可以显式地打开任何你已经关闭的 name。
为了防止预处理器把 name 展开(如把 NULL 扩展为 0),将它加上引号:
#pragma component(browser,off,references,"NULL")

Minimal Rebuild

Visual C++的 minimal rebuild(最小重建)特性需要编译器创建和存储依赖信息的 C++类,这将占用磁盘空间。为了节省磁盘空间,你可以在任何你不需收集依赖信息的时候仅用#pragma component(minirebuild,off),例如,在不变的头文件中。在不变的类后插入#pragma component(minrebuild,on)可重新打开依赖信息。
有关更多的信息参见 Enable Minimal Rebuild(/Gm)编译器选项。

data_seg

#pragma data_seg(["*section-name*"[,"*section-class*"]])
该指令指定数据的缺省段。例如:
#pragma data_seg("MY_DATA")
将#pragma 语句后分配的数据放在以名称为 MY_DATA 的段里。
使用 data_seg 编译指示分配的数据不会保留关于它的位置的任何信息。
第二个参量 section-class 用于与 Visual C++ 2.0 之前的版本兼容,现在已可忽略。

function

#pragma function(*function1*[,*function2*,...])
该指令指定对在生成的编译器编译指示参量表中指定调用的函数。如果你使用 intrinsic 编译指示(或/Oi)告诉编译器生成内在函数(被生成为内联代码,而非

函数调用的内在函数),就能用 function 编译指示来显式地强制调用一个函数。当一个函数编译指示出现时,它在第一个包含一个指定的内在函数的函数定义处发生作用,这个作用持续到源文件的结尾或直到一个说明这种相同的内在函数的编译指示出现为止。在全局层时,该 function 编译指示只能用在一个函数外。对于具有内部形式的函数表,参见#pragma intrinsic。

hdrstop

#pragma hdrstop[("*filename*")]

该指令用于控制预编译头文件的工作方式。filename 是预编译头文件使用或创建的名称(根据指定选项/Yu 或/Yc 决定)。如果 filename 不包含路径说明,预编译头文件将被假定在与源文件相同的目录中。当指定/YX 自动预编译头文件选项时,所有 filename 都被忽略。

当采用/YX 或/Yc 编译时,一个 C 或 C++文件包含一个 hdrstop 编译指示,该编译器将把编译状态存入到编译指示的位置。该编译指示之后任何代码的编译状态都不存储。

hdrstop 编译指示不可能出现在一个头文件中。它必须出现在源文件中。这指的是,它不能出现在任何数据、函数说明或函数定义中。

注意:除非/YX 选项或无文件名的/Yu 或/Yc 选项被指定,否则 hdrstop 编译指示将被忽略。

该指令使用 filename 命名编译状态存储的预编译头文件。hdrstop 和 filename 之间的一个空白是任选的。在 hdrstop 编译指示中说明的文件名称是一个字符串,而且必须服从 C 或 C++字符串的约束。尤其重要的是必须将其置于括号中,

如下例所示：
```
#pragma hdrstop("c:\projects\include\myinc.pch")
```
预编译头文件的名称由下列规则决定,顺序如下：
1. /Fp 编译器选项的参量。
2. #pragma hdrstop 的 filename 参量。
3. 以.PCH 为扩展名的源文件的基名称。

include_alias

```
#pragma include_alias("long_filename","short_filename")
#pragma include_alias(<long_filename>,<short_filename>)
```
该指令指定 short_filename 作为 long_filename 的别名。某些文件系统允许比 8.3 FAT 文件系统限定更长的文件名。编译器不能够简单地把更长的头文件名截短为 8.3 格式,因为这种更长的头文件名的开始八个字符可能不是唯一的。只要编译器遇到 long_filename 字符串,就用 short_filename 替换,并且代替查找 short_filename 头文件。这个编译指示必须出现在相应的#include 指令之前,例如：
```
//这两个文件的开头八个字符不是唯一的
#pragma include_alias("AppleSystemHeaderQuickdraw.h","quickdra.h")
#pragma include_alias("AppleSystemHeaderFruit.h","fruit.h")

#pragma include_alias( "GraphicsMenu.h", "gramenu.h")
```

```
#include "AppleSystemHeaderQuickdraw.h"
#include "AppleSystemHeaderFruit.h"
#include "GraphicsMenu.h"
```
无论是拼写,还是双引号或尖括号的用法,被搜索的别名都必须符合规格。include_alias 编译指示在该文件名中处理简单的字符串匹配,而在其它的文件名中是无效的。例如,给出如下指令:
```
#pragma include_alias("mymath.h","math.h")
#include "./mymath.h"
#include "sys/mymath.h"
```
没有别名使用(替换)被执行,因为头文件字符串并不匹配。用作/Yu、/Yc 和/YX 编译器选项参量的头文件名,或 hdrstop 编译指示的头文件名也不能被替换。例如,若你的源文件包含下列指令:
```
#include <AppleSystemHeaderStop.h>
```
相应的编译器选项应该是:
```
/YcAppleSystemHeaderStop.h
```
你可以用 include_alias 编译指示把任何头文件名映射为另一个。例如:
```
#pragma include_alias("api.h", "c:\version1.0\api.h")
#pragma include_alias(<stdio.h>, <newstdio.h>)
#include "api.h"
#include <stdio.h>
```
不要把双引号内的文件名和尖括号内的文件名相混淆,例如,对于上面给出的两个#pragma include_alias 指令,编译器在下面的#include 指令中不执行任何替

换：
```
#include <api.h>
include "stdio.h"
```
此外，下面的指令将导致错误：
```
#pragma  include_alias(<header.h>,"header.h") //错误
```
注意在错误信息中给出的文件名，或作为预先定义的__FILE__宏的值，是替换执行之后的文件的名称，例如，下面指令后：
```
#pragma include_alias("VeryLongFileName.H","myfile.h" )
#include "VeryLongFileName.H"
```
在 VERYLONGFILENAME.H 中的一个错误将导致如下错误消息：
```
myfile.h(15) : error c2059 :  syntax error
```
同时注意传递性是不支持的。如下指令中：
```
#pragma include_alias( "one.h", "two.h" )
#pragma include_alias( "two.h", "three.h")
#include "one.h"
```
编译器搜索的是文件 TWO.H 而不是 THREE.H。

inline_depth

```
#pragma inline_depth([0...255])
```
该指令通过控制一系列函数调用能被扩展的次数(从 0 到 255 次)，来控制内联扩展可发生的次数。这个编译指示控制标记为 inline 和 _inline 的函数的联编或在 /Ob2 选项下已经自动联编的函数。

inline_depth 编译指示控制一序列函数调用能被扩展的次数,例如,若联编深度为 4,且若 A 调用 B,B 调用 C,三个调用都将内联扩展。但如果最近的内联扩展为 2,只有 A 和 B 被扩展,C 保留为一个函数调用。
为使用这个编译指示,你必须设置/Ob 编译器选项为 1 或 2。使用这个编译指示的深度设置在该编译指示指令后的第一个函数处生效。如果你没有在圆括号中指定一个值,inline_depth 将设置联编深度为缺省值 8。
联编深度在扩展时只减不增。若联编深度为 6,且在扩展时,预处理器遇到一个联编深度值为 8 的 inline_depth 编译指示,该深度仍保持为 6。
联编深度 0 禁止联编扩展;联编深度 255 对联编扩展无限制,若使用一个未指定其值的编译指示,则将使用其缺省值。

inline_recursion

#pragma inline_recursion([{on|off}])
该指令控制直接或相互递归函数调用的联编扩展。该指令用于控制标记为 inline 和 __inline 的函数,或编译器在 Ob2 选项下自动扩展的函数。这个编译指示的用法需要一个设置为 1 或 2 的/Ob 编译器选项。inline_recursion 的缺省状态是 off。这个编译指示只是在其出现之后的第一个函数处起作用,并且不会影响该函数的定义。
inline_recursion 编译指示控制递归函数如何被扩展。如果 inline_recursion 关闭,且若一个联编函数调用其自身(直接或间接),该函数只被扩展一次。若 inline_recursion 为打开状态,则该函数将被扩展多次直到其达到 inline_depth 的值,或达到其容量值的限制。

intrinsic

#pragma intrinsic(*function1*[,*function2*,...])

该指令指定对于在编译指示参量表中说明的函数的调用是内在的。编译器生成象联编代码的内在函数,而并不像函数调用那样。具有内在格式的库函数如下表。当遇到一个 intrinsic 编译指示时,它在一个包含指定的内在函数的第一个函数定义处发生作用,其作用延续到源文件的结尾,或直到一个说明相同的内在函数的 function 编译指示出现为止。

intrinsic 编译指示只能用在一个函数定义之外(全局级)。

以下函数具有内在格式:

_disable	_outp	fabs	strcmp
_enable	_outpw	labs	strcpy
_inp	_rotl	memcmp	strlen
_inpw	_rotr	memcpy	
_lrotl	_strset	memset	
_lrotr	abs	strcat	

使用内在函数的程序更快,因为它们没有函数调用的额外开销,但由于产生的额外代码,它们会更大一些。

注意:_alloca 和 setjmp 函数总是联编生成的,这种行为不会受 intrinsic 编译指示的影响。

以下所列的浮点函数没有真正的内在格式。但它们有这样的版本:将参量直接送入浮点芯片,而不是将它们压入程序栈中:

acos	cosh	pow	tanh
asin	fmod	sinh	

当你指定/Oi 和/Og 编译器选项(或任何包括/Og、/Ox、/O1 和/O2 的选项)时,以下浮点函数有真正的内在格式:

atan	exp	log10	sqrt
atan2	log	sin	tan
cos			

你可以使用/Op 或/Za 编译器选项重迭真正的内在浮点选项的生成。这种情况下,该函数被生成为库例程,该例程直接把参量送入浮点芯片,而不是把它们压入程序栈。

message

#pragma message(*messagestring*)

该指令不终止编译,直接把一个字符串文字送到标准输出。message 编译指示的典型用法是在编译时显示信息消号。以下代码段使用 message 编译指示在编译时显示一个消息:

```
#if _M_IX86==500
#pragma message("Pentium processor build")
#endif
```

messagestring 参数可以是可扩展为一个字符串文字的宏,并且你可以以任何组合方式用字符串文字将这些宏合并起来。例如,下列语句显示了被编译的文件名以及该文件最后改动的日期和时间:

```
#pragma message("Compiling  "__FILE__  )
#pragma message("Last modified on " __TIMESTAMP__)
```

once

#pragma once
该指令指定该编译指示驻留的文件将只在一次建立中被编译器包括(打开)一次。该编译指示的一种普通用法如下:

```
//header.h
#pragma once
//接着是你的 C 或 C++代码
```

optimize

#pragma optimize("[*optimization-list*]",{on|off})
仅用于专业和企业版本的特征:代码优化只被 Visual C++专业版和企业版支持。更多的信息参见 Microsoft Visual C++联机编辑。
该指令指定在函数基中执行的优化。optimize 编译指示必须出现在一个函数之外,并且在编译指示出现后定义的第一个函数处产生作用。on 和 off 参量可以打开或关闭在 Optimization-list 中指定的选项。
optimization-list 可以是 0 或在表 2.2 中给出参数。

表 2.2 "优化"编译指示的参数

参量	优化类型
a	假定无别名
g	允许全局优化
p	提高浮点相容性
s 或 t	指定机器码的短或快序列
w	假定无交叉函数调用的别名
y	生成程序堆栈中的框架指针

这些是采用/O编译器选项的相同字母,例如:

```
#pragma optimize("atp", on)
```

用空字符串("")使用 optimize 编译指示是该指令的一种特殊形式,它可关闭所有的优化或恢复它们的原有(缺省的)设置。

```
#pragma optimize("",off)
.
.
.
#pragma optimize("", on)
```

pack

```
#pragma pack([n])
```

该指令指定结构和联合成员的紧凑对齐。而一个完整的转换单元的结构和联合

的紧凑对齐由/Zp 选项设置。紧凑对齐用 pace 编译指示在数据说明层设置。该编译指示在其出现后的第一个结构或联合说明处生效。该编译指示对定义无效。当你使用#pragma pack(*n*)时,这里 *n* 为 1、2、4、8 或 16。第一个结构成员之后的每个结构成员都被存储在更小的成员类型或 n 字节界限内。如果你使用无参量的#pragma pack,结构成员被紧凑为以/Zp 指定的值。该缺省/Zp 紧凑值为/Zp8。

编译器也支持以下增强型语法:

#pragma pack([[{push|pop},][标识符,]][*n*])

若不同的组件使用 pack 编译指示指定不同的紧凑对齐,这个语法允许你把程序组件组合为一个单独的转换单元。

带 push 参量的 pack 编译指示的每次出现将当前的紧凑对齐存储到一个内部编译器堆栈中。编译指示的参量表从左到右读取。如果你使用 push,则当前紧凑值被存储起来;如果你给出一个 n 的值,该值将成为新的紧凑值。若你指定一个标识符,即你选定一个名称,则该标识符将和这个新的的紧凑值联系起来。

带一个 pop 参量的 pack 编译指示的每次出现都会检索内部编译器堆栈顶的值,并且使该值为新的紧凑对齐值。如果你使用 pop 参量且内部编译器堆栈是空的,则紧凑值为命令行给定的值,并且将产生一个警告信息。若你使用 pop 且指定一个 n 的值,该值将成为新的紧凑值。

若你使用 pop 且指定一个标识符, 所有存储在堆栈中的值将从栈中删除,直到找到一个匹配的标识符,这个与标识符相关的紧凑值也从栈中移出,并且这个仅在标识符入栈之前存在的紧凑值成为新的紧凑值。如果未找到匹配的标识符,将使用命令行设置的紧凑值,并且将产生一个一级警告。缺省紧凑对齐为 8。

pack 编译指示的新的增强功能让你编写头文件,确保在遇到该头文件的前后的紧凑值是一样的。

```
/* File name: include1.h
*/
#pragma pack(push,enter_include1)
/* 你的包括文件代码...  */
#pragma pack(pop, enter_include1)
/* include1.h 结束 */
```

在上面的例子中,当前紧凑值与标识符 enter_include1 联系起来,并被压入头文件的项中。头文件末尾的 pack 编译指示删除所有可能出现在头文件中的干预紧凑值,并且删除与 enter_include1 相关的紧凑值。因此确保该头文件的前后的紧凑值是相同的。

这种新功能也允许你使用代码,例如头文件,它可以使用 pack 编译指示设置不同于在你的代码中设置的紧凑值的紧凑对齐:

```
#pragma pack(push,before_include1)
#include "include1.h"
#pragma pack( pop,before_include1)
```

在上面的例子中,对于出现在 include.h 中的紧凑值的任何变化,你的代码是受到保护的。

setlocale

#pragma setlocale("*locale_string*")

该指令在翻译宽字符常量和字符串文字时定义其场所(国家和语言)。由于用于转换多字节字符为宽位字符的算法可能由于场所或编译而不同,该执行文件在不同的场所运行也可能不同。这个编译指示提供了在编译时给出目标场所的方法。这保证了宽字符串以正确的格式进行存储。缺省的 locale__string(场所字符串)是“C”。“C”场所将每个该串中的字符映射为一个 wchar_t(unsigned short)型的值。

warning

#pragma warning(*warning-specifier*:*warning-number-list*
[,*warning-specifier*:*warning-number-list*...])
#pragma warning(push[,*n*])
#pragma warning(pop)
该指令允许选择性地改变编译器的警告消息。
warning-specifier 可以是如下值之一:

警告指示符	含义
once	只显示一次指定的消息
default	将缺省的编译器行为应用于指定的消息
1,2,3,4	把给定的警告级应用于指定的警告消息
disable	不发出指定警告消息
error	作为错误报告指定的警告

warning-number-list(警告编号表)可以包含任何警告编号。在相同的编译指示指令中可指定多个选项如下:

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```
其功能想当于:
```
#pragma warning( disable : 4507 34 )  // 禁止警告消息 4507 和 34.
#pragma warning( once : 4385 )// 仅发出警告 4385 一次
#pragma warning( error : 164 )// 作为一个错误报告警告 164
```
对于那些与代码生成有关,且大于 4699 的警告编号来说,这个 warning 编译指示仅当放在函数定义外时有效。如果警告编号大于 4699 或用在函数体内,则忽略该编译指示。下面的例子指出了禁止 warning 编译指示的正确位置,且随后恢复一个代码生成警告消息的产生。
```
int a;
#pragma warning( disable : 4705 )
void func()
{
    a;
}
#pragma warning( default : 4705 )
```
warning 编译指示也支持以下语法:
```
#pragma warning( push[,n])
#pragma warning( pop)
```
这里 n 代表警告级(1 到 4)。
编译指示 warning(push)存储所有警告的当前警告状态。编译指示 warning(push,*n*)存储所有警告的当前警告状态并设置全局警告级为 *n*。

编译指示 warning(pop)将上次压入栈的警告状态弹出,push 和 pop 之间警告状态的任何变化都将被取消。考虑这个例子:

```
#pragma warning(push)
#pragma warning(disable : 4705)
#pragma warning(disable : 4706)
#pragma warning(disable : 4707)
// 某些代码
#pragma warning(pop)
```

在这段代码的末尾,pop 恢复所有警告状态(包括 4705、4706 和 4707)为它在代码起始处的警告状态。

当你编写头文件时,可以用 push 和 pop 以确保对于用户造成的警告状态的变化,不会影响头部的正确编译。通常在头部的起始处使用 push,在末尾处使用 pop。例如,有一个在警告级 4 未彻底编译的头部。以下代码将警告级改为 3,然后在头部的末尾恢复原来的警告级:

```
#pragma warning(push,3)
//说明/定义
#pragma warning(pop)
```

附录　语　法　总　结

本附录描述了预处理器的规范语法。它包括了在第 1 章“预处理器”和第 2 章“编译指示指令”中讨论的预处理指令和操作符。
本附录包括以下主题：

- 定义
- 约定
- 预处理器语法

定　义

终结符是语法定义中的终点。其它任何解决方案都是不可能的。终结符包括保留字集和用户定义标识符。
非终结符在语法中是位置占用者。在本语法总结中大多数在其它地方定义。定义可以是递归的。下面的非终结符定义“Microsoft Visual C++ 6.0 参考库”的“Microsoft Visual C++ 6.0 语言参考手册”卷的附录 A“语法总结”中：
常量、常量表达式、标识符、关键字、运算符、标点
一个任选的组件可用下标 opt 给定。例如，下面给出了一个括在大括号中的可选表达式：

{*expression* $_{opt}$}

约　定

这些约定对语法的不同组件使用不同的字体。这些符号和字体如下：

属性	说明
非终结符	斜体类型指出非终结符
#include	英文字母中的终结符是必须输入的文字保留词和符号。这个上下文中的字母大小写是敏感的
$_{opt}$	非终结符后跟 $_{opt}$ 表示是任选的
缺省字体	这个字体中描述或列出的字符可以用作语句中的终结符

一个非终结符后跟一个冒号(:)引入其定义。分行列出另一种定义。

预处理器语法

#define 标识符 语言符号字符串 $_{opt}$
#define 标识符[(标识符 $_{opt}$,...,标识符 $_{opt}$)]　语言符号字符串 $_{opt}$
defined(标识符)
defined 标识符
#include "路径规格"

```
#include <路径规格>
#line 数字序列 "文件名"opt
#undef 标识符
#error 语言符号字符串
#pragma 语言符号字符串
条件的:
    if 部分 elif 部分opt else 部分opt endif 行
if 部分:
    if 行 文本
if 行:
    #if 常量表达式
    #ifdef 标识符
    #ifndef 标识符
elif 部分:
     elif 行 文本
     elif 部分 elif 行 文本
elif 行:
    #elif 常量表达式
else 部分:
    else 行 文本
else 行:
    #else
```

endif 行：
    #endif
数字序列：
    数字
    数字序列 数字
数字：以下之一
    0 1 2 3 4 5 6 7 8 9
语言符号字符串：
    语言符号组成的字符串
语言符号：
    关键字
    标识符
    常量
    运算符
    标点
文件名：
    合法的操作系统文件名
路径规格：
    合法的文件路径
文本：
    文本的任何序列
注意：以下的非终结符在“Microsoft Visual C++ 6.0 语言参考手册”中的附

录 A“语法总结”中解释:常量、常量表达式、标识符、关键字、运算符和标点。