

# //查看预处理结果

cpp 1.c

gcc -E 1.c

# //输出预处理文件

cpp 1.c -o 1.i

gcc -E 1.c -o 1.i

# #include

**include**的两种形式、cpp的搜索路径

#include “stdio.h” //1.源文件所在路径。

//2. -l 选项 所指定的路径（-l 路径）。

//3.环境变量 包含的路径（C\_INCLUDE\_PATH）（export命令）。

//4.预配置 的路径 （cpp -v）。

#include <stdio.h> //1. -l 选项 所指定的路径（-l 路径）。

//2.环境变量 包含的路径（C\_INCLUDE\_PATH）（export命令）。

//3.预配置 的路径 （cpp -v）。

# #define

宏定义指令 🡪 #define 宏 替换列表。

它的作用就是：定义一个宏，所以我们把它叫做宏定义指令。

## 宏定义中“#”的含义：

将参数 “字符串化”。

eg：

#define A(a) “123”#a”456”

int main()

{

char \*c = A(789); //c指向的字符串就是 “123””789””456”。

}

## 宏定义中“##”的含义：

以下代码：

#define A(a) qa

在使用该条宏定义指令时，

eg:

int main()

{

char \*b = A(b); //正确的结果应该是qb，但是实际结果是qa。

}

**//原因是【大小写字母 0123456789 “ ‘ $ \_】 🡪参数和这些字符连接，参数不替换。**

#define A(a,b) ab

**//同上，这样参数连接在一起的，预处理器不进行参数替换。**

**为了处理这种连接不替换的现象，引入了“##”，这样就可以实现既让他们连接在一起，又让他们参数替换。（也可以将“##”称为“预处理器粘合剂”）。**

**※当参数的前面有“#”或者“##”运算符时，参数不展开。**

# 空白字符

空白字符是以下字符的统称：

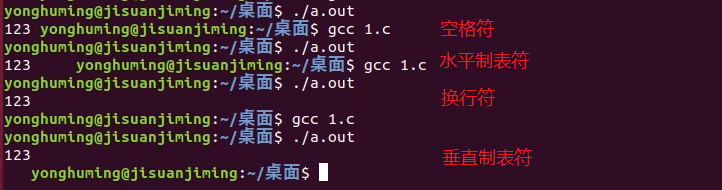
空格符 🡪 \x20 （注意：空格符是可打印字符）。

水平制表符 🡪 \t

换行符 🡪 \n

垂直制表符 🡪 \v

换页符 🡪 \f



# 预处理之指定路径

1. 头文件在源文件所在路径。
2. 将头文件所在路径添加到环境变量中（C\_INCLUDE——PATH）。
3. 通过 -I 指定路径（绝对路径/相对路径）。

gcc 1.c -I xxx -I xxx

1. 在文件包含指令中指定路径（绝对路径/相对路径）。

# #define 可变形参

eg:

#define A(…) \_\_VA\_ARGS\_\_ // \_\_VA\_ARGS\_\_ 就是可变参数对应的形参。

#define B(…) \_\_VA\_ARGS\_\_,a,b

#define C(…) a,b,\_\_VA\_ARGS\_\_

#define D(…) a,b,\_\_VA\_ARGS\_\_,\_\_VA\_ARGS\_\_

int main()

{

A(1, 2, 3); //得到的结果为 1, 2, 3 。

B(1, 2, 3); //得到的结果为 1, 2, 3, a, b。

C(1, 2, 3, 4, 5); //得到的结果为 1, 2, 3, 4, 5。

D(1, 2, 3); //得到的结果为 1, 2, 3, 4, 5, 3, 4, 5。

}

# #define、预定义宏

1. 预定义宏：由C实现预定义好的宏，我们可以在程序中直接使用。
2. C标准规定了一些预定义宏。
3. 一部分是强制要求定义的、一部分是可选的。
4. 除此之外，C实现本身也会定义一些宏提供我们使用。
5. 预定义宏不能被重定义、取消定义。

eg:

\_\_DATE\_\_ //源文件被预处理的日期（字符串）。

\_\_TIME\_\_ //源文件被预处理的时间（字符串）。

\_\_FILE\_\_ //当前文件的名称（字符串）。

\_\_LINE\_\_ //当前的行号（整型常量）。

\_\_STDC\_\_ //(整型常量)、用来表明C实现是符合C标准的。

\_\_STDC\_HOSTED\_\_ //如果是宿主实现，则为整型常量1；如果不是宿主实现，则为整型常量0。

\_\_STDC\_VERSION\_\_ //当前C实现所支持的标准版的版本（整型常量 – long int）。

# 条件包含指令

//使CPP可以根据条件，选择性的保留或删除源文件中的某些内容。

//以 #if/#ifdef/#ifndef开始、以#endif结束、中间配合着#elif/#else。

//#elif 🡪类似于 else if。

## #if/#endif

eg:

#if 条件表达式 //若条件表达式为真，则保留二者中间的内容，若为假，则删除中间的内容。

…（中间内容）

#endif

**注意：**条件表达式可以为常量、条件表达式、宏（未定义的标识符会被替换为0）等（但必须是**整型常量表达式**）。

## #defined

//defined 🡪预处理运算符 🡪预处理表达式：defined A。

//如果操作数是一个宏，那么表达式的值为=1，否则表达式的值为0。

eg:

#if defined A //预处理表达式 🡪字面意思是如果定义了宏A则保留。

xxxxxx

xxxxxx

#endif

## #ifdef/#ifndef

#if defined A == #ifdef A

#if !defined A == #ifndef A

所以，我们可以把#ifdef/#idndef理解为#if的一个变种。

# #undef

1. 用于取消已定义的宏。
2. 取消一个没有被定义的宏没有任何效果，但这是允许的，编译器不会报错或报警。

# #line

#line 十进制整数

#line 十进制整数 字符串

十进制整数 🡪 改变预定义宏 \_\_LINE\_\_ 的值（设置#line指令的下一行的行号）。

字符串 🡪 改变预定义宏 \_\_FILE\_\_ 的值。

# #error

预定义指令 🡪 抛出错误

eg:

#ifndef A

#error “错误原因：没有定义宏A”

#endif

int main(void)

{

}

//运行结果为： error：#error “错误原因：没有定义宏A”

#error “错误原因：没有定义宏A”

# #pragma

1、#pragma 🡪 编译指示

2、我们可以通过#pragma指令，来对编译器进行一些设置（命令行参数/选项）。

3、它并非指某一个指令，而是一个指令的集合。

4、它并非是一个标准的指令的集合，不同的C实现有着自己的#pragma指令集。

5、因此，#pragma指令是不具有可移植性的。

6、CPP将忽略它不认识的#pragma指令。

**拓展：**

<https://blog.csdn.net/lmhuanying1012/article/details/78549763>

每种C和C++的实现支持对其宿主机或操作系统唯一的功能。例如，一些程序需要精确控制超出数据所在的储存空间，或着控制特定函数接受参数的方式。#pragma指示使每个编译程序在保留C和C++语言的整体兼容性时提供不同机器和操作系统特定的功能。编译指示被定义为机器或操作系统特定的，并且通常每种编译程序是不同的。

语法：

#pragma token\_string

“token\_string”是一系列字符用来给出所需的特定编译程序指令和参数。数字符号“#”必须是包含编译指令的行中第一个非空白字符；而空白字符可以隔开数字符号“#”和关键字“pragma”。在#pragma后面，写任何翻译程序能够作为预处理符号分析的文本。#pragma的参数类似于宏扩展。

如果编译程序发现它不认得一个编译指示，它将给出一个警告，可是编译会继续下去。

为了提供新的预处理功能，或者为编译程序提供由实现定义的信息，编译指示可以用在一个条件语句内。C和C++编译程序可以识别下列编译程序指令。

|  |  |  |  |
| --- | --- | --- | --- |
| alloc\_text | comment | init\_seg\* | optimize |
| auto\_inline | component | inline\_depth | pack |
| 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 |

\*仅用于C++编译程序。

**1 alloc\_text**

#pragma alloc\_text( "textsection", function1, ... )

命名特别定义的函数驻留的代码段。该编译指示必须出现在函数说明符和函数定义之间。

alloc\_text编译指示不处理C++成员函数或重载函数。它仅能应用在以C连接方式说明的函数——就是说，函数是用extern "C"连接指示符说明的。如果你试图将这个编译指示应用于一个具有C++连接方式的函数时，将出现一个编译程序错误。

由于不支持使用\_\_based的函数地址，需要使用alloc\_text编译指示来指定段位置。由textsection指定的名字应该由双引号括起来。

alloc\_text编译指示必须出现在任何需要指定的函数说明之后，以及这些函数的定义之前。

在alloc\_text编译指示中引用的函数必须和该编译指示处于同一个模块中。如果不这样做，使以后一个未定义的函数被编译到一个不同的代码段时，错误会也可能不会被捕获。即使程序一般会正常运行，但是函数不会分派到应该在的段。

alloc\_text的其它限制如下：

它不能用在一个函数内部。

它必须用于函数说明以后，函数定义以前。

**2 auto\_inline**

#pragma auto\_inline( [{on | off}] )

当指定off时将任何一个可以被考虑为作为自动嵌入扩展候选的函数排除出该范围。为了使用auto\_inline编译指示，将其紧接着写在一个函数定义之前或之后（不是在其内部）。该编译指示将在其出现以后的第一个函数定义开始起作用。auto\_inline编译指示对显式的inline函数不起作用。

**3 bss\_seg**

#pragma data\_seg( ["section-name"[, "section-class"] ] )

为未初始化数据指定缺省段。data\_seg编译指示除了工作于已初始化数据而不是未初始化的以外具有一样的效果。在一些情况下，你能使用bss\_seg将所有未初始化数据安排在一个段中来加速你的装载时间。

#pragma bss\_seg( "MY\_DATA" )

将导致把#pragma语句之后的未初始化的数据安排在一个叫做MY\_DATA的段中。

用bss\_seg编译指示分配的数据不包含任何关于其位置的信息。

第二个参数section-class是用于兼容2.0版本以前的Visual C++的，现在将忽略它。

**4 check\_stack**

#pragma check\_stack([ {on | off}] )

#pragma check\_stack{+ | –}

如果指定off（或者“-”）指示编译程序关闭堆栈探测，或者指定on（或“+”）打开堆栈探测。如果没有给出参数，堆栈探测将根据默认设置决定。该编译指示将在出现该指示之后的第一个函数开始生效。堆栈探测既不是宏和能够生成嵌入代码函数的一部分。

如果你没有给出check­\_stack编译指示的参数，堆栈检查将恢复到在命令行指定的行为。详细情况见编译程序参考。#pragma check\_stack和/Gs选项的互相作用情况在表2.1中说明。

表 2.1 使用check\_stack编译指示

|  |  |  |
| --- | --- | --- |
| 编译指示 | 用/Gs选项编译？ | 行为 |
| #pragma check\_stack()或#pragma check\_stack | 是 | 后续的函数关闭堆栈检查 |
| #pragma check\_stack()或#pragma check\_stack | 否 | 后续的函数打开堆栈检查 |
| #pragma check\_stack(on)或#pragma check\_stack(+) | 是或者否 | 后续的函数打开堆栈检查 |
| #pragma check\_stack(off)或#pragma check\_stack(-) | 是或者否 | 后续的函数关闭堆栈检查 |

**5 code\_seg**

#pragma code\_seg( ["section-name"[,"section-class"] ] )

指定分配函数的代码段。code\_seg编译指示为函数指定默认的段。你也能够像段名一样指定一个可选的类名。使用没有段名字符串的#pragma code\_seg将恢复分配到编译开始时候的状态。

**6 const\_seg**

#pragma const\_seg( ["section-name"[, "section-class"] ] )

指定用于常量数据的默认段。data\_seg编译指示除了可以工作于所有数据以外具有一样的效果。你能够使用该编译指示将你的常量数据保存在一个只读的段中。

#pragma const\_seg( "MY\_DATA" )

导致在#pragma语句后面的常量数据分配在一个叫做MY\_DATA的段中。

用const\_seg编译指示分配的数据不包含任何关于其位置的信息。

第二个参数section-class是用于兼容2.0版本以前的Visual C++的，现在将忽略它。

**7 comment**

#pragma comment( comment-type [, commentstring] )

将描述记录安排到目标文件或可执行文件中去。comment-type是下面说明的五个预定义标识符中的一个，用来指定描述记录的类型。可选的commentstring是一个字符串文字值用于为一些描述类型提供附加的信息。因为commentstring是一个字符串文字值，所以它遵从字符串文字值的所有规则，例如换码字符、嵌入的引号（"）和联接。

**7-1 compiler**

在目标文件中放置编译程序名和版本号。该描述记录被连接程序忽略。如果你为这个记录类型提供一个commentstring参数，编译程序将生成一个警告。

**7-2 exestr**

将commentstring放置到目标文件中去。在连结时，这个字符串再被放到可执行文件去中。当可执行文件被装载时这个字符串不会被装入内存，然而，它可以被一个能够在文件中搜索可打印字符串的程序找到。该描述记录的一个用处是在可执行文件中嵌入版本号或者类似的信息。

**7-3 lib**

将一个库搜索记录放置到目标文件中去。该描述类型必须有包含你要连接程序搜索的库名（和可能的路径）的commentstring参数。因为在目标文件中该库名先于默认的库搜索记录，所以连接程序将如同你在命令行输入这些库一样来搜索它们。你可以在一个源文件中放置多个库搜索记录，每个记录将按照它们出现在源文件中的顺序出现在目标文件中。

**7-4 linker**

在目标文件中放置连接程序选项。你可以用这个描述类型指定连接程序选项来代替在Project Setting对话框中Link页内的选项。例如，你可以指定/include选项以强迫包含一个符号：

#pragma comment(linker, "/include:\_\_mySymbol")

**7-5 user**

在目标文件中包含一个普通描述记录。commentstring参数包含描述的文本。该描述记录将被连接程序忽略。

下面的编译指示导致连接程序在连接时搜索EMAPI.LIB库。连接程序首先在当前工作目录然后在LIB环境变量指定的路径中搜索。

#pragma comment( lib, "emapi" )

下面的编译指示导致编译程序将其名字和版本号放置到目标文件中去。

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

#pragma comment( compiler )

注意，对于具有commentstring参数的描述记录，你可以使用其它用作字符串文字量的宏来提供宏扩展为字符串文字量。你也能够联结任何字符串文字量和宏的组合来扩展成为一个字符串文字量。例如，下面的语句是可以接受的：

#pragma comment( user, "Compiled on " \_\_DATE\_\_ " at " \_\_TIME\_\_ )

**8 component**

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

#pragma component( minrebuild, on | off )

从源文件内控制浏览信息和依赖信息的收集。

**8-1 浏览信息（Browser）**

你可以将收集打开或关闭，你也可以指定收集时忽略特别的名字。

使用on或off在编译指示以后控制浏览信息的收集。例如：

#pragma component(browser, off)

终止编译程序收集浏览信息。

注意，为了用这个编译指示打开浏览信息的收集，必须先从Project Setting对话框或者命令行允许浏览信息。

references选项可以有也可以没有name参数。使用没有name参数的references选项将打开或者关闭引用信息的收集（然而继续收集其它浏览信息）。例如：

#pragma component(browser, off, references)

终止编译程序收集引用信息。

使用有name和off参数的references选项将阻止从浏览信息窗口中出现引用到的名字。用这个语法将忽略你不感兴趣的名字和类型从而减少浏览信息文件的大小。例如：

#pragma component(browser, off, references, DWORD)

从这一点以后忽略DWORD的引用。你能够用on恢复DWORD的引用收集：

#pragma component(browser, on, references, DWORD)

这是唯一的方法可以恢复收集指定名字的引用，你必须显式地打开任何你关闭的名字。

为了防止预处理程序扩展名字（就像扩展NULL到0），用引号括起来：

#pragma component(browser, off, references, "NULL")

**8-2 最小化重建（Minimal Rebuild）**

Visual C++的最小化重建功能要求编译程序创建并保存需要大量磁盘空间的C++类依赖信息。为了节省磁盘空间，你能够在你不需要收集依赖信息时使用#pragma component(minrebuild,off)，例如，没有改变过头文件。在未修改过的类之后插入#pragma component(minrebuild,on)重新打开依赖信息。

详见Enable Minimal Rebuild(/Gm)编译程序选项。

**9 data\_seg**

#pragma data\_seg( ["section-name"[, "section-class"] ] )

指定数据的默认段。例如：

#pragma data\_seg( "MY\_DATA" )

导致在#pragma语句后分配的数据保存在一个叫做MY\_DATA的段中。

用data\_seg编译指示分配的数据不包含任何关于其位置的信息。

第二个参数section-class是用于兼容2.0版本以前的Visual C++的，现在将忽略它。

**10 function**

#pragma function( function1 [, function2, ...] )

指定必须生成对编译指示中参数列表内函数的调用。如果你使用intrinsic编译指示（或者/Oi）来告诉编译程序生成内含函数（内含函数如同嵌入代码一样生成，不作为一个函数调用），你能够用function编译指示显式地强迫函数调用。当遇到一个function编译指示，它将在其后面遇到的第一个包含有内含函数的函数定义处生效。其持续作用到源文件的尾部或者出现对同一个内含函数指定intrinsic编译指示。function编译指示只能用于函数外——在全局层次。

为了列出具有内含形式的函数表，参见#pragma intrinsic。

**11 hdrstop**

#pragma hdrstop [( "filename" )]

控制预编译头文件的工作方式。filename是要使用或者创建（依赖于是否指定了/Yu或/Yc）预编译头文件的名字。如果 filename不包括一个指定路径，将假定预编译头文件和源文件处于同一个目录中。当指定自动预编译头文件选项/YX时，所有指定的文件名将被忽略。

如果有/YX或者/Yc选项，而且C或C++文件包含了一个hdrstop编译指示时，编译程序保存编译指示之前的编译状态。编译指示之后的编译状态不被保存。

hdrstop编译选项不能出现在一个头文件内。它只能出现在源文件的文件级，它也不能出现在任何数据或者函数的说明或定义之中。

注意，除非指定没有文件名的/YX选项或者/Yu或/Yc选项，否则hdrstop编译指示将被忽略。

用一个文件名命名要保存编译状态的预编译头文件。在hdrstop和filename之间的空格是可选的。在hdrstop编译指示中的文件名是一个字符串，这样它服从于C或C++的字符串规则。特别的，你必须像下面例子里面显示的用引号括起来。

#pragma hdrstop( "c:\projects\include\myinc.pch" )

预编译头文件的文件名按照如下规则决定，按照优先次序：

/Fp编译程序选项的参数；

由#pragma hdrstop的filename参数；

原文件名的基本文件名加上.PCH扩展名。

**12 include\_alias**

#pragma include\_alias( "long\_filename", "short\_filename" )

#pragma include\_alias( <long\_filename>, <short\_filename> )

指定作为long\_filename别名的short\_filename。一些文件系统允许超出8.3FAT文件系统限制的长头文件名。编译程序不能简单地将长文件名截断为8.3名字，因为长头文件名的前8个字符可能不是唯一的。无论何时编译程序遇到long\_filename串，它代替short\_filename，并且用short\_filename搜索头文件。这个编译指示必须出现在相应的#include指示之前。例如：

// First eight characters of these two files not unique.

#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"

这个别名在搜索时精确匹配，包括拼写和双引号、尖括号。include\_alias编译指示在文件名上执行简单的字符串匹配，不进行其它的文件名验证。例如，给出下列指示：

#pragma include\_alias("mymath.h", "math.h")

#include "./mymath.h"

#include "sys/mymath.h"

并不执行别名替代，因为头文件名字符串没有精确匹配。另外，在/Yu，/Yc和/YX编译程序选项，或hdrstop编译指示中作为参数的头文件名不被替换。例如，如果你的源文件包含下列指示：

#include <AppleSystemHeaderStop.h>

相应的编译程序选项必须是：

/YcAppleSystemHeaderStop.h

你能够用include­\_alias编译指示将任何头文件映射到其它文件。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>

不要混淆用双引号和尖括号括起来的文件名。例如，给出上面的#pragma include\_alias指示时，在下面的#include指示中编译程序不执行替换。

#include <api.h>

#include "stdio.h"

还有，下面的指示将产生一个错误：

#pragma include\_alias(<header.h>, "header.h") // Error

注意，在错误信息中报告的文件名，或者预定义宏\_\_FILE\_\_的值，是执行替换以后的文件名。例如，在下列指示之后：

#pragma include\_alias( "VeryLongFileName.H", "myfile.h" )

#include "VeryLongFileName.H"

文件VeryLongFileName.H产生下列错误信息：

myfile.h(15) : error C2059 : syntax error

还要注意的是不支持传递性。给出下面的指示：

#pragma include\_alias( "one.h", "two.h" )

#pragma include\_alias( "two.h", "three.h" )

#include "one.h"

编译程序将搜索two.h而不是three.h。

**13 init\_seg**

C++特有

#pragma init\_seg({ compiler | lib | user | "section-name" [, "func-name"]} )

指定影响启动代码执行的关键字或代码段。因为全局静态对象的初始化可以包含执行代码，所以你必须指定一个关键字来定义什么时候构造对象。在使用需要初始化的动态连接库（DLL）或程序库时使用init\_seg编译指示是尤其重要的。

init\_seg编译指示的选项有：

**13-1 compiler**

由Microsoft C运行时间库保留。在这个组中的对象将第一个构造。

**13-2 lib**

用于第三方类库开发者的初始化。在这个组中的对象将在标记为构造compiler的对象之后，其它对象之前构造。

**13-3 user**

用于任何其它用户。在这个组中的对象将最后构造。

**13-4 section-name**

允许显式地指定初始化段。在用户指定的section-name中的对象将不会隐式地构造，而它们的地址将会被放置在由section-name命名的段中。

**13-5 func-name**

指定当程序退出时，作为atexit函数调用的函数。这个函数必须具有和atexit函数相同的形式：

int funcname(void (\_\_cdecl \*)(void));

如果你需要延迟初始化，你能够选择指定显式的段名。随后你必须调用每个静态对象的构造函数。

**14 inline\_depth**

#pragma inline\_depth( [0... 255] )

通过控制能够被扩展的一系列函数调用（从0到255次）来控制嵌入函数扩展的发生次数，这个编译指示控制用inline，\_\_inline标记的或在/Ob2选项下能自动嵌入的嵌入函数。

inline\_depth编译指示控制能够被扩展的一系列函数调用。例如，如果嵌入深度是4，并且如果A调用B然后调用C，所有的3次调用都将做嵌入扩展。然而，如果设置的最近一次嵌入深度是2，则只有A和B被扩展，而C仍然作为函数调用。

为了使用这个编译指示，你必须设置编译程序选项/Ob为1或者2。用这个编译指示指定的深度设定在该指示后面的第一个函数开始生效。如果你在括号内不指定一个值，inline\_depth设置嵌入深度到默认值8。

在扩展时，嵌入深度可以被减少而不能被增加。如果嵌入深度是6，同时在扩展过程中预处理程序遇到一个inline\_depth编译指示设置为8，则深度保持为6。

嵌入深度0将拒绝嵌入扩展，深度255将设置在嵌入扩展时没有限制。如果用一个没有指定值的编译指示，则使用为默认值。

**15 inline\_recursion**

#pragma inline\_recursion( [{on | off}] )

控制直接或者相互间的递归函数调用式的嵌入扩展。用这个编译指示控制用inline，\_\_inline标记的或在/Ob2选项下能自动嵌入的嵌入函数。使用这个编译指示需要设置编译程序选项/Ob为1或者2。默认的inline\_recursion状态是off。这个编译指示在出现该编译指示之后第一个函数调用起作用，并不影响函数的定义。

inline\_recursion编译指示控制如何扩展递归函数。如果inline\_recursion是off，并且如果一个嵌入函数调用了它自己（直接的或者间接的），函数将仅仅扩展一次。如果inline\_recursion是on,函数将扩展多次直到达到inline\_depth的值或者容量限制。

**16 intrinsic**

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

指定对在编译指示参数表中函数调用是内含的。编译程序像嵌入代码一样生成内含函数，而不是函数调用。下面列出了具有内含形式的库函数。一旦遇到intrinsic编译指示，它从第一个包含指定内含函数的函数定义开始起作用。作用持续到源文件尾部或者出现包含相同内含函数的function编译指示。intrinsic编译指示只能用在函数定义外——在全局层次。

下列函数具有内含形式：

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

使用内含函数的程序更快，因为它们没有函数调用的额外代价，然而因为有附加的代码生成，可能比较大。

注意，\_alloca和setjmp函数总是内含的，这个行为不受intrinsic编译指示影响。

下列浮点函数没有内含形式。然而它们具有直接将参数通过浮点芯片传送而不是推入程序堆栈的版本。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| acos | asin | cosh | fmod | pow | sinh |
| tanh |  |  |  |  |  |

当你同时指定/Oi和/Og编译程序选项（或者任何包含/Og，/Ox，/O1和/O2的选项）时下列浮点函数具有真正的内含形式。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| atan | exp | log10 | sqrt | atan2 | log |
| sin | tan | cos |  |  |  |

你可以用编译程序选项/Op或/Za来覆盖真内含浮点选项的生成。在这种情况下，函数会像一般库函数一样被生成，同时直接将参数通过浮点芯片传送而不是推入程序堆栈。

**17 message**

#pragma message( messagestring )

不中断编译，发送一个字符串文字量到标准输出。message编译指示的典型运用是在编译时显示信息。

下面的代码段用message编译指示在编译过程中显示一条信息：

#if \_M\_IX86 == 500

#pragma message( "Pentium processor build" )

#endif

messagestring参数可以是一个能够扩展成字符串文字量的宏，并且你能够用字符串文字量和宏的任何组合来构造。例如，下面的语句显示被编译文件的文件名和文件最后一次修改的日期和时间。

#pragma message( "Compiling " \_\_FILE\_\_ )

#pragma message( "Last modified on " \_\_TIMESTAMP\_\_ )

**18 once**

#pragma once

指定在创建过程中该编译指示所在的文件仅仅被编译程序包含（打开）一次。该编译指示的一种常见用法如下：

//header.h

#pragma once

// Your C or C++ code would follow:

**19 optimize**

仅在专业版和企业版中存在

#pragma optimize( "[optimization-list]", {on | off} )

代码优化仅有Visual C++专业版和企业版支持。详见Visual C++ Edition。

指定在函数层次执行的优化。optimize编译选项必须在函数外出现，并且在该编译指示出现以后的第一个函数定义开始起作用。on和off参数打开或关闭在optimization-list指定的选项。

optimization-list能够是0或更多个在表2.2中给出的参数：

表 2.2   optimize编译指示的参数

|  |  |
| --- | --- |
| 参数 | 优化类型 |
| a | 假定没有别名。 |
| g | 允许全局优化。 |
| p | 增强浮点一致性。 |
| s 或 t | 指定更短或者更快的机器代码序列。 |
| w | 假定在函数调用中没有别名。 |
| y | 在程序堆栈中生成框架指针。 |

这些和在/O编译程序选项中使用的是相同的字母。例如：

#pragma optimize( "atp", on )

用空字符串（""）的optimize编译指示是一种特别形式。它要么关闭所有的优化选项，要么恢复它们到原始（或默认）的设定。

#pragma optimize( "", off )

#pragma optimize( "", on )

**20 pack**

#pragma pack( [ n] )

指定结构和联合成员的紧缩对齐。尽管用/Zp选项设定整个翻译单元的结构和联合成员的紧缩对齐，可以用pack编译指示在数据说明层次设定紧缩对齐。从出现该编译指示后的第一个结构或者联合说明开始生效。这个编译指示不影响定义。

当你使用#pragma pack(n)，其中n是1，2，4，8或者16，第一个以后的每个结构成员保存在较小的成员类型或者n字节边界上。如果你使用没有参数的#pragma pack，结构成员将被紧缩到由/Zp指定的值。默认的/Zp紧缩的大小是/Zp8。

编译程序还支持下面的增强语法：

#pragma pack( [ [ { push | pop}, ] [ identifier, ] ] [ n ] )

该语法允许你将使用不同紧缩编译指示的组件合并到同一个翻译单元内。

每次出现有push参数的pack编译指示将保存当前的紧缩对齐值到一个内部的编译程序堆栈。编译指示的参数列表从左向右读取。如果你使用了push，当前紧缩值被保存。如果你提供了一个n值，这个值将成为新的紧缩值。如果你指定了一个你选定的标示符，这个标示符将和新的紧缩值关联。

每次出现有pop参数的pack编译指示从内部编译程序堆栈顶部取出一个值并将那个值作为新的紧缩对齐。如果你用了pop，而内部编译程序堆栈是空的，对齐值将从命令行得到，同时给出一个警告。如果你用了pop并指定了n的值，那个值将成为新的紧缩值。如果你用了pop并指定了一个标示符，将移去所有保存在堆栈中的的值直到匹配的找到匹配的标示符，和该标示符关联的紧缩值也被从堆栈中移出来成为新的紧缩值。如果没有找到匹配的标示符，将从命令行获取紧缩值并产生一个1级警告。默认的紧缩对齐是8。

pack编译指示的新的增强功能允许你编写头文件保证在使用头文件之前和其后的紧缩值是一样的：

/\* File name: include1.h

\*/

#pragma pack( push, enter\_include1 )

/\* Your include-file code ... \*/

#pragma pack( pop, enter\_include1 )

/\* End of include1.h \*/

在前面的例子中，进入头文件时将当前紧缩值和标示符enter\_include1关联并推入，被记住。在头文件尾部的pack编译选项移去所有在头文件中可能遇到的紧缩值并移去和enter\_include1关联的紧缩值。这样头文件保证了在使用头文件之前和其后的紧缩值是一样的。

新功能也允许你在你的代码内用pack编译指示为不同的代码，例如头文件设定不同的紧缩对齐。

#pragma pack( push, before\_include1 )

#include "include1.h"

#pragma pack( pop, before\_include1 )

在上一个例子中，你的代码受到保护，防止了在include.h中的任何紧缩值的改变。

**21 pointers\_to\_members**

C++特有

#pragma pointers\_to\_members(pointer-declaration, [most-general-representation] )

指定是否能够在相关类定义之前说明一个指向类成员的指针，并且用于控制指针的大小和解释指针的代码。你能够在你的源代码中使用pointers\_to\_members编译知识来代替/vmx编译程序选项。

pointer-declaration参数指出是否在相关函数定义之前或其后你已经说明了一个指向成员的指针。pointer-declaration参数是下面两个符号之一：

|  |  |
| --- | --- |
| 参数 | 说明 |
| full\_generality | 生成安全的，但是有时不能优化的代码。如果有一些指向成员的指针在相关类定义之前说明，你要用full\_generality。这个参数总是使用由most-general-representation指定的指针表示方式。 |
| best\_case | 对于所有指向成员的指针用最佳的表示方式生成安全的，优化的代码。需要在说明一个指向类成员指针之前定义类。默认是best\_case。 |

most-general-representaion参数指出在一个翻译单元中编译程序能够安全引用任何指向类成员指针的最小指针表示方式。这个参数可以是下列之一：

|  |  |
| --- | --- |
| 参数 | 说明 |
| single\_inheritance | 最普通的表示方式是单继承，指向成员函数。如果用于指向具有多重或者虚拟继承方式类成员的指针，将产生一个错误。 |
| multi\_inheritance | 最普通的表示方式是多重继承，指向成员函数。如果用于指向具有虚拟继承方式类成员的指针，将产生一个错误。 |
| virtual\_inheritance | 最普通的表示方式是虚拟继承，指向成员函数。不会产生错误。当使用#pragma pointers\_to\_members (full\_generality)时这是默认的参数。 |

**22 setlocale**

#pragma setlocale( "locale-string" )

定义用于翻译宽字符常数和字符串文字量时用的地区（国家和语言）。由于用于从多字节字符转换到宽字符的算法根据地区或者由于在运行可执行程序不同的地方进行编译而不同，这个编译指示提供一种在编译时指定目标地区的方式。这保证宽字符字符串将以正确的格式保存。默认的locale-string是“C”。“C”地区将字符串中的每个字符作为wchar\_t（即unsigned int）映射其值。

**23 vtordisp**

C++特有

#pragma vtordisp({on | off} )

允许隐藏的附加vtordisp构造函数/析构函数替换成员。vtordisp编译指示仅能够用于具有虚拟基类的代码。如果派生类从一个虚拟基类重载了一个虚拟函数，并且如果派生类的构造函数或析构函数用指向虚拟基类的指针调用了这个函数，编译程序将根据虚拟基类在类中引入一个附加的隐藏“vtordisp”域。

vtodisp编译选项影响它后面的类布局。/vd0和/vd1选项为整个模块指定了相同的行为。指定off将禁止隐藏的vtordisp成员，指定on（默认）将在它们需要的时候允许vtordisp。仅在不可能出现类的构造函数和析构函数通过this指针调用其指向对象中的虚拟函数时才关闭vtordisp。

#pragma vtordisp( off )

class GetReal : virtual public { ... };

#pragma vtordisp( on )

**24 warning**

#pragma warning( warning-specifier : warning-number-list [,warning-specifier : warning-number-list...] )

#pragma warning( push[ , n ] )

#pragma warning( pop )

允许有选择地修改编译程序警告信息的行为。

warning-specifier能够是下列值之一：

|  |  |
| --- | --- |
| warning-specifier | 含义 |
| once | 只显示指定信息一次。 |
| default | 对指定信息应用默认的编译程序选项。 |
| 1,2,3,4 | 对指定信息引用给定的警告等级。 |
| disable | 不显示指定信息。 |
| error | 对指定信息作为错误显示。 |

warning-number\_list能够包含任何警告编号。如下，在一个编译指示中可以指定多个选项：

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

这等价于：

#pragma warning( disable : 4507 34 ) // Disable warning messages

                                             // 4507 and 34.

#pragma warning( once : 4385 )        // Issue warning 4385

                                             // only once.

#pragma warning( error : 164 )        // Report warning 164

                                             // as an error.

对于那些关于代码生成的，大于4699的警告标号，warning编译指示仅在函数定义外时有效。如果指定的警告编号大于4699并且用于函数内时被忽略。下面例子说明了用warning编译指示禁止、然后恢复有关代码生成警告信息的正确位置：

int a;

#pragma warning( disable : 4705 )

void func()

{

    a;

}

#pragma warning( default : 4705 )

warning编译指示也支持下面语法：

#pragma warning( push [ ,n ] )

#pragma warning( pop )

这里n表示警告等级（1到4）。

warning(push)编译指示保存所有警告的当前警告状态。warning(push,n)保存所有警告的当前状态并将全局警告等级设置为n。

warning(pop)弹出最后一次推入堆栈中的警告状态。任何在push和pop之间改变的警告状态将被取消。考虑下面的例子：

#pragma warning( push )

#pragma warning( disable : 4705 )

#pragma warning( disable : 4706 )

#pragma warning( disable : 4707 )

// Some code

#pragma warning( pop )

在这些代码的结束，pop恢复了所有警告的状态（包括4705，4706和4707）到代码开始时候的样子。

当你编写头文件时，你能用push和pop来保证任何用户修改的警告状态不会影响正常编译你的头文件。在头文件开始的地方使用push，在结束地方使用pop。例如，假定你有一个不能顺利在4级警告下编译的头文件，下面的代码改变警告等级到3，然后在头文件的结束时恢复到原来的警告等级。

#pragma warning( push, 3 )

// Declarations/ definitions

#pragma warning( pop )

# \_\_pragma 预处理运算符

pragma(“XXX”) 🡪 #pragma xxx

# # 空指令/无效指令

这个指令只会被CPP简单的删除，没有任何效果。

#

#define A xxx //着重的效果（因人而异）

#