在编写程序的时候，我们经常要用到#pragma 指令来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。

1. #pragma message 指令

message 能够在编译消息输出窗口中输出相应的消息，这对于源代码信息的控制非常重要的。格式如下：

#pragma message（“消息文本”）

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候进行检查，假设我们希望判断自己有没有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!"。

2. #pragma code_seg 指令

格式如下：

```
#pragma code_seg([[{push |
pop},][identifier,]]["segment-name",]["segment-class"])
```

该指令用来指定函数在.obj 文件中存放的节，观察.obj 文件可以使用 VC 自带的 dumpbin 命令行程序，函数在.obj 文件中默认的存放节为.text 节。

（1）如果 code_seg 没有带参数的话，则函数存放在.txt 节中；

（2）push(可选参数)：将一个记录放到内部编译器的堆栈中，可选参数（记录名)可以为一个标识符或者节名;pop(可选参数)将一个记录从堆栈顶端弹出，该记录可以为一个标识符或者节名；

（3）identifier（可选参数)：当使用 push 指令时，为压入堆栈的记录指派的一个标识符，当该标识符被删除的时候和其相关的堆栈中的记录将被弹出堆栈；

（4）"segment-name"（可选参数)：表示函数存放的节名；

例如：

```
//默认情况下，函数被存放在.txt 节中
void func1() { // stored in .txt }

//将函数存放在.my_data1 节中
#pragma code_seg(".my_data1")
void func2() { // stored in my_data1 }

//r1 为标识符,将函数放入.my_data2 节中
#pragma code_seg(push, r1, ".my_data2)
void func3() { // stored in my_data2 }

int main() { }
```

3. #pragma once 指令

格式如下：

#pragma once

这是一个比较常用的指令，只要在头文件的最开始加入这条指令就能够保证头文件只被被编译一次。

（1）#pragma once 是编译相关的，就是说这个编译系统上能用，但在其他编译系统不一定可以，也就是说移植性差，不过现在基本上已经是每个编译器都有这个定义了。

（2）#ifndef / #define / #endif 是 C++语言相关的，它是通过 C++语言中的宏定义来避免头文件被多次编译的。所以在所有支持 C++语言的编译器上都是有效的，如果写的程序要跨平台，最好使用这种方式。

关于#pragma once 和#ifndef / #define / #endif 的详细区别参见 8.。

4. #pragma hdrstop 指令

格式如下：

#pragma hdrstop

它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(Borland C++ Builder)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。

5. #pragma warning 指令

格式如下：

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

#pragma warning(push[,n])

#pragma warning(pop)

主要用到的警告表示有如下几个：

once：只显示一次(警告/错误等)消息；

default：重置编译器的警告行为到默认状态；

1,2,3,4：四个警告级别；

disable：禁止指定的警告信息；

error：将指定的警告信息作为错误报告。

【例示】

```
#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
```

另外，#pragma warning 也支持如下格式

```
#pragma warning( push [ ,n ] ) //这里 n 代表一个警告等级(1---4)
#pragma warning( pop )
#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

【例示】

```
#pragma warning( push )
#pragma warning( disable : 4705 )
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
#pragma warning( pop ) //重新保存所有的警告信息(包括 4705，4706 和 4707)
```

在使用标准 C++进行编程的时候经常会得到很多的警告信息，而这些警告信息都是不必要的提示，所以我们可以使用#pragma warning(disable:4786)来禁止该类型的警告。

在 VC 中使用 ADO 时也会得到不必要的警告信息，这个时候我们可以通过 #pragma warning(disable:4146)来消除该类型的警告信息。

6. #pragma comment 指令

格式如下：

#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(注释类型)：可以指定为五种预定义的标识符的其中一种。

五种预定义的标识符分别如下：

（1）compiler：将编译器的版本号和名称放入目标文件中，本条注释记录将被编译器忽略，如果你为该记录类型提供了 commentstring 参数，编译器将会产生一个警告。

例如：#pragma comment(compiler)

（2）exestr：将 commentstring 参数放入目标文件中，在链接的时候这个字符串将被放入到可执行文件中，当操作系统加载可执行文件的时候，该参数字符串不会被加载到内存中。但是，该字符串可以被 dumpbin 之类的程序查找出并打印出来，你可以用这个标识符将版本号码之类的信息嵌入到可执行文件中！

（3）lib：这是一个非常常用的关键字，用来将一个库文件链接到目标文件中。它可以帮我们连入一个库文件。

例如：#pragma comment(lib, "user32.lib") //将 user32.lib 库文件加入到本工程中

（4）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")

你可以在程序中设置下列链接选项

/DEFAULTLIB

/EXPORT

/INCLUDE

/MERGE

/SECTION

（5）user：将一般的注释信息放入目标文件中，commentstring 参数包含注释的文本信息,这个注释记录将被链接器忽略。

例如：#pragma comment(user, "Compiled on " __DATE__ " at " __TIME__)

7. #pragma pack 指令

用于控制对齐。

例如：

```
#pragma pack(push)
#pragma pack(1)
struct s_1
{
char szname[1];
int a;
};
#pragma pack(pop)
struct s_2
{
char szname[1];
int a;
};
```

则

```
printf("s_1 size : %d\n", sizeof(struct s_1));
```

和

```
printf("s_2 size : %d\n", sizeof(struct s_2));
```

分别得到 5 和 8。

8. #pragma once 和 #ifndef / #define / #endif 的区别

两者的共同点都是为了避免同一个文件被 include 多次，但是各有千秋。

在能够支持这两种方式的编译器上，二者并没有太大的区别，但是两者仍然还是有一些细微的区别

方式一：

```
#ifndef __SOMEFILE_H__
#define __SOMEFILE_H__
... ... // 一些声明语句
#endif
```

优点：

#ifndef 由语言支持，移植性好。它依赖于宏名字不能冲突，这不光可以保证同一个文件不会被包含多次，也能保证内容完全相同的两个文件不会被不小心同时包含。

另外，为了保证不同头文件中的宏名不冲突，故采取类似于_ABC_H_的取名方式。其中，abc.h 为当前头文件名。

缺点：

如果不同头文件的宏名不小心“撞车”，可能就会导致头文件明明存在，编译器却硬说找不到声明的状况。

方式二：

```
#pragma once
```

... ... // 一些声明语句

优点：

#pragma once 由编译器提供保证：同一个文件不会被包含多次。注意这里所说的“同一个文件”是指物理上的一个文件，而不是指内容相同的两个文件。于是不必再费劲想个宏名了，当然也就可以避免宏的名字冲突问题了。

缺点：

如果某个头文件有多份拷贝，本方法不能保证他们不被重复包含。

综上，一般可以这样处理：

```
# infndef XX
# define XX
# if _MSC_VER > 1000
# pragma once
# endif
      .
      .
# endif
```

注意：_MSC_VER 是出于版本兼容性考虑，定义 Defines the compiler version. Defined as 1200 for Microsoft Visual C++ 6.0. Always defined.

【注】由于#ifndef 方式可以通过前面介绍的特殊的宏的取名方式来避免名称冲突问题，于是其缺点也就不复存在了，进而#ifndef 方式就更常用了。