<https://blog.csdn.net/phenixyf/article/details/52993336>

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。

一、#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。

二、#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。

三、#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。

四、#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。

五、#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。

六、#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)。

七、#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")

八、#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。  
  
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。  
  
2、如何避免内存对齐的影响？  
那么，能不能既达到提高性能的目的，又能节约一点空间呢？有一点小技巧可以使用。比如我们可以将上面的结构改成：  
struct TestStruct2  
{  
   char c1;  
   char c2;  
   short s;  
   int i;  
};  
  
这样一来，每个成员都对齐在其自然边界上，从而避免了编译器自动对齐。在这个例子中，sizeof(TestStruct2)的值为8。这个技巧有一个重要的作用，尤其是这个结构作为API的一部分提供给第三方开发使用的时候。第三方开发者可能将编译器的默认对齐选项改变，从而造成这个结构在你的发行的DLL 中使用某种对齐方式，而在第三方开发者哪里却使用另外一种对齐方式。这将会导致重大问题。  
比如，TestStruct1 结构，我们的DLL 使用默认对齐选项，对齐为c1 00000000, s 00000002, c2 00000004, i 00000008，同时sizeof(TestStruct1)的值为12。  
  
而第三方将对齐选项关闭，导致c1 00000000, s 00000001, c2 00000003, i 00000004，同时sizeof(TestStruct1)的值为8。  
  
除此之外我们还可以利用#pragma pack（）来改变编译器的默认对齐方式（当然一般编译器也提供了一些改变对齐方式的选项，这里不讨论）。  
  
使用指令#pragma pack (n)，编译器将按照n 个字节对齐。  
使用指令#pragma pack ()，编译器将取消自定义字节对齐方式。  
  
在#pragma pack (n)和#pragma pack ()之间的代码按n 个字节对齐。但是，成员对齐有一个重要的条件,即每个成员按自己的方式对齐.也就是说虽然指定了按n 字节对齐,但并不是所有的成员都是以n 字节对齐。其对齐的规则是,每个成员按其类型的对齐参数(通常是这个类型的大小)和指定对齐参数(这里是n 字节)中较小的一个对齐，即：min( n, sizeof( item )) 。并且结构的长度必须为所用过的所有对齐参数的整数倍,不够就补空字节。看如下例子：  
#pragma pack(8)  
struct TestStruct4  
{  
   char a;  
   long b;  
};  
struct TestStruct5  
{  
   char c;  
   TestStruct4 d;  
   long long e;  
};  
#pragma pack()  
  
问题：  
A)  
sizeof(TestStruct5) = ?  
  
B)  
TestStruct5 的c 后面空了几个字节接着是d?  
TestStruct4 中,成员a 是1 字节默认按1 字节对齐,指定对齐参数为8,这两个值中取1,a  
  
按1 字节对齐;成员b 是4 个字节,默认是按4 字节对齐,这时就按4 字节对齐,所以sizeof(TestStruct4)应该为8;TestStruct5 中,c 和TestStruct4 中的a 一样,按1 字节对齐,而d 是个结构,它是8 个字节,它  
按什么对齐呢?对于结构来说,它的默认对齐方式就是它的所有成员使用的对齐参数中最大的一个, TestStruct4 的就是4.所以,成员d 就是按4 字节对齐.成员e 是8 个字节,它是默认按8字节对齐,和指定的一样,所以它对到8 字节的边界上,这时,已经使用了12 个字节了,所以又添加了4 个字节的空,从第16 个字节开始放置成员e.这时,长度为24,已经可以被8(成员e 按8字节对齐)整除.这样,一共使用了24 个字节.内存布局如下（\*表示空闲内存，1 表示使用内存。单位为1byete）：  
a b  
TestStruct4 的内存布局：1\*\*\*,1111,  
  
c  
TestStruct4.a TestStruct4.b d  
TestStruct5 的内存布局： 1\*\*\*, 1\*\*\*, 1111, \*\*\*\*，11111111  
  
这里有三点很重要：  
首先，每个成员分别按自己的方式对齐,并能最小化长度。  
其次，复杂类型(如结构)的默认对齐方式是它最长的成员的对齐方式,这样在成员是复杂类型时,可以最小化长度。  
然后，对齐后的长度必须是成员中最大的对齐参数的整数倍,这样在处理数组时可以保证每一项都边界对齐。  
  
补充一下,对于数组,比如:char a[3];它的对齐方式和分别写3 个char 是一样的.也就是说它还是按1 个字节对齐.如果写: typedef char Array3[3];Array3 这种类型的对齐方式还是按1个字节对齐,而不是按它的长度。  
  
但是不论类型是什么,对齐的边界一定是1,2,4,8,16,32,64....中的一个。  
  
另外，注意别的#pragma pack 的其他用法：  
#pragma pack(push) //保存当前对其方式到packing stack  
#pragma pack(push,n) 等效于  
#pragma pack(push)  
#pragma pack(n) //n=1,2,4,8,16 保存当前对齐方式，设置按n 字节对齐

#pragma pack(pop) //packing stack 出栈，并将对其方式设置为出栈的对齐方

http://c.biancheng.net/cpp/html/469.html