**1、-> 在c语言中是什么意思？**

****

**2、c语言中的星号“\*”**



3、**#ifdef**

**条件编译**命令最常见的形式为：

|  |  |
| --- | --- |
| 12345 | #ifdef 标识符//程序段1#else//程序段2#endif |

它的作用是：当标识符已经被定义过(一般是用#define命令定义)，则对程序段1进行编译，否则编译程序段2。

其中#else部分也可以没有，即：

|  |  |
| --- | --- |
| 123 | #ifdef 标识符//程序段1#endif |

这里的“程序段”可以是语句组，也可以是命令行。这种条件编译可以**提高C源程序的通用性**。如果一个C源程序在不同计算机系统上运行，而不同的计算机又有一定的差异。例如，我们有一个数据类型，在Windows平台中，应该使用long类型表示，而在其他平台应该使用float表示，这样往往需要对源程序作必要的修改，这就降低了程序的通用性。可以用以下的条件编译：

|  |  |
| --- | --- |
| 12345 | #ifdef WINDOWS#define MYTYPE long#else#define MYTYPE float#endif |

如果在Windows上编译程序，则可以**在程序的开始加上**

|  |  |
| --- | --- |
| 1 | #define WINDOWS |

这样则编译下面的命令行：

|  |  |
| --- | --- |
| 1 | #define MYTYPE long |

如果在这组条件编译命令之前未出现以下命令行：

|  |  |
| --- | --- |
| 1 | #define WINDOWS |

则预编译后程序中的MYTYPE都用float代替。这样，源程序可以不必作任何修改就可以用于不同类型的计算机系统。当然以上介绍的只是一种简单的情况，可以根据此思路设计出其它的条件编译

**4、#ifndef：**这是"if not defined"的简写，是宏定义的一种，它是可以根据是否已经定义了一个变量来进行分支选择，一般用于调试等等。实际上确切的说这应该是预处理功能中三种（**宏定义，文件包含和条件编译**）中的第三种----条件编译。

**#define x //定义一个宏**

**...**

**#endif**

**//C语言在对程序进行编译时，会先根据[预处理命令](http://baike.baidu.com/view/1334643.htm%22%20%5Ct%20%22_blank)进行“预处理”。C语言[编译系统](http://baike.baidu.com/view/3279760.htm%22%20%5Ct%20%22_blank)包括预处理，编译和链接等部分。**

**#ifndef x**//先测试x是否被宏定义过

**#define x**

程序段1 //如果x没有被宏定义过，定义x，并[编译程序](http://baike.baidu.com/view/454895.htm%22%20%5Ct%20%22_blank)段 1

**#else**

程序段2 //如果x已经定义过了则编译程序段2的语句，“忽视”程序段 1。

**#**endif**//**终止**if**

**5、#if：**#if和#endif是一组同时使用的，叫做条件编译指令。#if与#define、#include等指令一样是由预处理器这个强大的工具处理的，预处理器可以在编译前处理c程序。

使用条件编译指令，如果满足#if后面的条件，就编译#if和#endif之间的程序段，否则不编译。

例如：

**#define LEN\_MAX 3**

**#define LenMax 6**

**#if LEN\_MAX <LenMax**

**#undef LEN\_MAX**

**#define LEN\_MAX LenMax**

**#endif**

定义了LEN\_MAX 和LenMax，如果LEN\_MAX <LenMax成立为真，则取消LEN\_MAX的定义，重新定义LEN\_MAX，也就是说LEN\_MAX等于6。这时预处理器会保留中间程序（#if与#endif行会消失），让编译器编译中间行。

如果LEN\_MAX <LenMax不成立，预处理器会将删除#if到#endif中的程序（包括#if与#endif行），则编译器会看不到中间的程序。

这样的处理不会在目标代码中占用空间，减少程序运行时间。它可以用于程序调试使用。

**6、#undef：取消以前定义的宏定义；取消在先前程序中对预处理器的定义。**

例子1：

**#include <iostream.h>**

**#include<string.h>**

**#define MAX 5**

**#undef MAX**

**void main()**

**{**

**char name[MAX]="abcd"; //只能用abcd,否则会提示说超出长度,原因是由于"\0"字符**

**cout<<"MAX = "<<MAX<<endl;**

**for(int i=0;i<MAX;i++)**

**cout<<name<<" "<<endl;**

**}**

得到如下错误消息：

未定义符号 'MAX'

初始值设定项太多

在此程序中对 MAX 定义后又取消了定义。编译该程序时报错。由于未定义 MAX，char name[MAX] 的数组大小不可用，因此编译器发出第二个错误消息。

例子2：

**#define TEST\_A 1**

**#define TEST\_CLASS\_A clase T1**

**#include "TEST.h"**

**#undef TEST\_A**

**#undef TEST\_CLASS\_A**

在文件#include "TEST.h" 中宏定义#define TEST\_A 1和#define TEST\_CLASS\_A clase T1是有效的，过了这一语句，宏定义就释放掉了；所以在TEST.h里，这个宏是有效的，然后出了这个头文件，又无效了。

**在TEST.h（或TEST.cpp）中1就是TEST\_A，clase T1就是TEST\_CLASS\_A clase**，这只在TEST中有效；

**7、#elif：**#else指令用于某个#if指令之后，当前面的#if指令的条件不为真时，就编译#else后面的代码。#endif指令将终止上面的条件块。#elif预处理指令综合了#else和#if指令的作用。

#elif 使您得以创建**复合条件指令**。如果前面的 #if 和前面的任何 #elif（可选）指令表达式的计算结果都不是 true，则将计算 #elif 表达式。

如果 #elif 表达式计算为 true，编译器将计算位于 #elif 和下一个条件指令之间的所有代码。例如：

**#define VC7**

**#if debug**

**Console.Writeline("Debugbuild");**

**#elif VC7**

**Console.Writeline("VisualStudio7");**

**#endif**



使用 #elif 更简单，因为每个 #if 都需要一个 #endif，而 #elif 即使在没有匹配的 #endif 时也可以使用。

**8、defined：**检查某个名称的常量是否存在

检查该名称的常量是否已定义。如果该名称的常量已定义，返回 TRUE；未定义则返回 FALSE。

9、**#pragma pack：**程序编译器对结构的存储的特殊处理确实提高CPU存储变量的速度，但是有时候也带来了一些麻烦，我们也屏蔽掉变量默认的对齐方式，**自己可以设定变量的对齐方式**。

用 途 ：设定变量以n字节对齐方式

作 用 ：指定结构体、联合以及类成员

重要规则 ：每个成员分别对齐

编译器中提供了#pragma pack(n)来设定变量以n字节对齐方式。n字节对齐就是说变量存放的起始地址的偏移量有两种情况：第一、如果n大于等于该变量所占用的字节数，那么偏移量必须满足默认的对齐方式；第二、如果n小于该变量的类型所占用的字节数，那么偏移量为n的倍数，不用满足默认的对齐方式。**结构的总大小也有个约束条件，分下面两种情况：如果n大于所有成员变量类型所占用的字节数，那么结构的总大小必须为占用空间最大的变量占用的空间数的倍数；否则必须为n的倍数。**

**面举例说明其用法。**

**#pragma pack(push) //保存对齐状态**

**#pragma pack(4)//设定为4字节对齐**

**struct test**

**{**

**char m1;**

**double m4;**

**int m3;**

**};**

**#pragma pack(pop)//恢复对齐状态**

以上结构体的大小为16，下面分析其存储情况，首先为m1分配空间，其偏移量为0，满足我们自己设定的对齐方式（4字节对齐），m1大小为1个字节。接着开始为m4分配空间，这时其偏移量为1，需要补足3个字节，这样使偏移量满足为n=4的倍数（**因为sizeof(double)大于4**）,m4占用8个字节。接着为m3分配空间，这时其偏移量为12，满足为4的倍数，m3占用4个字节。这时已经为所有成员变量分配了空间，**共分配了16个字节**，满足为n的倍数。**(常见数据类型所占字节：4字节长度的int，1字节长度的char和2字节长度的short型数据)**

如果把上面的#pragma pack(4)改为#pragma pack(8)，那么我们可以得到结构的大小为24。

***什么是对齐，以及为什么要对齐呢？***

*现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定变量的时候经常在特定的内存地址访问，这就需要****各类型数据按照一定的规则在空间上排列****，而不是顺序的一个接一个的排放，这就是对齐。*

*对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对数据存放进行对齐，会在****存取效率****上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出，而如果存放在奇地址开始的地方，就可能会需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该int数据。显然在读取效率上下降很多。这也是****空间和时间的博弈****。*

对齐的实现：通常，我们写程序的时候，不需要考虑对齐问题。编译器会替我们选择适合目标平台的对齐策略。当然，我们也可以通知给编译器传递预编译指令而改变对指定数据的对齐方法。 但是，正因为我们一般不需要关心这个问题，所以因为编辑器对数据存放做了对齐，而我们不了解的话，常常会对一些问题感到迷惑。最常见的就是struct数据结构的sizeof结果，出乎意料。为此，我们需要对对齐算法所了解。

作用：指定结构体、联合以及类成员的packing alignment;

语法：#pragma pack( [show] | [push | pop],[ identifier], n )

说明：

1，pack提供数据声明级别的控制，对定义不起作用；

2，调用pack时不指定参数，n将被设成默认值；

3，一旦改变数据类型的alignment（对齐方式），直接效果就是占用memory的减少，但是performance会下降。

语法具体分析：

1，show：可选参数；显示当前packing aligment的字节数，以warning message的形式被显示；

2，**push**：可选参数；将当前指定的packing alignment数值进行**压栈操作**，这里的栈是the internal compiler stack（内部编译器堆栈），同时设置当前的packing alignment为n；如果n没有指定，则将当前的packing alignment数值压栈；

3，pop：可选参数；从internal compiler stack中删除最顶端的record；如果没有指定n，则当前栈顶record即为新的packing alignment数值；如果指定了n，则n将成为新的packing aligment数值；如果指定了identifier，则internal compiler stack中的record都将被pop直到identifier被找到，然后pop出identitier，同时设置packing alignment数值为当前栈顶的record；如果指定的identifier并不存在于internal compiler stack，则pop操作被忽略；

4，identifier：可选参数；当同push一起使用时，赋予当前被压入栈中的record一个名称；当同pop一起使用时，从internal compiler stack中pop出所有的record直到identifier被pop出，如果identifier没有被找到，则忽略pop操作；

5，n：可选参数；指定packing的数值，以字节为单位；缺省数值是8，合法的数值分别是1、2、4、8、16。

**在相同的对齐方式下，结构体内部数据定义的顺序不同，结构体整体占据内存空间也不同，如下： 设结构体如下定义： struct A { int a; char b; short c; }; 结构体A中包含了4字节长度的int一个，1字节长度的char一个和2字节长度的short型数据一个。所以A用到的空间应该是7字节。但是因为编译器要对数据成员在空间上进行对齐。所以使用sizeof(strcut A)值为8。 现在把该结构体调整成员变量的顺序。 struct B { char b; int a; short c; }; 这时候同样是总共7个字节的变量，但是sizeof(struct B？)的值却是12。**

struct B { char b; int a; short c; }; 为什么是12个字节呢？

**假设B从地址空间0x0000开始排放。该例子中没有定义指定对齐值，在笔者环境下，该值默认为4。**

**第一个成员变量b的自身对齐值是1，比指定或者默认指定对齐值4小，所以其有效对齐值为1，所以其存放地址0x0000符合0x0000%1=0.**

**第二个成员变量a，其自身对齐值为4，所以有效对齐值也为4，所以只能存放在起始地址为0x0004到0x0007这四个连续的字节空间中，符合0x0004%4=0, 且紧靠第一个变量。**

**第三个变量c,自身对齐值为2，所以有效对齐值也是2，可以存放在0x0008到0x0009这两个字节空间中，符合0x0008%2=0。所以从0x0000到0x0009存放的都是B内容。**

**再看数据结构B的自身对齐值为其变量中最大对齐值(这里是a）和指定对齐值(这里是4)中较小的那个，所以就是4，所以结构体的有效对齐值也是4。根据结构体圆整的要求，0x0009到0x0000=10字节，（10+2）%4=0。所以0x0000A到0x000B也为结构体B所占用。故B从0x0000到0x000B共有12个字节,sizeof(struct B)=12;**

**下面我们使用预编译指令#pragma pack (value)来告诉编译器，使用我们指定的对齐值来取代缺省的。**

**#pragma pack (2) /\*指定按2字节对齐，等价于#pragma pack(push,2)\*/**

**struct C { char b; int a; short c; };**

**#pragma pack () /\*取消指定对齐，恢复缺省对齐,等价于#pragma pack(pop)\*/**

**sizeof(struct C)值是8。为什么是8呢？解释如下：**

第一个变量b的自身对齐值为1，指定对齐值为2，所以，其有效对齐值为1，假设C从0x0000开始，那么b存放在0x0000，符合0x0000%1=0;

第二个变量，自身对齐值为4，指定对齐值为2，所以有效对齐值为2，所以顺序存放在0x0002、0x0003、0x0004、0x0005四个连续字节中，符合0x0002%2=0。

第三个变量c的自身对齐值为2，所以有效对齐值为2，顺序存放在0x0006、0x0007中，符合0x0006%2=0。所以从0x0000到0x00007共八字节存放的是C的变量。

又C的自身对齐值为4，所以C的有效对齐值为2。又8%2=0,C只占用0x0000到0x0007的八个字节。所以sizeof(struct C)=8.

**修改对齐值为1：**

**#pragma pack (1) /\*指定按1字节对齐\*/**

**struct D { char b; int a; short c; };**

**#pragma pack () /\*取消指定对齐，恢复缺省对齐\*/**

 **sizeof(struct D)值为7。**

**！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！对于char型数据，其自身对齐值为1，对于short型为2，对于int,float,long类型，其自身对齐值为4，double,long long类型，其自身对齐值为8，单位字节。**

**10、extern "c"用法解析**

extern "C"的主要作用就是为了能够正确**实现C++代码调用其他C语言代码。**加上extern "C"后，会指示编译器这部分代码按C语言的进行编译，而不是C++的。由于C++支持**函数重载**，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

例如，假设某个函数的原型为：

void foo( int x, int y );

该函数被C编译器编译后在符号库中的名字为**\_foo**，而C++编译器则会产生像\_foo\_int\_int之类的名字（不同的编译器可能生成的名字不同，但是都采用了相同的机制，生成的新名字称为“mangled name”）。

**\_foo\_int\_int**这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。 例如，在C++中，函数void foo( int x, int y )与void foo( int x, float y )编译生成的符号是不相同的，后者为**\_foo\_int\_float**。

原文链接：<http://www.jianshu.com/p/5d2eeeb93590>

著作权归作者所有，转载请联系作者获得授权，并标注“简书作者”。

***函数重载****解释如下：重载函数是函数的一种特殊情况，为方便使用，C++允许在同一范围中声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同，也就是说用同一个运算符完成不同的运算功能。这就是重载函数。重载函数常用来实现功能类似而所处理的数据类型不同的问题。*

**11、#pragma once**

这是一个比较常用的**C/C++杂注**，只要**在头文件的最开始**加入这条杂注，就能够保证头文件**只被编译一次**。

#pragma once是编译器相关的，就是说即使这个编译系统上有效，但在其他编译系统也不一定可以，不过现在基本上已经是每个编译器都有这个杂注了。

#ifndef，#define，#endif是C/C++语言中的宏定义，通过宏定义**避免文件多次编译**。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

我还看到一种用法是把两者放在一起的：

**#pragma once**

**#ifndef \_\_SOMEFILE\_H\_\_**

**#define \_\_SOMEFILE\_H\_\_**

**... ... // 一些声明语句**

**#endif**

看起来似乎是想兼有两者的优点。不过只要使用了#ifndef就会有宏名冲突的危险，所以混用两种方法似乎不能带来更多的好处，倒是会让一些不熟悉的人感到困惑。