GNU C 的一大特色就是__attribute__ 机制。__attribute__ 可以设置函数属性（Function Attribute ）、变量属性（Variable Attribute ）和类型属性（Type Attribute ）。

__attribute__ 书写特征是：__attribute__ 前后都有两个下划线，并切后面会紧跟一对原括弧，括弧里面是相应的__attribute__ 参数。

__attribute__ 语法格式为：__attribute__ ((attribute-list))

其位置约束为：放于声明的尾部“ ；”之前。

关键字__attribute__ 也可以对结构体（struct ）或共用体（union ）进行属性设置。大致有六个参数值可以被设定，即：aligned, packed, transparent_union, unused, deprecated 和 may_alias 。

在使用__attribute__ 参数时，你也可以在参数的前后都加上“__”（两个下划线），例如，使用__aligned__而不是 aligned ，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。

aligned (alignment)

该属性设定一个指定大小的对齐格式（以字节 为单位），例如：

```
struct S {
short b[3];
} __attribute__ ((aligned (8)));
typedef int int32_t __attribute__ ((aligned (8)));
```

该声明将强制编译器确保（尽它所能）变量类 型为 struct S 或者 int32_t 的变量在分配空间时采用 8 字节对齐方式。

如上所述，你可以手动指定对齐的格式，同 样，你也可以使用默认的对齐方式。如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
struct S {
short b[3];
```

} __attribute__ ((aligned));

这里，如果 sizeof （ short ）的大小为 2 （ byte ），那么，S 的大小就为 6 。取一个 2 的次方值，使得该值大于等于 6 ，则该值为 8 ，所以编译器将设置 S 类型的对齐方式为 8 字节。

aligned 属性使被设置的对象占用更多的空间，相反的，使用 packed 可以减小对象占用的空间。

需要注意的是，attribute 属性的效力与你的连接器也有关，如果你的连接器最大只支持 16 字节对齐，那么你此时定义 32 字节对齐也是无济于事的。

packed

使用该属性对 struct 或者 union 类型进行定义，设定其类型的每一个变量的内存约束。当用在 enum 类型 定义时，暗示了应该使用最小完整的类型（it indicates that the smallest integral type should be used）。

下面的例子中，packed_struct 类型的变量数组中的值将会紧紧的靠在一起，但内部的成员变量 s 不会被“pack”，如果希望内部的成员变量也被 packed 的话，unpacked-struct 也需要使用 packed 进行相应的约束。

```
struct unpacked_struct
{
    char c;
    int i;
};

struct packed_struct
{
   char c;
   int  i;
   struct unpacked_struct s;
```

}__attribute__ ((__packed__));

下面的例子中使用__attribute__ 属性定义了一些结构体及其变量，并给出了输出结果和对结果的分析。

程序代 码为：

```
1 struct p
2
3 {
4
5 int a;
6
7 char b;
8
9 short c;
10
11 }__attribute__((aligned(4))) pp;
12
13 struct m
14
15 {
16
17 char a;
18
19 int b;
20
21 short c;
22
23 }__attribute__((aligned(4))) mm;
24
25 struct o
26
27 {
28
29 int a;
30
31 char b;
32
```

```
33 short c;
34
35 }oo;
36
37 struct x
38
39 {
40
41 int a;
42
43 char b;
44
45 struct p px;
46
47 short c;
48
49 }__attribute__((aligned(8))) xx;
50
51 int main()
52
53 {
54
55 printf("sizeof(int)=%d,sizeof(short)=
%d.sizeof(char)=
%d\n",sizeof(int),sizeof(short),sizeof(char));
56
57 printf("pp=%d,mm=%d \n", sizeof(pp),sizeof(mm));
58
59 printf("oo=%d,xx=%d \n", sizeof(oo),sizeof(xx));
60
61 return 0;
62
63 }
```

输出结 果：

sizeof(int)=4,sizeof(short)=2.sizeof(char)=1

pp=8,mm=12

oo=8,xx=24

分析：

sizeof(pp):

sizeof(a)+sizeof(b)+sizeof(c)=4+1+1=6<8 所以 sizeof(pp)=8

sizeof(mm):

sizeof(a)+sizeof(b)+sizeof(c)=1+4+2=7

但是 a 后面需要用 3 个字节填充，但是 b 是 4 个字节，所以 a 占用 4 字节，b 占用 4 个字节，而 c 又要占用 4 个字节。所以 sizeof(mm)=12

sizeof(oo):

sizeof(a)+sizeof(b)+sizeof(c)=4+1+2=7

因为默 认是以 4 字节对齐，所以 sizeof(oo)=8

sizeof(xx):

sizeof(a)+ sizeof(b)=4+1=5

sizeof(pp)=8; 即 xx 是采用 8 字节对齐的，所以要在 a ，b 后面添 3 个空余字节，然后才能存储 px ，

4+1+（3）+8+1=17

因为 xx 采用的对齐是 8 字节对齐，所以 xx 的大小必定是 8 的整数倍，即 xx 的大小是一个比 17 大又是 8 的倍数的一个最小值，由此得到

17<24 ，所以 sizeof(xx)=24

函数属性（Function Attribute）

函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。__attribute__机制也很容易同非 GNU 应用程序做到兼容之功效。

GNU CC 需要使用 –Wall 编译器来击活该功能，这是控制警告信息的一个很好的方式。下面介绍几个常见的属性参数。

__attribute__ format

该__attribute__属性可以给被声明的函数加上类似 printf 或者 scanf 的特征，它可以使编译器检查函数声明和函数实际调用参数之间的格式化字符串是

否匹配。该功能十分有用，尤其是处理一些很难发现的 bug。
format 的语法格式为：
format (archetype, string-index, first-to-check)
format 属性告诉编译器，按照 printf, scanf,
strftime 或 strfmon 的参数表格式规则对该函数的参数进行检查。
“archetype”指定是哪种风格；“string-index”指定传入函数的第几个参数是格式化字符串；“first-to-check”指定从函数的第几个参数开始按上述规则进行检查。
具体使用格式如下：
__attribute__((format(printf,m,n)))
__attribute__((format(scanf,m,n)))
其中参数 m 与 n 的含义为：
m：第几个参数为格式化字符串（format string）；
n：参数集合中的第一个，即参数“...”里的第一个参数在函数参数总数排在第几，注意，有时函数参数里还有“隐身”的呢，后面会提到；
在使用上，__attribute__((format(printf,m,n)))是常用的，而另一种却很少见到。下面举例说明，其中 myprint 为自己定义的一个带有可变参数的函数，其功能类似于 printf：

```
//m=1；n=2
extern void myprint(const char *format,...) __attribute__((format(printf,1,2)));
//m=2；n=3
extern void myprint(int l，const char *format,...)
__attribute__((format(printf,2,3)));
```

需要特别注意的是，如果 myprint 是一个函数的成员函数，那么 m 和 n 的值可有点“悬乎”了，例如：

```
//m=3；n=4
extern void myprint(int l，const char *format,...)
__attribute__((format(printf,3,4)));
```

其原因是，类成员函数的第一个参数实际上一个“隐身”的“this”指针。（有点 C++基础的都知道点 this 指针，不知道你在这里还知道吗？）

这里给出测试用例：attribute.c，代码如下：

```
1：
2：extern void myprint(const char *format,...)
__attribute__((format(printf,1,2)));
3：
4：void test()
5：{
6：   myprint("i=%d\n",6);
7：   myprint("i=%s\n",6);
8：   myprint("i=%s\n","abc");
9：   myprint("%s,%d,%d\n",1,2);
10：}
```

运行$gcc –Wall –c attribute.c attribute 后，输出结果为：

attribute.c: In function `test':
attribute.c:7: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: too few arguments for format

如果在 attribute.c 中的函数声明去掉__attribute__((format(printf,1,2)))，再重新编译，既运行$gcc –Wall –c attribute.c attribute 后，则并不会输出任何警告信息。
注意，默认情况下，编译器是能识别类似 printf 的“标准”库函数。
__attribute__ noreturn
该属性通知编译器函数从不返回值，当遇到类似函数需要返回值而却不可能运行到返回值处就已经退出来的情况，该属性可以避免出现错误信息。C 库函数中的 abort（）和 exit（）的声明格式就采用了这种格式，如下所示：

extern void exit(int)　　__attribute__((noreturn));extern void abort(void) __attribute__((noreturn)); 为了方便理解，大家可以参考如下的例子：

```
 1 //name: noreturn.c      ;测试 __attribute__((noreturn))
 2 extern void myexit();
 3
 4 int test(int n)
 5 {
 6             if ( n > 0 )
 7            {
 8                     myexit();
 9                   /* 程序不可能到达这里*/
10            }
11             else
12                     return 0;
13 }
```

编译显示的输出信息为：

$gcc –Wall –c noreturn.c
noreturn.c: In function `test':
noreturn.c:12: warning: control reaches end of non-void function

警告信息也很好理解，因为你定义了一个有返回值的函数 test 却有可能没有返回值，程序当然不知道怎么办了！
加上__attribute__((noreturn))则可以很好的处理类似这种问题。把 extern void myexit();修改为：
extern void myexit() __attribute__((noreturn));之后，编译不会再出现警告信息。
__attribute__ const

该 属性只能用于带有数值类型参数的函数上。当重复调用带有数值参数的函数时，由于返回值是相同的，所以此时编译器可以进行优化处理，除第一次需要运算外， 其它只需要返回第一次的结果就可以了，进而可以提高效率。该属性主要适用于没有静态状态（static state）和副作用的一些函数，并且返回值仅仅依赖输入的参数。
为了说明问题，下面举个非常“糟糕”的例子，该例子将重复调用一个带有相同参数值的函数，具体如下：

```
extern int square(int n) __attribute__ ((const));...  for (i = 0; i < 100; i++ ) { total += square (5) + i; }
```

通过添加__attribute__((const))声明，编译器只调用了函数一次，以后只是直接得到了相同的一个返回值。
事实上，const 参数不能用在带有指针类型参数的函数中，因为该属性不但影响函数的参数值，同样也影响到了参数指向的数据，它可能会对代码本身产生严重甚至是不可恢复的严重后果。
并且，带有该属性的函数不能有任何副作用或者是静态的状态，所以，类似 getchar（）或 time（）的函数是不适合使用该属性的。
-finstrument-functions
该参数可以使程序在编译时，在函数的入口和出口处生成 instrumentation 调用。恰好在函数入口之后并恰好在函数出口之前，将使用当前函数的地址和调用地址来调用下面的
profiling
函数。（在一些平台上，__builtin_return_address 不能在超过当前函数范围之外正常工作，所以调用地址信息可能对 profiling 函数是无效的。）

```
void __cyg_profile_func_enter(void *this_fn, void *call_site);
void __cyg_profile_func_exit(void *this_fn, void *call_site);
```

其中，第一个参数 this_fn 是当前函数的起始地址，可在符号表中找到；第二个参数 call_site 是指调用处地址。
instrumentation
也可用于在其它函数中展开的内联函数。从概念上来说，profiling 调用将

指出在哪里进入和退出内联函数。这就意味着这种函数必须具有可寻址形式。如 果函数包含内联，而所有使用到该函数的程序都要把该内联展开，这会额外地增加代码长度。如果要在 C 代码中使用 extern inline 声明，必须提供这种函数的可寻址形式。

可 对函数指定 no_instrument_function 属性，在这种情况下不会进行 Instrumentation 操作。例如，可以在以下情况下使用 no_instrument_function 属性：上面列出的 profiling 函 数、高优先级的中断例程以及任何不能保证 profiling 正常调用的函数。

no_instrument_function

如果使用了-finstrument-functions

，将在绝大多数用户编译的函数的入口和出口点调用 profiling 函数。使用该属性，将不进行 instrument 操作。

constructor/destructor

若 函数被设定为 constructor 属性，则该函数会在 main（）函数执行之前被自动的执行。类似的，若函数被设定为 destructor 属性，则该 函数会在 main（）函数执行之后或者 exit（）被调用后被自动的执行。拥有此类属性的函数经常隐式的用在程序的初始化数据方面。

这两个属性还没有在面向对象 C 中实现。

同时使用多个属性

可以在同一个函数声明里使用多个__attribute__，并且实际应用中这种情况是十分常见的。使用方式上，你可以选择两个单独的__attribute__，或者把它们写在一起，可以参考下面的例子：

/* 把类似 printf 的消息传递给 stderr 并退出 */extern void die(const char *format, ...) __attribute__((noreturn)) __attribute__((format(printf, 1, 2))); 或者写成 extern void die(const char *format, ...) __attribute__((noreturn, format(printf, 1, 2))); 如果带有该属性的自定义函数追加到库的头文件里，那么所以调用该函数的程序都要做相应的检查。

和非 GNU 编译器的兼容性

庆幸的是，__attribute__设计的非常巧妙，很容易作到和其它编译器保持兼容，也就是说，如果工作在其它的非 GNU 编译器上，可以很容易的忽略该属性。即使__attribute__使用了多个参数，也可以很容易的使用一对圆括弧进行处理，例如：

/* 如果使用的是非 GNU C, 那么就忽略__attribute__ */#ifndef __GNUC__# define __attribute__(x) /*NOTHING*/#endif
需要说明的是，__attribute__适用于函数的声明而不是函数的定义。所以，当需要使用该属性的函数时，必须在同一个文件里进行声明，例如：

/* 函数声明 */void die(const char *format, ...)
__attribute__((noreturn))
__attribute__((format(printf,1,2))); void die(const char *format, ...){
/* 函数定义 */}

Specifying Attributes of Variables

`aligned` (*alignment*)This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On a 68040, this could be used in conjunction with an `asm` expression to access the`move16` instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

for more: http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Variable-Attributes.html#Variable-Attributes

下面来看一个不一样的 HelloWorld 程序:

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

static __attribute__((constructor))
void before()
{

  printf("Hello");
}

static __attribute__((destructor))
void after()
{
  printf(" World!\n");
}

int main(int args,char ** argv)
{

  return EXIT_SUCCESS;
}
```

1
9
我们知道这是一个 HelloWorld 程序，所以输出的结果就是"Hello World!"，很简单，不需要对这点过多关心.

下面我们来关心关心别的:

```
__attribute__((constru
1ctor))
2
3__attribute__((destruc
 tor))
```

解释一下：__attribute__((constructor)) 在 main() 之前执行,__attribute__((destructor)) 在 main()执行结束之后执行.

上面的例子中我没有在 main 函数中添加任何的输出，所以看不到具体的信息.这点可以自己尝试 ~

如果要在 main()之前或者是执行完成之后，需要执行很多的前处理动作或者是后处理动作，我们应该怎么处理?

也许，你需要下面这些东西:

```
__attribute__((constructor(PRI
ORITY)))
__attribute__((destructor(PRIO
RITY)))
```

PRIORITY: 优先级.

好吧，下面就来试试：

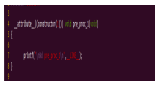
执行的输出如下:

从输出的信息看，前处理都是按照优先级先后执行的，而后处理则是相反的，好吧，我们使用 GDB 调试验证一下:

从调试的信息也是验证了上面的结果.

另外一个问题，优先级有没有范围的？

其实刚开始我写的程序中的优先级是 1,我们将上面的程序改一下，然后编译看一下会有什么样的结果:



0-100(包括 100),是内部保留的，所以在编码的时候需要注意.

关于__attribute__的用法，可以有另外一种写法，先声明函数，然后再定义.

glibc 多采用第一种写法.

关于 linux 内核中的"__attribute__ ((packed))"

引用:

__attrubte__ ((packed)) 的作用就是告诉编译器取消结构在编译过程中的优化对齐,按照实际占用字节数进行对齐。

#define __u8   unsigned char

```
#define __u16   unsigned short
/* __attribute__ ((packed)) 的位置约束是放于声明的尾部“；”之前 */
struct str_struct{
        __u8    a;
        __u8    b;
        __u8    c;
        __u16   d;
} __attribute__ ((packed));
/*  当用到 typedef 时，要特别注意__attribute__ ((packed))放置的位置，相当于：
  *  typedef struct str_stuct str;
  *  而 struct str_struct 就是上面的那个结构。
  */
typedef struct {
        __u8    a;
        __u8    b;
        __u8    c;
        __u16   d;
} __attribute__ ((packed)) str;
/* 在下面这个 typedef 结构中，__attribute__ ((packed))放在结构名 str_temp 之后，其作用是被忽略的，注意与结构 str 的区别。 */
typedef struct {
        __u8    a;
        __u8    b;
        __u8    c;
        __u16   d;
}str_temp __attribute__ ((packed));
typedef struct {
        __u8    a;
        __u8    b;
        __u8    c;
        __u16   d;
}str_nopacked;
int main(void)
```

```
{
	printf("sizeof str = %d\n", sizeof(str));
	printf("sizeof str_struct = %d\n", sizeof(struct str_struct));
	printf("sizeof str_temp = %d\n", sizeof(str_temp));
	printf("sizeof str_nopacked = %d\n", sizeof(str_nopacked));
	return 0;
}
```

编译运行:

引用:

```
[root@localhost root]# ./packedtest
sizeof str = 5
sizeof str_struct = 5
sizeof str_temp = 6
sizeof str_nopacked = 6
```

GNU C 的一大特色就是__attribute__机制。__attribute__可以设置函数属性（Function Attribute）、变量属性（Variable Attribute）和类型属性（Type Attribute）。

__attribute__书写特征是：__attribute__前后都有两个下划线，并且后面会紧跟一对括弧，括弧里面是相应的__attribute__参数。

__attribute__语法格式为：

__attribute__ ((attribute-list))

其位置约束：放于声明的尾部“；”之前。

函数属性（Function Attribute）：函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。

__attribute__机制也很容易同非 GNU 应用程序做到兼容之功效。

GNU CC 需要使用 –Wall 编译器来击活该功能，这是控制警告信息的一个很好的方式。

packed 属性：使用该属性可以使得变量或者结构体成员使用最小的对齐方式，即对变量是一字节对齐，对域（field）是位对齐。

网络通信通常分为基于数据结构的和基于流的。HTTP 协议就是后者的一个例子。

有时为了提高程序的处理速度和数据处理的方便，会使用基于数据结构的通信（不需要对流进行解析）。但是，当需要在多平台间进行通信时，基于数据结构的通信，往往要十分注意以下几个方面：

[1] 字节序

[2] 变量长度

[3] 内存对齐

在常见的系统架构中（Linux X86，Windows），非单字节长度的变量类型，都是低字节在前，而在某些特定系统中，如 Soalris Sparc 平台，高字节在前。如果在发送数据前不进行处理，那么由 Linux X86 发向 Soalris Sparc 平台的数据值，势必会有极大的偏差，进而程序运行过程中无法出现预计的正常结果，更严重时，会导致段错误。

对于此种情况，我们往往使用同一的字节序。在系统中，有 ntohXXX(), htonXXX()等函数，负责将数据在网络字节序和本地字节序之间转换。虽然每种系统的本地字节序不同，但是对于所有系统来说，网络字节序是固定的 -----高字节在前。所以，可以以网络字节序为通信的标准，发送前，数据都转换为网络字节序。

转换的过程，也建议使用 ntohXXX(), htonXXX()等标准函数，这样代码可以轻松地在各平台间进行移植（像通信这种很少依赖系统 API 的代码，做成通用版本是不错的选择）。

变量的长度，在不同的系统之间会有差别，如同是 Linux2.6.18 的平台，在 64 位系统中，指针的长度为 8 个字节，而在 32 位系统中，指针又是 4 个字 节的长度---此处只是举个例子，很少有人会将指针作为数据发送出去。下面是我整理的在 64 位 Linux 系统和 32 位 Linux 系统中，几种常见 C 语言变 量的长度：

	short	int	long	long long	ptr	time_t
32 位	2	4	4	8	4	4
64 位	2	4	8	8	8	8

在定义通信用的结构体时，应该考虑使用定常的数据类型，如 uint32_t，4 字节的固定长度，并且这属于标准 C 库(C99)，在各系统中都可使用。

内存对齐的问题，也与系统是 64 位还是 32 位有关。如果你手头有 32 位和 64 位系统，不妨写个简单的程序测试一下，你就会看到同一个结构体，即便使用了定 常的数据类型，在不同系统中的大小是不同的。对齐往往是以 4 字节或 8 字节为准的，只要你写的测试程序，变量所占空间没有对齐到 4 或 8 的倍数即可，举个简单 的测试用的结构体的例子吧：

```
struct student
{
   char name[7];
   uint32_t id;
   char subject[5];
};
```

在每个系统上看下这个结构体的长度吧。

内存对齐，往往是由编译器来做的，如果你使用的是 gcc，可以在定义变量时，添加__attribute__，来决定是否使用内存对齐，或是内存对齐到几个字节，以上面的结构体为例：

1)到 4 字节，同样可指定对齐到 8 字节。

```
struct student
{
   char name[7];
   uint32_t id;
   char subject[5];
} __attribute__ ((aligned(4)));
```

2)不对齐，结构体的长度，就是各个变量长度的和

```
struct student
{
   char name[7];
   uint32_t id;
   char subject[5];
} __attribute__ ((packed));
```

One of the best (but little known) features of GNU C is

the **__attribute__** mechanism, which allows a developer to attach characteristics to function declarations to allow the compiler to perform more error checking. It was designed in a way to be compatible with non-GNU implementations, and we've been using this for *years* in highly portable code with very good results.

Table of Contents

Note that __**attribute**__ spelled with two underscores before and two after, and there are always *two* sets of parentheses surrounding the contents. There is a good reason for this - see below. Gnu CC needs to use the **-Wall** compiler directive to enable this (yes, there is a finer degree of warnings control available, but we are very big fans of max warnings anyway).

__attribute__ format

This **__attribute__** allows assigning **printf**-like or **scanf**-like characteristics to the declared function, and this enables the compiler to check the format string against the parameters provided throughout the code. This is *exceptionally* helpful in tracking down hard-to-find bugs.

There are two flavors:

- **__attribute__((format(printf,*m*,*n*)))**
- **__attribute__((format(scanf,*m*,*n*)))**

but in practice we use the first one much more often.

The (*m*) is the number of the "format string" parameter, and (*n*) is the number of the first variadic parameter. To see some examples:

```
/* like printf() but to standard error only */
extern void eprintf(const char *format, ...)
```

```
    __attribute__((format(printf, 1, 2)));  /*
1=format 2=params */

/* printf only if debugging is at the desired level
*/
extern void dprintf(int dlevel, const char
*format, ...)
    __attribute__((format(printf, 2, 3)));  /*
2=format 3=params */
```

With the functions so declared, the compiler will examine the argument lists

```
$ cat test.c
1  extern void eprintf(const char *format, ...)
2               __attribute__((format(printf, 1,
2)));
3
4  void foo()
5  {
6       eprintf("s=%s\n", 5);              /* error on
this line */
7
8       eprintf("n=%d,%d,%d\n", 1, 2);     /* error on
this line */
9  }

$ cc -Wall -c test.c
test.c: In function `foo':
test.c:6: warning: format argument is not a pointer
(arg 2)
test.c:8: warning: too few arguments for format
```

Note that the "standard" library functions - `printf` and the like - are already understood by the compiler by default.

__attribute__ noreturn

This attribute tells the compiler that the function won't ever return, and this can be used to suppress errors about code paths not being reached. The C library functions `abort()` and `exit()` are both declared with this attribute:

```
extern void exit(int)   __attribute__((noreturn));
extern void abort(void) __attribute__((noreturn));
```

Once tagged this way, the compiler can keep track of paths through the code and suppress errors that won't ever happen due to the flow of control never returning after the function call.

In this example, two nearly-identical C source files refer to an "`exitnow()`" function that never returns, but without the **__attribute__**tag, the compiler issues a warning. The compiler is correct here, because it has no way of knowing that control doesn't return.

```
$ cat test1.c
extern void exitnow();

int foo(int n)
{
    if ( n > 0 )
    {
        exitnow();
        /* control never reaches this point */
    }
    else
        return 0;
}

$ cc -c -Wall test1.c
test1.c: In function `foo':
test1.c:9: warning: this function may return with or without a value
```

But when we add **__attribute__**, the compiler suppresses the spurious warning:

```
$ cat test2.c
extern void exitnow() __attribute__((noreturn));

int foo(int n)
{
    if ( n > 0 )
        exitnow();
    else
        return 0;
}
```

```
$ cc -c -Wall test2.c
no warnings!
```

__attribute__ const

This attribute marks the function as considering *only* its numeric parameters. This is mainly intended for the compiler to optimize away repeated calls to a function that the compiler knows will return the same value repeatedly. It applies mostly to math functions that have no static state or side effects, and whose return is solely determined by the inputs.

In this highly-contrived example, the compiler normally *must* call the **square()** function in every loop even though we know that it's going to return the same value each time:

```
extern int square(int n) __attribute__((const));

...
    for (i = 0; i < 100; i++ )
    {
        total += square(5) + i;
    }
```

By adding **__attribute__((const))**, the compiler can choose to call the function just once and cache the return value.

In virtually every case, **const** can't be used on functions that take pointers, because the function is not considering just the function parameters but *also the data the parameters point to*, and it will almost certainly break the code very badly in ways that will be nearly impossible to track down.

Furthermore, the functions so tagged cannot have any side effects or static state, so things like **getchar()** or **time()** would behave very poorly under these circumstances.

Putting them together

Multiple **__attributes__** can be strung together on a single declaration, and this is not uncommon in practice. You can either use two separate **__attribute__**s, or use one with a comma-separated list:

```
/* send printf-like message to stderr and exit */
extern void die(const char *format, ...)
    __attribute__((noreturn))
     __attribute__((format(printf, 1, 2)));

/*or*/

extern void die(const char *format, ...)
    __attribute__((noreturn, format(printf, 1, 2)));
```

If this is tucked away safely in a library header file, *all* programs that call this function receive this checking.

Compatibility with non-GNU compilers

Fortunately, the **__attribute__** mechanism was cleverly designed in a way to make it easy to quietly eliminate them if used on platforms other than GNU C. Superficially, **__attribute__** appears to have multiple parameters (which would typically rule out using a macro), but the *two* sets of parentheses effectively make it a single parameter, and in practice this works very nicely.

```
/* If we're not using GNU C, elide __attribute__ */
#ifndef __GNUC__
#  define  __attribute__(x)  /*NOTHING*/
#endif
```

Note that **__attribute__** applies to function *declarations*, not *definitions*, and we're not sure why this is. So when defining a function that merits this treatment, an extra declaration must be used (in the same file):

```
/* function declaration */
void die(const char *format, ...)
__attribute__((noreturn))
                __attribute__((format(printf,1,2)));

void die(const char *format, ...)
{
    /* function definition */
}
```

Other References

We'll note that there are many more attributes available, including those for **variables** and **types**, and they are not covered here: we have chosen to just touch on the high points. Those wishing more information can find it in the GNU online documentation at http://gcc.gnu.org:

GCC 4.0
- GCC 4.0 Function Attributes
- GCC 4.0 Variable Attributes
- GCC 4.0 Type Attributes

GCC 3.2
- GCC 3.2 Function Attributes
- GCC 3.2 Variable Attributes
- GCC 3.2 Type Attributes

GCC 3.1
- GCC 3.1 Function Attributes
- GCC 3.1 Variable Attributes
- GCC 3.1 Type Attributes

GCC 3.0.4
- Function Attributes
- Variable Attributes
- Type Attributes

GCC 2.95.3
- Function Attributes
- Variable Attributes
- Type Attributes

参考：

http://blog.sina.com.cn/s/blog_644c3be70100i8ii.html

http://hi.baidu.com/srbadxecnihqtue/item/039535e051a0d30f8d3ea8b1

http://www.cnblogs.com/respawn/archive/2012/07/09/2582078.html

http://qq164587043.blog.51cto.com/261469/187562

http://my.oschina.net/u/174242/blog/72760

http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Function-Attributes.html

http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Type-Attributes.html#Type-

Attributes

http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Variable-Attributes.html#Variable-Attributes

http://www.unixwiz.net/techtips/gnu-c-attributes.html