gcc attribute 机制

GNU C 的一大特色（却不被初学者所知）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。

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：

```
//name: noreturn.c ；测试__attribute__((noreturn))
extern void myexit();

int test(int n)
{
 if ( n > 0 )
 {
 myexit();
 /* 程序不可能到达这里*/
```

```
 }
 else
 return 0;
}
```

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 */
}
```

更多的属性含义参考：
http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Function-Attributes.html

变量属性（Variable Attributes）
关键字__attribute__也可以对变量（variable）或结构体成员（structure field）进行属性设置。这里给出几个常用的参数的解释，更多的参数可参考本文给出的连接。
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)
该属性规定变量或结构体成员的最小的对齐格式，以字节为单位。例如：

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

编译器将以 16 字节（注意是字节 byte 不是位 bit）对齐的方式分配一个变量。也可以对结构体成员变量设置该属性，例如，创建一个双字对齐的 int 对，可以这么写：

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

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:

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

选择针对目标机器最大的对齐方式，可以提高拷贝操作的效率。
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。

下面的例子中，x 成员变量使用了该属性，则其值将紧放置在 a 的后面：

```
struct test
{
char a;
int x[2] __attribute__ ((packed));
};
```

其它可选的属性值还可以是：cleanup，common，nocommon，deprecated，mode，section，shared，tls_model，transparent_union，unused，vector_size，weak，dllimport，dlexport 等，

详细信息可参考：

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

类型属性（Type Attribute）

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 f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __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 more-aligned-int 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 f[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，my-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，my-unpacked-struct 也需要使用 packed 进行相应的约束。

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

struct my_packed_struct
{
 char c;
 int i;
 struct my_unpacked_struct s;
}__attribute__ ((__packed__));
```

其它属性的含义见：

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

变量属性与类型属性举例

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：

```
struct p
{
int a;
char b;
char c;
}__attribute__((aligned(4))) pp;

struct q
{
int a;
char b;
struct n qn;
char c;
}__attribute__((aligned(8))) qq;
```

```
int main()
{

printf("sizeof(int)=%d,sizeof(short)=%d.sizeof(char)=%d\n",sizeof(int),sizeof(short),
sizeof(char));
printf("pp=%d,qq=%d \n", sizeof(pp),sizeof(qq));

return 0;
}
```

输出结果：

```
sizeof(int)=4,sizeof(short)=2.sizeof(char)=1
pp=8,qq=24
```

分析：

sizeof(pp):

sizeof(a)+ sizeof(b)+ sizeof(c)=4+1+1=6<23=8= sizeof(pp)

sizeof(qq):

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

sizeof(qn)=8;即 qn 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 qn，

4+1+（3）+8+1=17

因为 qq 采用的对齐是 8 字节对齐，所以 qq 的大小必定是 8 的整数倍，即 qq 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+8=24= sizeof(qq)