Steve Friedl
UNIXWIZ.NET

Unixwiz.net - Software Consulting Central

# Using GNU C __attribute__

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

More Tech Tips

Home ▌ Stephen J. Friedl ▌ Software Consultant ▌ Orange County, CA USA ▌
steve@unixwiz.net ▌