filename: c__attribute__directives_01-20251210.txt
https://nshipster.com/__attribute__/

__attribute__
January 14^th, 2013

A recurring theme of this publication has been the importance of a healthy relationship with the compiler. Like any craft, one's effectiveness as a practitioner is contingent on how they treat their tools. Take good care of them, and they'll take good care of you.

__attribute__ is a compiler directive that specifies characteristics on declarations, which allows for more error checking and advanced optimizations.

The syntax for this keyword is __attribute__ followed by two sets of parentheses (the double parentheses makes it easy to "macro out", especially with multiple attributes). Inside the parentheses is a comma-delimited list of attributes. __attribute__ directives are placed after function, variable, and type declarations.

```
// Return the square of a number
int square(int n) __attribute__((const));

// Declare the availability of a particular API
void f(void) __attribute__((availability(macosx,introduced=10.4,deprecated=10.6)));

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

If this is starting to remind you of ISO C's #pragma, you're not alone.

In fact, when __attribute__ was first introduced to GCC, it was faced with some resistance by some who suggested that #pragma be used exclusively for the same purposes.

There were, however, two very good reasons why __attribute__ was added:

1. It was impossible to generate #pragma commands from a macro (before the C99 _Pragma operator).
2. There is no telling what the same #pragma might mean in another compiler.

Quoth the GCC Documentation for Function Attributes:
> These two reasons applied to almost any application that might have been proposed for #pragma. It was basically a mistake to use #pragma for anything.

Indeed, if you look at modern Objective-C-in the headers of Apple frameworks and well-engineered open-source projects-__attribute__ is used for myriad purposes. (By contrast, #pragma's main claim to fame these days is decoration: #pragma mark)

So without further ado, let's take a look at the most important attributes:

GCC

format
: The format attribute specifies that a function takes printf, scanf, strftime or strfmon style arguments which should be type-checked against a format string.

```
extern int my_printf (void *my_object, const char *my_format, ...) __attribute__((format(printf, 2, 3)));
```

Objective-C programmers can also use the __NSString__ format to enforce the same rules as format strings in NSString +stringWith
Format: and NSLog().

nonnull
: The nonnull attribute specifies that some function parameters should be non-null pointers.

```
extern void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull (1, 2)));
```

Using nonnull encodes expectations about values into an explicit contract, which can help catch any NULL pointer bugs lurking in any calling code. Remember: compile-time errors â\211« run-time errors.

noreturn
: A few standard library functions, such as abort and exit, cannot return. GCC knows this automatically. The noreturn attribute specifies this for any other function that never returns.

For example, AFNetworking uses the noreturn attribute for its network request thread entry point method. This method is used when spawning the dedicated network NSThread to ensure that the detached thread continues execution for the lifetime of the application.

pure / const
: The pure attribute specifies that a function has no effects except the return value, such that their return value depends only on the parameters and/or global variables. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be.

  The const attribute specifies that a function does not examine any values except their arguments, and have no effects except the return value. Note that a function that has pointer arguments and examines the data pointed to must not be declared const. Likewise, a function that calls a non-const function usually must not be const. It does not make sense for a const function to return void.

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

pure and const are both attributes that invoke a functional programming paradigm in order to allow for significant performance optimizations. const can be thought as a stricter form of pure since it doesn't depend on global values or pointers.

For example, because the result of a function declared const does not depend on anything other than the arguments passed in, the result of the function can cache that result and return any time the function is called with that same combination of arguments. (i.e. we know that the square of a number is constant, so we only need to compute it once).

unused

This attribute, attached to a function, means that the function is meant to be possibly unused. GCC will not produce a warning for this function.

The same effect can be accomplished with the __unused keyword. Declare this on parameters that are not used in the method implementation. Knowing that little bit of context allows the compiler to make optimizations accordingly. You're most likely to use __unused in delegate method implementations, since protocols frequently provide more context than is often necessary, in order to satisfy a large number of potential use cases.

LLVM

Like many features of GCC, Clang supports __attribute__, adding its own small set of extensions.

To check the availability of a particular attribute, you can use the __has_attribute directive.

availability

Clang introduces the availability attribute, which can be placed on declarations to describe the lifecycle of that declaration relative to operating system versions. Consider the function declaration for a hypothetical function f:

```
void f(void) __attribute__((availability(macosx,introduced=10.4,deprecated=10.6,obsoleted=10.7)));
```

The availability attribute states that f was introduced in OS X Tiger, deprecated in OS X Snow Leopard, and obsoleted in OS X Lion.

This information is used by Clang to determine when it is safe to use f: for example, if Clang is instructed to compile code for OS X Leopard, a call to f() succeeds. If Clang is instructed to compile code for OS X Snow Leopard, the call succeeds but Clang emits a warning specifying that the function is deprecated. Finally, if Clang is instructed to compile code for OS X Lion, the call fails because f() is no longer available.

The availability attribute is a comma-separated list starting with the platform name and then including clauses specifying important milestones in the declaration's lifetime (in any order) along with additional information.

* introduced: The first version in which this declaration was introduced.
* deprecated: The first version in which this declaration was deprecated, meaning that users should migrate away from this API.
* obsoleted: The first version in which this declaration was obsoleted, meaning that it was removed completely and can no longer be used.
* unavailable: This declaration is never available on this platform.
* message Additional message text that Clang will provide when emitting a warning or error about use of a deprecated or obsoleted declaration. Useful to direct users to replacement APIs.

Multiple availability attributes can be placed on a declaration, which may correspond to different platforms. Only the availability attribute with the platform corresponding to the target platform will be used; any others will be ignored. If no availability attribute specifies availability for the current target platform, the availability attributes are ignored.

Supported Platforms:

* ios: Apple's iOS operating system. The minimum deployment target is specified by the -mios-version-min=*version* or -miphoneos-version-min=*version* command-line arguments.
* macosx: Apple's OS X operating system. The minimum deployment target is specified by the -mmacosx-version-min=*version* command-line argument.

overloadable

Clang provides support for C++ function overloading in C. Function overloading in C is introduced using the overloadable attribute. For example, one might provide several overloaded versions of a tgsin function that invokes the appropriate standard function computing the sine of a value with float, double, or long double precision:

```
#include <math.h>
float __attribute__((overloadable)) tgsin(float x) { return sinf(x); }
double __attribute__((overloadable)) tgsin(double x) { return sin(x); }
long double __attribute__((overloadable)) tgsin(long double x) { return sinl(x); }
```

Note that overloadable only works for functions. You can overload method declarations to some extent by using generic return and parameter types, like id and void *.

Context is king when it comes to compiler optimizations. By providing constraints on how to interpret your code, you're increases the chance that the generated code is as efficient as possible. Meet your compiler half-way, and you'll always be rewarded.

And __attribute__ isn't just for the compiler either: The next person to see the code will appreciate the extra context, too. So go the extra mile for the benefit of your collaborator, successor, or just

2-years-from-now-(and-you've-forgotten-everything-about-this-code) you.

Because in the end, the love you take is equal to the love you make.
