
6.37 Specifying Attributes of Types

The keyword __attribute__ allows you to specify special attributes of struct and union types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Seven attributes are currently defined for types: aligned, packed, transparent_union, unused, deprecated, visibility, and may_alias. Other attributes are defined for functions (see Function Attributes) and for variables (see Variable Attributes).

You may also specify any one of these attributes with `__' preceding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use __aligned__ instead of aligned.

You may specify type attributes in an enum, struct or union type declaration or definition, or for other types in a typedef declaration.

For an enum, struct or union type, you may specify attributes either between the enum, struct or union tag and the name of the type, or just past the closing curly brace of the definition. The former syntax is preferred.

See Attribute Syntax, for details of the exact syntax for using attributes.

aligned (alignment)
: This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

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

force the compiler to insure (as far as it can) that each variable whose type is struct S or more_aligned_int will be allocated and aligned at least on a 8-byte boundary. On a SPARC, having all variables of type struct S aligned to 8-byte boundaries allows the compiler to use the ldd and std (doubleword load and store) instructions when copying one variable of type struct S to another, thus improving run-time efficiency.

Note that the alignment of any given struct or union type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of thestruct or union in question. This means that you can effectively adjust the alignment of a struct or union type by attaching an aligned attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire struct or union type.

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

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each short is 2 bytes, then the size of the entire struct S type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire struct S type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well. See below.

Note that the effectiveness of aligned attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying aligned(16) in an __attribute__ will still only provide you with 8 byte alignment. See your linker documentation for further information.

packed

This attribute, attached to struct or union type definition, specifies that each member (other than zero-width bitfields) of the structure or union is placed to minimize the memory required. When attached to an enum definition, it indicates that the smallest integral type should be used.

Specifying this attribute for struct and union types is equivalent to specifying the packed attribute on each of the structure or union members. Specifying the -fshort-enums flag on the line is equivalent to specifying the packed attribute on all enum definitions.

In the following example struct my_packed_struct's members are packed closely together, but the internal layout of its s member is not packed—to do that, struct my_unpacked_struct would need to be packed too.

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

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

You may only specify this attribute on the definition of an enum, struct or union, not on a typedef which does not also define the enumerated type, structure or union.

transparent_union

This attribute, attached to a union type definition, indicates that any function parameter having that union type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent union type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding argument can be any pointer expression. If the union member type is a pointer, qualifiers like const on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of the first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the wait function must accept either a value of type int * to comply with Posix, or a value of type union wait * to comply with the 4.1BSD interface. If wait's parameter were void *, wait would accept both kinds of arguments, but it would also accept any other pointer type and this would make argument type checking less useful. Instead, <sys/wait.h> might define the interface as follows:

```
typedef union __attribute__ ((__transparent_union__))
  {
    int *__ip;
    union wait *__up;
  } wait_status_ptr_t;

pid_t wait (wait_status_ptr_t);
```

This interface allows either int * or union wait * arguments to be passed, using the int * calling convention. The program can call wait with arguments of either type:

```
int w1 () { int w; return wait (&w); }
int w2 () { union wait w; return wait (&w); }
```

With this interface, wait's implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)
{
  return waitpid (-1, p.__ip, 0);
}
```

unused
: When attached to a type (including a union or a struct), this attribute means that variables of that type are meant to appear possibly unused. GCC will not produce a warning for any variables of that type, even if the variable appears to do nothing. This is often the case with lock or thread classes, which are usually defined and then not referenced, but contain constructors and destructors that have nontrivial bookkeeping functions.

deprecated
deprecated (msg)
: The deprecated attribute results in a warning if the type is used anywhere in the source file. This is useful when identifying types that are expected to be removed in a future version of a program. If possible, the warning also includes the location of the declaration of the deprecated type, to enable users to easily find further information about why the type is deprecated, or what they should do instead. Note that the warnings only occur for uses and then only if the type is being applied to an identifier that itself is not being declared as deprecated.

```
typedef int T1 __attribute__ ((deprecated));
T1 x;
typedef T1 T2;
T2 y;
typedef T1 T3 __attribute__ ((deprecated));
T3 z __attribute__ ((deprecated));
```

results in a warning on line 2 and 3 but not lines 4, 5, or 6. No warning is issued for line 4 because T2 is not explicitly deprecated. Line 5 has no warning because

T3 is explicitly deprecated. Similarly for line 6. The optional msg argument, which must be a string, will be printed in the warning if present.

The deprecated attribute can also be used for functions and variables (see Function Attributes, see Variable Attributes.)

may_alias
Accesses through pointers to types with this attribute are not subject to type-based alias analysis, but are instead assumed to be able to alias any other type of objects. In the context of 6.5/7 an lvalue expression dereferencing such a pointer is treated like having a character type. See -fstrict-aliasing for more information on aliasing issues. This extension exists to support some vector APIs, in which pointers to one vector type are permitted to alias pointers to a different vector type.

Note that an object of a type with this attribute does not have any special semantics.

Example of use:

```
typedef short __attribute__((__may_alias__)) short_a;

int
main (void)
{
  int a = 0x12345678;
  short_a *b = (short_a *) &a;

  b[1] = 0;

  if (a == 0x12345678)
    abort();

  exit(0);
}
```

If you replaced short_a with short in the variable declaration, the above program would abort when compiled with -fstrict-aliasing, which is on by default at -O2 or above in recent GCC versions.

visibility
In C++, attribute visibility (see Function Attributes) can also be applied to class, struct, union and enum types. Unlike other type attributes, the attribute must appear between the initial keyword and the name of the type; it cannot appear after the body of the type.

Note that the type visibility is applied to vague linkage entities associated with the class (vtable, typeinfo node, etc.). In particular, if a class is thrown as an exception in one shared object and caught in another, the class must have default visibility. Otherwise the two shared objects will be unable to use the same typeinfo node and exception handling will break.

To specify multiple attributes, separate them by commas within the double parentheses: for example, `__attribute__ ((aligned (16), packed))'.

6.37.1 ARM Type Attributes

On those ARM targets that support dllimport (such as Symbian OS), you can use the notshared attribute to indicate that the virtual table and other similar data for a class should not be exported from a DLL. For example:

```
class __declspec(notshared) C {
public:
  __declspec(dllimport) C();
  virtual void f();
}
```

```
__declspec(dllexport)
C::C() {}
```

In this code, C::C is exported from the current DLL, but the virtual table for C is not exported. (You can use __attribute__ instead of __declspec if you prefer, but most Symbian OS code uses__declspec.)

6.37.2 MeP Type Attributes

Many of the MeP variable attributes may be applied to types as well. Specifically, the based, tiny, near, and far attributes may be applied to either. The io and cb attributes may not be applied to types.

6.37.3 i386 Type Attributes

Two attributes are currently defined for i386 configurations: ms_struct and gcc_struct.

ms_struct
gcc_struct
: If packed is used on a structure, or if bit-fields are used it may be that the Microsoft ABI packs them differently than GCC would normally pack them. Particularly when moving packed data between functions compiled with GCC and the native Microsoft compiler (either via function call or as data in a file), it may be necessary to access either format.

  Currently -m[no-]ms-bitfields is provided for the Microsoft Windows X86 compilers to match the native Microsoft compiler.

6.37.4 PowerPC Type Attributes

Three attributes currently are defined for PowerPC configurations: altivec, ms_struct and gcc_struct.

For full documentation of the ms_struct and gcc_struct attributes please see the documentation in i386 Type Attributes.

The altivec attribute allows one to declare AltiVec vector data types supported by the AltiVec Programming Interface Manual. The attribute requires an argument to specify one of three vector types:vector__, pixel__ (always followed by unsigned short), and bool__ (always followed by unsigned).

```
__attribute__((altivec(vector__)))
__attribute__((altivec(pixel__))) unsigned short
__attribute__((altivec(bool__))) unsigned
```

These attributes mainly are intended to support the __vector, __pixel, and __bool AltiVec keywords.

6.37.5 SPU Type Attributes

The SPU supports the spu_vector attribute for types. This attribute allows one to declare vector data types supported by the Sony/Toshiba/IBM SPU Language Extensions Specification. It is intended to support the __vector keyword.