ISO/IEC JTC 1/SC 22/WG14 N1229

Potential Extensions For Inclusion In a Revision of ISO/IEC 9899

Introduction

This paper introduces some of the extensions made to standard ISO-C by the gcc and Microsoft compilers. As discussed during the Portland meeting in 2006, the working group agreed to examine potential extensions that are common existing practice to evaluate them for inclusion in a potential future revision. Much of the text in this paper is extracted from the relevant documentation of those compilers.

Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of foo ().

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type void, and thus effectively no value.)

This feature is especially useful in making macro definitions â€œsafeâ€ (so that they evaluate each operand exactly once). For example, the â€œmaximumâ€ function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either a or b twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here taken as int), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit-field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use typeof (see Typeof).

Jumping into a statement expression with goto or using a switch statement outside the statement expression with a case or default label inside the statement expression is not permitted. Jumping into a statement expression with a computed goto (see Labels as Values) yields undefined behavior. Jumping out of a statement expression is permitted, but if the statement expression is part of a larger expression then it is unspecified which other subexpressions of that expression have been evaluated except where the language definition requires certain subexpressions to be evaluated before or after the statement expression. In any case, as with a function call the evaluation of a statement expression is not interleaved with the evaluation of other parts of the containing expression. For example,

```
  foo (), (({ bar1 (); goto a; 0; }) + bar2 ()), baz();
```

will call foo and bar1 and will not call baz but may or may not call
bar2. If bar2 is called, it will be called after foo and before bar1.

NB - C++ compatibility issue:
In G++, the result value of a statement expression undergoes array
and function pointer decay, and is returned by value to the enclosing
expression. For instance, if A is a class, then

```
            A a;

            ({a;}).Foo ()
```

will construct a temporary A object to hold the result of the statement
expression, and that will be used to invoke Foo. Therefore the this
pointer observed by Foo will not be the address of a.

Any temporaries created within a statement within a statement expression
will be destroyed at the statement's end. This makes statement expressions
inside macros slightly different from function calls. In the latter case
temporaries introduced during argument evaluation will be destroyed
at the end of the statement that includes the function call. In the
statement expression case they will be destroyed during the statement
expression. For instance,

```
     #define macro(a)  ({__typeof__(a) b = (a); b + 3; })
     template<typename T> T function(T a) { T b = a; return b + 3; }

     void foo ()
     {
       macro (X ());
       function (X ());
     }
```

will have different places where temporaries are destroyed. For the macro
case, the temporary X will be destroyed just after the initialization
of b. In the function case that temporary will be destroyed when the
function returns.

These considerations mean that it is probably a bad idea to use
statement-expressions of this form in header files that are designed to
work with C++. (Note that some versions of the GNU C Library contained
header files using statement-expression that lead to precisely this bug.)

Locally Declared Labels

GCC allows you to declare local labels in any nested block scope. A
local label is just like an ordinary label, but you can only reference
it (with a goto statement, or by taking its address) within the block
in which it was declared.

A local label declaration looks like this:

```
     __label__ label;
```

or

```
     __label__ label1, label2, /* ... */;
```

Local label declarations must come at the beginning of the block, before
any ordinary declarations or statements.

The label declaration defines the label name, but does not define the
label itself. You must do this in the usual way, with label:, within
the statements of the statement expression.

The local label feature is useful for complex macros. If a macro contains
nested loops, a goto can be useful for breaking out of them. However,
an ordinary label whose scope is the whole function cannot be used:
if the macro can be expanded several times in one function, the label
will be multiply defined in that function. A local label avoids this
problem. For example:

```
     #define SEARCH(value, array, target)              \
     do {                                              \
       __label__ found;                                \
       typeof (target) _SEARCH_target = (target);      \
       typeof (*(array)) *_SEARCH_array = (array);     \
       int i, j;                                       \
       int value;                                      \
       for (i = 0; i < max; i++)                       \
         for (j = 0; j < max; j++)                     \
           if (_SEARCH_array[i][j] == _SEARCH_target)  \
             { (value) = i; goto found; }              \
       (value) = -1;                                   \
      found:;                                          \
     } while (0)
```

This could also be written using a statement-expression:

```
     #define SEARCH(array, target)                     \
     ({                                                \
       __label__ found;                                \
       typeof (target) _SEARCH_target = (target);      \
       typeof (*(array)) *_SEARCH_array = (array);     \
       int i, j;                                       \
       int value;                                      \
       for (i = 0; i < max; i++)                       \
         for (j = 0; j < max; j++)                     \
           if (_SEARCH_array[i][j] == _SEARCH_target)  \
             { value = i; goto found; }                \
       value = -1;                                     \
      found:                                           \
       value;                                          \
     })
```

Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator `&&'. The value has type void *. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
     void *ptr;
     /* ... */
     ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement (see footnote[1]), goto *exp;. For example,

```
     goto *ptr;
```

Any expression of type void * is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
     static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
     goto *array[i];
```

Note that this does not check whether the subscript is in bounds â€” array indexing in C never does that.

Such an array of label values serves a purpose much like that of the switch statement. The switch statement is cleaner, so use that rather than an array unless the problem does not fit a switch statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You may not use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

An alternate way to write the above example is

```
     static const int array[] = { &&foo - &&foo, &&bar - &&foo,
                                  &&hack - &&foo };
     goto *(&&foo + array[i]);
```

This is more friendly to code living in shared libraries, as it reduces the number of dynamic relocations that are needed, and by consequence, allows the data to be read-only.

Footnotes

[1] The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

Referring to a Type with typeof

Another way to refer to the type of an expression is with typeof. The syntax of using of this keyword looks like sizeof, but the construct acts semantically like a type name defined with typedef.

There are two ways of writing the argument to typeof: with an expression or with a type. Here is an example with an expression:

```
     typeof (x[0](1))
```

This assumes that x is an array of pointers to functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
     typeof (int *)
```

Here the type described is that of pointers to int.

If you are writing a header file that must work when included in ISO C programs, write __typeof__ instead of typeof. See Alternate Keywords.

A typeof-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of sizeof or typeof.

typeof is often useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe â€œmaximumâ€ macro that operates on any arithmetic type and evaluates each of its arguments exactly once:

```
     #define max(a,b) \
       ({ typeof (a) _a = (a); \
           typeof (b) _b = (b); \
         _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for a and b. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Some more examples of the use of typeof:

* This declares y with the type of what x points to.

```
             typeof (*x) y;
```

* This declares y as an array of such values.

```
typeof (*x) y[4];
```

* This declares y as an array of pointers to characters:

```
typeof (typeof (char *)[4]) y;
```

It is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using typeof, and why it might be a useful way to write, rewrite it with these macros:

```
#define pointer(T)  typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, array (pointer (char), 4) is the type of arrays of 4 pointers to char.

Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of x if that is nonzero; otherwise, the value of y.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

Case Ranges

You can specify a range of consecutive values in a single case label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual case labels, one for each integer value from low to high, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the ..., for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
     case 1...5:
```

Inquiring on Alignment of Types or Variables

The keyword __alignof__ allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like sizeof.

For example, if the target machine requires a double value to be aligned on an 8-byte boundary, then __alignof__ (double) is 8. This is true on many RISC machines. On more traditional machine designs, __alignof__ (double) is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd address. For these machines, __alignof__ reports the recommended alignment of a type.

If the operand of __alignof__ is an lvalue rather than a type, its value is the required alignment for its type, taking into account any minimum alignment specified with GCC's __attribute__ extension (see Variable Attributes). For example, after this declaration:

```
     struct foo { int x; char y; } foo1;
```

the value of __alignof__ (foo1.y) is 1, even though its actual alignment is probably 2 or 4, the same as __alignof__ (int).

It is an error to ask for the alignment of an incomplete type.

Once alignment as a concept/attribute is introduced one needs this

a) in macros, where the alignment of the element is not needed

b) in situations where aliasing needed and the actual type is not known

c) when calling posix_memalign

Thread-Local Storage

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. The run-time model GCC uses to implement this originates in the IA-64 processor-specific ABI, but has since been migrated to other processors as well. It requires significant support from the linker (ld), dynamic linker (ld.so), and system libraries (libc.so and libpthread.so), so it is not available everywhere.

At the user level, the extension is visible with a new storage class keyword: __thread. For example:

```
     __thread int i;
     extern __thread struct state s;
     static __thread char *p;
```

The __thread specifier may be used alone, with the extern or static specifiers, but with no other storage class specifier. When used with extern or static, __thread must appear immediately after the other storage class specifier.

The __thread specifier may be applied to any global, file-scoped static, function-scoped static, or static data member of a class. It may not be applied to block-scoped automatic or non-static data member.

When the address-of operator is applied to a thread-local variable, it is evaluated at run-time and returns the address of the current thread's instance of that variable. An address so obtained may be used by any thread. When a thread terminates, any pointers to thread-local

variables in that thread become invalid.

No static initialization may refer to the address of a thread-local variable.

There is more information on TLS at http://people.redhat.com/drepper/tls.pdf.

Attribute Syntax

This section describes the syntax with which __attribute__ may be used, and the constructs to which attribute specifiers bind, for the C language. Some details may vary for C++. Because of infelicities in the grammar for attributes, some forms described here may not be successfully parsed in all cases.

Note: There are some problems with the semantics of attributes in C++. For example, there are no manglings for attributes, although they may affect code generation, so problems may arise when attributed types are used in conjunction with templates or overloading. Similarly, typeid does not distinguish between types with different attributes. Support for attributes in C++ may be restricted in future to attributes on declarations only, but not on nested declarators.

***See Function Attributes, for details of the semantics of attributes ***applying to functions. See Variable Attributes, for details of the ***semantics of attributes applying to variables. See Type Attributes, ***for details of the semantics of attributes applying to structure, union ***and enumerated types.

An attribute specifier is of the form __attribute__ ((attribute-list)). An attribute list is a possibly empty comma-separated sequence of attributes, where each attribute is one of the following:

* Empty. Empty attributes are ignored.
* A word (which may be an identifier such as unused, or a reserved word such as const).
* A word, followed by, in parentheses, parameters for the attribute. These parameters take one of the following forms:
    - An identifier. For example, mode attributes use this form.
    - An identifier followed by a comma and a non-empty comma-separated list of expressions. For example, format attributes use this form.
    - A possibly empty comma-separated list of expressions. For example, format_arg attributes use this form with the list being a single integer constant expression, and alias attributes use this form with the list being a single string constant.

An attribute specifier list is a sequence of one or more attribute specifiers, not separated by any other tokens.

In GNU C, an attribute specifier list may appear after the colon following a label, other than a case or default label. The only attribute it makes sense to use after a label is unused. This feature is intended for code generated by programs which contains labels that may be unused but which is compiled with -Wall. It would not normally be appropriate to use in it human-written code, though it could be useful in cases where the code that jumps to the label is contained within an #ifdef conditional. GNU C++ does not permit such placement of attribute lists, as it is permissible for a declaration, which could begin with an attribute list, to be labelled in C++. Declarations cannot be labelled in C90 or C99, so the ambiguity does not arise there.

An attribute specifier list may appear as part of a struct, union or enum specifier. It may go either immediately after the struct, union or enum keyword, or after the closing brace. It is ignored if the content of the structure, union or enumerated type is not defined in the specifier in which the attribute specifier list is usedâ€”that is, in usages such as struct __attribute__((foo)) bar with no following opening brace. Where attribute specifiers follow the closing brace, they are considered to relate to the structure, union or enumerated type defined, not to any enclosing declaration the type specifier appears in, and the type defined is not complete until after the attribute specifiers.

Otherwise, an attribute specifier appears as part of a declaration, counting declarations of unnamed parameters and type names, and relates to that declaration (which may be nested in another declaration, for example in the case of a parameter declaration), or to a particular declarator within a declaration. Where an attribute specifier is applied to a parameter declared as a function or an array, it should apply to the function or array rather than the pointer to which the parameter is implicitly converted, but this is not yet correctly implemented.

Any list of specifiers and qualifiers at the start of a declaration may contain attribute specifiers, whether or not such a list may in that context contain storage class specifiers. (Some attributes, however, are essentially in the nature of storage class specifiers, and only make sense where storage class specifiers may be used; for example, section.) There is one necessary limitation to this syntax: the first old-style parameter declaration in a function definition cannot begin with an attribute specifier, because such an attribute applies to the function instead by syntax described below (which, however, is not yet implemented in this case). In some other cases, attribute specifiers are permitted by this grammar but not yet supported by the compiler. All attribute specifiers in this place relate to the declaration as a whole. In the obsolescent usage where a type of int is implied by the absence of type specifiers, such a list of specifiers and qualifiers may be an attribute specifier list with no other specifiers or qualifiers.

At present, the first parameter in a function prototype must have some type specifier which is not an attribute specifier; this resolves an ambiguity in the interpretation of void f(int (__attribute__((foo)) x)), but is subject to change. At present, if the parentheses of a function declarator contain only attributes then those attributes are ignored, rather than yielding an error or warning or implying a single parameter of type int, but this is subject to change.

An attribute specifier list may appear immediately before a declarator (other than the first) in a comma-separated list of declarators in a declaration of more than one identifier using a single list of specifiers and qualifiers. Such attribute specifiers apply only to the identifier before whose declarator they appear. For example, in

```
__attribute__((noreturn)) void d0 (void),
    __attribute__((format(printf, 1, 2))) d1 (const char *, ...),
     d2 (void)
```

the noreturn attribute applies to all the functions declared; the format attribute only applies to d1.

An attribute specifier list may appear immediately before the comma, = or semicolon terminating the declaration of an identifier other than a function definition. At present, such attribute specifiers apply to the declared object or function, but in future they may attach to the outermost adjacent declarator. In simple cases there is no difference, but, for example, in

```
void (****f)(void) __attribute__((noreturn));
```

at present the noreturn attribute applies to f, which causes a warning since f is not a function, but in future it may apply to the function ****f. The precise semantics of what attributes in such cases will apply to are not yet specified. Where an assembler name for an object or function is specified (see Asm Labels), at present the attribute must follow the asm specification; in future, attributes before the asm specification may apply to the adjacent declarator, and those after it to the declared object or function.

An attribute specifier list may, in future, be permitted to appear after the declarator in a function definition (before any old-style parameter declarations or the function body).

Attribute specifiers may be mixed with type qualifiers appearing inside the [] of a parameter array declarator, in the C99 construct by which such qualifiers are applied to the pointer to which the array is implicitly converted. Such attribute specifiers apply to the pointer, not to the array, but at present this is not implemented and they are ignored.

An attribute specifier list may appear at the start of a nested declarator. At present, there are some limitations in this usage: the attributes correctly apply to the declarator, but for most individual attributes the semantics this implies are not implemented. When attribute specifiers follow the * of a pointer declarator, they may be mixed with any type qualifiers present. The following describes the formal semantics of this syntax. It will make the most sense if you are familiar with the formal specification of declarators in the ISO C standard.

Consider (as in C99 subclause 6.7.5 paragraph 4) a declaration T D1, where T contains declaration specifiers that specify a type Type (such as int) and D1 is a declarator that contains an identifier ident. The type specified for ident for derived declarators whose type does not include an attribute specifier is as in the ISO C standard.

If D1 has the form (attribute-specifier-list D), and the declaration T D specifies the type â€œderived-declarator-type-list Typeâ€ for ident, then T D1 specifies the type â€œderived-declarator-type-list attribute-specifier-list Typeâ€ for ident.

If D1 has the form * type-qualifier-and-attribute-specifier-list D, and the declaration T D specifies the type â€œderived-declarator-type-list Typeâ€ for ident, then T D1 specifies the type â€œderived-declarator-type-list type-qualifier-and-attribute-specifier-list Typeâ€ for ident.

For example,

```
     void (__attribute__((noreturn)) ****f) (void);
```

specifies the type â€œpointer to pointer to pointer to pointer to non-returning function returning voidâ€ . As another example,

```
     char *__attribute__((aligned(8))) *f;
```

specifies the type â€œpointer to 8-byte-aligned pointer to charâ€ . Note again that this does not work with most attributes; for example, the usage of `aligned' and `noreturn' attributes given above is not yet supported.

For compatibility with existing code written for compiler versions that did not implement attributes on nested declarators, some laxity is allowed in the placing of attributes. If an attribute that only applies to types is applied to a declaration, it will be treated as applying to the type of that declaration. If an attribute that only applies to declarations is applied to the type of a declaration, it will be treated as applying to that declaration; and, for compatibility with code placing the attributes immediately before the identifier declared, such an attribute applied to a function return type will be treated as applying to the function type, and such an attribute applied to an array element type will be treated as applying to the array type. If an attribute that only applies to function types is applied to a pointer-to-function type, it will be treated as applying to the pointer target type; if such an attribute is applied to a function return type that is not a pointer-to-function type, it will be treated as applying to the function type.

Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword __attribute__ allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently defined for functions: noreturn, returns_twice, pure, warn_unused_result, and nonnull.

noreturn
    A few standard library functions, such as abort and exit, cannot return. GCC knows this automatically. Some programs define their

own functions that never return. You can declare them noreturn to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));

void
fatal (/* ... */)
{
  /* ... */ /* Print error message. */ /* ... */
  exit (1);
}
```

The noreturn keyword tells the compiler to assume that fatal cannot return. It can then optimize without regard to what would happen if fatal ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

The noreturn keyword does not affect the exceptional path when that applies: a noreturn-marked function may still return to the caller by throwing an exception or calling longjmp.

Do not assume that registers saved by the calling function are restored before calling the noreturn function.

It does not make sense for a noreturn function to have a return type other than void.

returns_twice

The returns_twice attribute tells the compiler that a function may return more than one time. The compiler will ensure that all registers are dead before calling such a function and will emit a warning about the variables that may be clobbered after the second return from the function. Examples of such functions are setjmp and vfork. The longjmp-like counterpart of such function, if any, might need to be marked with the noreturn attribute.

pure

Many functions have no effects except the return value and 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. These functions should be declared with the attribute pure. For example,

```
int square (int) __attribute__ ((pure));
```

says that the hypothetical function square is safe to call fewer times than the program says.

Some of common examples of pure functions are strlen or memcmp. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between two consecutive calls (such as feof in a multithreading environment).

warn_unused_result

The warn_unused_result attribute causes a warning to be emitted if a caller of the function with this attribute does not use its return value. This is useful for functions where not checking the result is either a security problem or always a bug, such as realloc.

```
int fn () __attribute__ ((warn_unused_result));
int foo ()
{
  if (fn () < 0) return -1;
  fn ();
  return 0;
}
```

results in warning on line 5.

nonnull (arg-index, ...)
The nonnull attribute specifies that some function parameters should be non-null pointers. For instance, the declaration:

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

causes the compiler to check that, in calls to my_memcpy, arguments dest and src are non-null. If the compiler determines that a null pointer is passed in an argument slot marked as non-null a warning is issued. The compiler may also choose to make optimizations based on the knowledge that certain function arguments will not be null.

If no argument index list is given to the nonnull attribute, all pointer arguments are marked as non-null. To illustrate, the following declaration is equivalent to the previous example:

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

Specifying Attributes of Variables

The keyword __attribute__ allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Some attributes are currently defined generically for variables. Other attributes are defined for variables on particular target systems. Other attributes are available for functions (see Function Attributes) and for types (see Type Attributes).

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

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

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));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the declared variable or field 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 or fields that you have aligned this way.

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.

cleanup (cleanup_function)
The cleanup attribute runs a function when the variable goes out of scope. This attribute can only be applied to auto function scope variables; it may not be applied to parameters or variables with static storage duration. The function must take one parameter, a pointer to a type compatible with the variable. The return value of the function (if any) is ignored.
*** COMPETES WITH THE MSVC try/finally APPROACH

unused
This attribute, attached to a variable, means that the variable is meant to be possibly unused.

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. Six attributes are currently defined for types: aligned, packed, transparent_union, unused, deprecated 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 the aligned and transparent_union attributes either in a typedef declaration or just past the closing curly brace of a complete enum, struct or union type definition and the packed attribute only past the closing brace of a definition.

You may also specify attributes between the enum, struct or union tag and the name of the type rather than after the closing brace.

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 the struct 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 packedâ€”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 a enum, struct or union, not on a typedef which does not also define the enumerated type, structure or union.

transparent_union

*** NOT SURE IF THIS IS APPROPRIATE

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

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.

The try-except Statement

The try-except statement is a Microsoft extension to the C language that enables applications to gain control of a program when events that normally terminate execution occur. Such events are called exceptions, and the mechanism that deals with exceptions is called structured exception handling.

Exceptions can be either hardware- or software-based. Even when applications cannot completely recover from hardware or software exceptions, structured exception handling makes it possible to display error information and trap the internal state of the application to help diagnose the problem. This is especially useful for intermittent problems that cannot be reproduced easily.

Syntax

try-except-statement:

```
__try compound-statement

__except ( expression ) compound-statement
```

The compound statement after the __try clause is the guarded section. The compound statement after the __except clause is the exception handler. The handler specifies a set of actions to be taken if an exception is raised during execution of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the __except clause.
3. If an exception occurs during execution of the guarded section or in any routine the guarded section calls, the__except expression is evaluated and the value returned determines how the exception is handled. There are three values:

   EXCEPTION_CONTINUE_SEARCH
   Exception is not recognized. Continue to search up the stack for a handler, first for containing try-except statements, then for handlers with the next highest precedence.

   EXCEPTION_CONTINUE_EXECUTION
   Exception is recognized but dismissed. Continue execution at the point where the exception occurred.

   EXCEPTION_EXECUTE_HANDLER
   Exception is recognized. Transfer control to the exception handler by executing the __except compound statement, then continue execution at the point the exception occurred.

Because the __except expression is evaluated as a C expression, it is limited to a single value, the conditional-expression operator, or the

comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

Note: Structured exception handling works with C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, the C++ exception handling mechanism is much more flexible, in that it can handle exceptions of any type.

Note: For C++ programs, C++ exception handling should be used instead of structured exception handling. For more information, see Exception Handling in the C++ Language Reference.

Each routine in an application can have its own exception handler. The __except expression executes in the scope of the __try body. This means it has access to any local variables declared there.

The __leave keyword is valid within a try-except statement block. The effect of __leave is to jump to the end of the try-except block. Execution resumes after the end of the exception handler. Although a goto statement can be used to accomplish the same result, a goto statement causes stack unwinding. The __leave statement is more efficient because it does not involve stack unwinding.

Exiting a try-except statement using the longjmp run-time function is considered abnormal termination. It is illegal to jump into a __try statement, but legal to jump out of one. The exception handler is not called if a process is killed in the middle of executing a try-except statement.

Example

Following is an example of an exception handler and a termination handler. See The try-finally Statement for more information about termination handlers.

```
.
.
.
puts("hello");
__try{
   puts("in try");
   __try{
      puts("in try");
      RAISE_AN_EXCEPTION();
   }__finally{
      puts("in finally");
   }
}__except( puts("in filter"), EXCEPTION_EXECUTE_HANDLER ){
   puts("in except");
}
puts("world");
```

This is the output from the example, with commentary added on the right:

```
hello
in try              /* fall into try                     */
in try              /* fall into nested try                */
in filter           /* execute filter; returns 1 so accept  */
in finally          /* unwind nested finally               */
in except           /* transfer control to selected handler */
world               /* flow out of handler                 */
```

The try-finally Statement

Microsoft Specific

The try-finally statement is a Microsoft extension to the C language that enables applications to guarantee execution of cleanup code when execution of a block of code is interrupted. Cleanup consists of such tasks as deallocating memory, closing files, and releasing file

handles. The try-finally statement is especially useful for routines that have several places where a check is made for an error that could cause premature return from the routine.

try-finally-statement:

    __try compound-statement

    __finally compound-statement

The compound statement after the __try clause is the guarded section. The compound statement after the __finally clause is the termination handler. The handler specifies a set of actions that execute when the guarded section is exited, whether the guarded section is exited by an exception (abnormal termination) or by standard fall through (normal termination).

Control reaches a __try statement by simple sequential execution (fall through). When control enters the __try statement, its associated handler becomes active. Execution proceeds as follows:

1. The guarded section is executed.
2. The termination handler is invoked.
3. When the termination handler completes, execution continues after the __finally statement. Regardless of how the guarded section ends (for example, via a goto statement out of the guarded body or via a return statement), the termination handler is executed before the flow of control moves out of the guarded section.

The __leave keyword is valid within a try-finally statement block. The effect of __leave is to jump to the end of the try-finally block. The termination handler is immediately executed. Although a goto statement can be used to accomplish the same result, a goto statement causes stack unwinding. The __leave statement is more efficient because it does not involve stack unwinding.

Exiting a try-finally statement using a return statement or the longjmp run-time function is considered abnormal termination. It is illegal to jump into a __try statement, but legal to jump out of one. All __finally statements that are active between the point of departure and the destination must be run. This is called a "local unwind."

The termination handler is not called if a process is killed while executing a try-finally statement.

Note: Structured exception handling works with C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, the C++ exception handling mechanism is much more flexible, in that it can handle exceptions of any type.

Note: For C++ programs, C++ exception handling should be used instead of structured exception handling. For more information, see Exception Handling in the C++ Language Reference.

See the example for the try-except statement to see how the try-finally statement works.

        *** COMPETES WITH THE gcc attribute((cleanup)) APPROACH