The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.3 Type qualifiers

type qualifier syntax

type-qualifier:
const
restrict
volatile

Commentary

Rationale

Type qualifiers were introduced in part to provide greater control over optimization. Several important optimization techniques are based on the principle of "cacheing": under certain circumstances the compiler can remember the last value accessed (read or written) from a location, and use this retained value the next time that location is read. (The memory, or "cache", is typically a hardware register.) If this memory is a machine register, for instance, the code can be smaller and faster using the register rather than accessing external memory.

C90

Support for **restrict** is new in C99.

C++

Support for **restrict** is new in C99 and is not specified in the C++ Standard.

Other Languages

Some languages use the keyword **read**, or **readonly** as a type qualifier to indicate that an object can only be read from. BCPL uses **MANIFEST**.

Common Implementations

The keyword **noalias** was included in some drafts of the C90 Standard. It provided functionality whose intended use was similar to that provided by the keyword **restrict** in C99.

Coding Guidelines

declaration specifier ordering ??

A guideline on the relative order of type qualifiers within a declaration specifier is given elsewhere.

Usage

Developers do not always make full use of the **const** qualifier. An automated analysis[5] of programs whose declarations contained a relatively high percentage (29%) of **const** qualifiers found that it would have been possible to declare 70% of the declarations using this qualifier. Engblom[4] reported that for real-time embedded C code 17% of object declarations contained the **const** type qualifier.

Table 1476.1: Common token sequences containing *type-qualifier*s (as a percentage of each *type-qualifier*). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
; volatile	0.1	36.1	**{ const**	0.2	5.6
, const	0.2	32.8	**const unsigned**	6.2	1.4
(const	0.2	28.1	**const struct**	11.1	1.3
(volatile	0.0	26.2	**volatile unsigned**	25.6	1.1
; const	0.1	14.1	**const void**	5.3	0.8
identifier **volatile**	0.0	11.4	**volatile struct**	15.5	0.4
{ volatile	0.1	11.0	**volatile int**	7.4	0.1
const char	54.1	10.4	**volatile** identifier	36.2	0.0
static const	1.5	10.0	**volatile (**	8.9	0.0
static volatile	0.3	8.6	**const** identifier	17.6	0.0

Constraints

1477 Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

restrict constraint

Commentary

The specification of the **restrict** qualifier only defines the behavior for pointers that refer to objects (because it is only intended to deal with such quantities). In the case of function parameters having array type (e.g., `void f(float a[restrict][100]);`) the implicit conversion to pointer type occurs before this constraint is applied (this intent is expressed in WG14/N521, an example in the standard, and the rationale).

restrict formal definition
array converted to pointer
EXAMPLE compatible function prototypes
function declarator static

Semantics

1478 The properties associated with qualified types are meaningful only for expressions that are lvalues.[112)]

qualifier meaningful for lvalues

Commentary

Although type qualifiers are specified in terms of creating new type, they really act as modifiers of the declarator. Use of these keywords gives a type additional properties. However, they do not change its representation or alignment requirements. These properties are associated with the declared object, not its value (although they all specify something about the possible attributes of the values by the objects declared using them). A qualified type can be applied to a non-lvalue through the use of a cast operator (e.g., `(const long)1`).

qualifiers representation and alignment

C++

The C++ Standard also associates properties of qualified types with rvalues (3.10p3). Such cases apply to constructs that are C++ specific and not available in C.

Common Implementations

Some implementations support directives that allow the developer to specify which areas of storage objects or literals are to be held in.

Coding Guidelines

The fact that type qualifiers are only meaningful for lvalue expressions does not prevent developers using them in other contexts. Such usage is redundant and does not affect how a translator should interpret the behavior of a program (the issue of redundant code is discussed elsewhere). Is there a worthwhile benefit recommending against the use of type qualifiers in contexts where they have no meaning? The usage, in the visible source and not via a typedef name, suggests that the author may believe it has some affect on the behavior of the program. In practice such usage is very rare and the developer misconception harmless.

redundant code

1479 If the same qualifier appears more than once in the same *specifier-qualifier-list*, either directly or via one or more **typedef**s, the behavior is the same as if it appeared only once.

qualifier appears more than once

Commentary

Having the same qualifier appear more than once in the same *specifier-qualifier-list* may be redundant, but it is harmless. Support for such usage can simplify the automatic generation of C source code and reduce the amount of coordination needed between different development groups (e.g., over who is responsible for ensuring a given qualifier appears in a chain of typedef names; qualifiers can appear on in typedef name defined by a group, independent of any references it makes to typedef names defined by other development groups). It can also reduce maintenance costs for existing source (e.g., a change to the definition of a typedef name does not have any cascading affects on any existing declarations it appears in with qualifiers). The same permission is given for function specifiers.

function specifier appears more than once

C90

The following occurs within a Constraints clause.

The same type qualifier shall not appear more than once in the same specifier list or qualifier list, either directly or via one or more **typedef***s.*

Source code containing a declaration with the same qualifier appearing more than once in the same *specifier-qualifier-list* will cause a C90 translator to issue a diagnostic.

C++

7.1.5p1 *However, redundant cv-qualifiers are prohibited except when introduced through the use of typedefs (7.1.3) or template type arguments (14.3), in which case the redundant cv-qualifiers are ignored.*

The C++ Standard does not define the term *prohibited*. Applying common usage to this term suggests that it is to be interpreted as a violation of a diagnosable (because "no diagnostic is required", 1.4p1, has not been specified) rule.

The C++ specification is intermediate between that of C90 and C99.

```
1  const const int cci; /* does not change the conformance status of program */
2                       // in violation of a diagnosable rule
```

Coding Guidelines

Is there a worthwhile benefit recommending against having the same qualifier appears more than once in the same *specifier-qualifier-list*?

There is an obvious organizational and maintenance benefit in allowing the same qualifier to occur more than once via typedefs. Having the same qualifier appear more than once in the visible source of a *specifier-qualifier-list* suggests that insufficient attention was invested by the original author. This usage also requires subsequent readers to invest more cognitive effort in comprehending the declaration. However, support for this usage is new in C99 and measurements of source show an underuse of the most common qualifier (i.e., **const**). Overuse of qualifiers does not look like it will be an issue that needs addressing via a guideline. Such usage is redundant and does not affect the behavior of a program (the issue of redundant code is discussed elsewhere).

redundant code

Example

```
1  const const int glob_1;
2  const int const long const extern const signed const glob_2;
3
4  typedef const int C_I;
5  typedef const C_I const_C_I;
```

---

const qualified attempt modify

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. 1480

Commentary

assignment operator
modifiable lvalue
restrict
formal definition

Modifying a const-qualified object requires the use of a pointer to it (plus an appropriate cast operation; attempting to modify the object using its declared type is a constraint violation). The undecidability of detecting all such pointer usages at translation time and the overhead of performing the check during program execution resulted in the committee specifying the behavior as undefined, rather than a constraint violation.

The author of the source may intend a use of the **const** qualifier to imply a read-only object, and a translator may treat it as such. However, a translator is not required to perform any checks at translation or execution time to ensure that the object is not modified (via some pointer to it).

C++

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [Example: a member function called for an object (9.3) can modify the object.] 3.10p10

The C++ Standard specifies a different linkage for some objects declared using a const-qualified type. static internal linkage

Other Languages

Languages that contain some form of readonly qualifier usually specify that so qualified objects cannot be modified and attempts to perform such modifications are treated as errors. There is usually some indirect mechanism that will have the effect of modifying such objects, and few implementations are required to detect such modifications during program execution.

Common Implementations

Some freestanding implementations place static storage duration, const-qualified, objects in read-only memory. While it might be possible to write to this memory, the value of objects held there are not modified by such operations. Some hosted implementations place static storage duration, const-qualified, objects in a region of storage marked as read-only (some processor memory management units support such types of memory and automatically perform checks on accesses to it).

Many implementations place const-qualified objects in the same kind of storage as other objects. However, this does not mean that if these qualified objects are modified, the modified value is actually used by a program. A translator may reduce optimization overhead by assuming const-qualified objects are never modified. This could result in values held in registers being reused, after the object held in storage had been modified.

Coding Guidelines

There are many different ways of attempting to modify a const-qualified object. Enumerating all such cases would create an unnecessarily large number of guidelines, and is not guaranteed to catch all cases. To be able to attempt to modify a const-qualified object, without a translator issuing a diagnostic, it is necessary to use an explicit cast. A single guideline covering this case is discussed elsewhere and deals with the issue at the point potential for problems starts. pointer conversion constraints

Example

```
1  extern const int g_1 = 3;
2  extern int g_2;
3
4  void f(const int *pi);
5  /*
6   * The declaration of pi implies that the statements within f will
7   * not modify its value.  However, an optimizer cannot assume that
8   * the value *pi will remain unchanged throughout the execution of
9   * f.  For instance the function f may modify g_2; whose address
10  * is passed as an argument in the second call below.
11  */
12
13 void g(void)
14 {
15 const int *loc_1 = &g_1;
16 int *loc_2 = &g_2;
17
18 f(loc_1);
19 f((const int *)loc_2);
20 }
```

Also see an example elsewhere. EXAMPLE const pointer

volatile qualified attempt modify

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.[113)] 1481

Commentary

const qualified attempt modify 1480

The issues are the same as for the const-qualified case.

Common Implementations

The extent to which the behavior, in this case, differs from that intended will depend on the optimizations performed by an implementation. If an implementation has reused a value, held in a register because it was accessed via a non-volatile-qualified type, there will be no access to the volatile-qualified object. Presumably the object was declared to have volatile-qualified type because accesses to it caused some external affect. The difference on the behavior of a program, because of less accesses to such a qualified object can only be guessed at.

Coding Guidelines

pointer conversion constraints

Like the const-qualified case, the appropriate guideline recommendation is discussed elsewhere.

Example

```
1  extern volatile int glob_1;
2
3  void f(int *p_1)
4  {
5  (*p_1)++;
6  (*p_1)++;
7  }
8
9  void g(void)
10 {
11 f((int *)glob_1);
12 }
```

volatile last-stored value

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. 1482

Commentary

The **volatile** qualifier informs the implementation that it would be unsafe to optimize accesses to objects declared using it.

Other Languages

The **volatile** field modifier in Java is used to provide a method of ensuring that accesses to objects whose values may be modified by different threads are reconciled with the master copy in shared memory.

Common Implementations

The **volatile** qualifier is mainly seen in freestanding implementations and hosted implementations where storage is shared between multiple programs. The following are some of the behaviors often seen for objects declared using this qualifier:

- Updating an object on a periodic basis (e.g., a realtime clock).
- Modifying an object after every read from it (e.g., instance, an input device sending characters down a serial line).
- An object may not contain the value last written to it (e.g., an output port, where reading from that location returns the status of the write operation).

The only meaningful volatile-qualified objects are often declared by the implementation, as part of a vendor supplied API. This is because having an object modified in ways unknown to the implementation usually involves associating it with some external device. Such associations usually require making use of some implementation provided language extension, or by passing the address of the object to a vendor supplied library function.

Coding Guidelines

Qualifying an object declaration with **volatile** may inform the translator that it may be modified in ways unknown, but how well will developers understand the implications of the changing value? For instance, in:

```
1  extern volatile long seconds_since_midnight;
2
3  void f(void)
4  {
5  int now_hours = (seconds_since_midnight / 3600);
6  int now_minutes = (seconds_since_midnight / 60);
7  int now_seconds = (seconds_since_midnight % 60);
8  }
```

the time at the start of the evaluation of the expression assigned to `now_hours` may have been 9:59:59 and at the start of the evaluation of the expression assigned to `now_minutes` 10:00:00. The values of the three objects would then be identical to the time obtained at 9:00:00.

Rev 1482.1

A sequence of related expressions that accesses the value of the same volatile-qualified object shall be checked to ensure the changeability of the volatile nature of the objects value has been taken into account.

1483 Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3.

Commentary

This is a requirement on the implementation. The possibility of a third-party library configuring a hardware device to update the contents of an object, whose address was passed as a parameter, means that implementations cannot assume that volatile-qualified objects contain known values, just because the implementation provides no mechanism for associating them with a hardware device.

C++

There is no equivalent statement in the C++ Standard. But it can be deduced from the following two paragraphs:

A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input. 1.9p5

The observable behavior of the abstract machine is its sequence of reads and writes to **volatile** *data and calls to library I/O functions.*[6)] 1.9p6

1484 Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.[114)]

Commentary

This is a requirement on the implementation. As well as performing the same number of accesses required by the abstract machine, updates to the stored value must have occurred in the order in which sequence points are reached (multiple accesses to the same volatile-qualified object between successive sequence points results in undefined behavior). The extent to which this requirement gives developers the ability to predict the value last stored in an object will depend on the uniqueness of the ordering of sequence points. These issues are also discussed in a C Standard example given elsewhere.

object modified once between sequence points
sequence points
abstract machine
example

---

footnote 112

112) The implementation may place a **const** object that is not **volatile** in a read-only region of storage. 1485

Commentary

The following are some of possible benefits of using read-only storage:

- the cost of ROM is significantly lower than that of RAM,
- the contents of ROM are not lost when power to the computer is switched off, and
- it may be possible to cause write accesses to regions of storage that an operating system has marked as being read-only to raise an exception that can be caught by a program debugging tool.

const volatile 1496
EXAMPLE

In some cases a **const** object that is **volatile** can be in a read-only region of storage.

C++

The C++ Standard does not make this observation.

Common Implementations

Sometimes the storage is physically read-only, as in ROM hardware. Sometimes implementations make use of a host operating system's ability to configure regions of storage as read-only, after the program image has been loaded, and enforcing that requirement (invariably through hardware assisted, memory management support).

Moreover, the implementation need not allocate storage for such an object if its address is never used. 1486

Commentary

An implementation need not allocate storage for any object whose address is never used. The difference with const-qualified objects is that a translator often knows in advance what their value is going to be, through out the execution of a program (it appears in the initialization of the objects definition). If this value is a constant expression the translator can substitute it wherever the identifier denoting the const-qualified object occurs. A translator could treat members of a const-qualified structure object in a similar fashion. If such substitutions were made, the undefined behavior associated with any attempts to modify the value being to have no effect.

const 1480 qualified
attempt modify

C++

The C++ Standard does not make this observation. However, given C++ supports zero sized objects, 1.8p5, there may be other cases where implementations need not allocate storage.

Common Implementations

Replacing references to objects by their known values, at a given point in a program, is not limited to const-qualified objects. However, significantly more analysis is needed for translators to perform this kind of substitution on references to non-const-qualified objects.

---

footnote 113

113) This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address). 1487

Commentary

Such objects may exist in the addressable storage of a program and accessed by casting an integer value, representing that address, to a pointer type. The properties associated with the qualification of a pointed-to type always apply, no matter what location in storage is pointed at.

C++

The C++ Standard does not make this observation.

1488 What constitutes an access to an object that has volatile-qualified type is implementation-defined.

Commentary

This implementation-defined behavior is caused by the different possible ways an implementation can access an objects having bit-field types.

```
1   extern volatile int glob;
2
3   struct {
4           volatile unsigned int m_1 :5;
5                    unsigned int m_2 :5;
6           volatile unsigned int m_3 :5;
7          } x;
8
9   void f(void)
10  {
11  glob = glob + 1;
12  glob++;
13
14  x.m_1 = 2;
15  x.m_2 = 3;
16  int loc = x.m_2 + x.m_2;
17  }
```

The first increment of `glob` would generally be considered to involve two accesses. The second increment of `glob` could be performed in one access on some processors (those whose having an increment memory instruction), but two accesses on other processors (the majority, which effectively perform the sequence load/increment/store).

In the second set of assignments the access methods are likely to be fundamentally different. One or more bit-fields might be packed into the same storage unit. The assignment to `m_1` may need to read from the storage unit that holds its value, to obtain the value of the bits representing the member `m_2` (so they can be bitwise-AND'ed with the new value, unless the processor supports some form of bit-field access instruction). Assignments to non-bit-field objects do not normally involve a read of their value. An even more surprising access could occur through the assignment to `m_2`, which could also cause the members `m_1` or `m_3` to be accessed, even although they are not explicitly mentioned in the assignment.

bit-field packed into

bit-field in expression

Volatile-qualified objects can also be affected by translator optimizations. The evaluation of the expression assigned to `loc` could cause the value of `x.m_2` to be held in a register, so it would not need to be loaded for the second access. This optimization would potentially result in one fewer accesses to `x.m_1`.

C++

The C++ Standard does not explicitly specify any behavior.

[Note: . . . In general, the semantics of **`volatile`** *are intended to be the same in C++ as they are in C.]* 7.1.5.1p8

Common Implementations

In those implementations that support volatile-qualified objects whose value is modified in ways unknown to the implementation, the objects have non-bit-field scalar types.

Coding Guidelines

This implementation-defined behavior has the ability to generate many surprises for developers. However, implementations that support volatile-qualified objects whose value is modified in ways unknown to the implementation and whose object representation only occupies part of a storage unit are very rare. For this reason no guideline recommendations are made here.

An object that is accessed through a restrict-qualified pointer has a special association with that pointer. 1489

Commentary

The emerging common terminology usage is the term *restricted pointer* (which is also used in the standard) rather than *restrict-qualified pointer*.

restrict 1477 constraint

It is not possible to declare a restrict-qualified object, so all restrict-qualified pointers are created by the conversion of a non-restrict-qualified address. The equivalent association for objects accessed via their declared name can be obtained by using the **`register`** storage class in the objects declaration. Because it is not possible to take the address of such an object it cannot have any aliases.

unary & operand constraints

As the specification of what an object is composed of, and an example given in the standard show, the object accessed through a restrict-qualified pointer could be part of a larger object.

object contiguous sequence of bytes
restrict example 2

C90

Support for the **`restrict`** qualifier is new in C99.

C++

Support for the **`restrict`** qualifier is new in C99 and is not available in C++.

Other Languages

Optimizing the performance of identical operations on array objects is one of the intended uses of restrict-qualified pointers. Some languages (Fortran 90, Ada) achieve the same result by supporting operations that can be applied to every element in an array, or slice of an array.

---

restrict requires all accesses

This association, defined in 6.7.3.1 below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.[115)] 1490

Commentary

lifetime of object
restrict pointer lifetime

This association only applies within the lifetime of the restricted pointer object, which may be a subset of the lifetime of the pointed-to object. It enables optimizations to be localized to accesses of an object within a particular scope (e.g., a function definition or loop). The optimization being driven by the method used to access the object, rather than the object itself.

There is no requirement on translators to check that a particular restricted pointer is the only method used to access the pointed-to object. Such a requirement would be counter productive, since the purpose of this qualifier is to overcome the technical difficulties associated with deducing the information it denotes automatically (although algorithms for automatically deducing some cases are known[1]). If this requirement is not met the behavior is undefined. A translator that relies on the presence of the **`restrict`** qualifier, to perform optimizations, may produce different results in this case, than if the qualifier did not appear in the pointer declaration.

restrict undefined behavior

This association only applies if the access is through a restricted pointer. Assigning the value of a restricted pointer to a non-restricted pointer does not cause the special association to also be assigned. For instance, in the following example, accesses through the pointers `p` and `q` are not restrict-qualified.

```
1  void f(float * restrict r,
2         float * restrict s,
3         int len)
4  {
5  float *p = r,
```

```
6         *q = s;
7
8   while(len-- > 0)
9      *p++ = *q++;
10  }
```

Common Implementations

The **restrict** qualifier specifies behavior that involves a relationship between one pointer and all other pointers. An alternative approach is to specify relationships between sets of pointers (which may only be a relationship between two pointers). Such an approach enables aliasing behaviors to be specified that would not be possible using the **restrict** qualifier. Koes, Budiu, Venkataramani and Goldstein[9] produced a tool which analyses source for sets of pointers which were likely, but not guaranteed, to be independent of each other and an estimated optimization benefit if a compiler could assume they were independent. They found that in many cases developers needed only a few minutes to verify whether the pointers were actually independent and that worthwhile increases in program speed were possible (they modified `gcc` to accept and use pointer information provided by a new pragma).

Coding Guidelines

Use of the **restrict** qualifier relies on the original use meeting, and subsequent developers maintaining, the guarantee required by the standard. While it may be possible to use static analysis tools to verify the requirement in some cases (where the analysis does not consume huge amounts of resources), the availability of such tools is open to question. Without a guaranteed method of enforcement, and no established practices for avoiding the problem, no guideline is given here.

Example

When defining a macro to take the place of a function, it is not necessary to be concerned with the scope of the argument passed, provided the lifetime of the pointer used is that of the body of the macro. In the following example, the claim being made by the restricted pointer only needs to exist within the block created by the macro body.

```
1   float *c;
2
3   #define WG14_N448(N, A, B)      \
4      {                            \
5      int n = (N);                 \
6      float *restrict a = (A);     \
7      float *const b = (B);        \
8      int i;                       \
9      for (i = 0; i < n; i++)      \
10        a[i] = b[i] + c[i];       \
11     }
```

In the following two examples the objects accessed by the restricted pointers are subobjects of a larger whole. There can be so called *edge effects* where the two subobject storage areas meet. At this edge point it is possible that accesses to elements from both subobjects will be loaded into the same cache line. The affect may be that assumptions about cache line interaction, made when deciding what machine code to generate, are no longer true. The performance impact of optimizer assumptions about the cache not being met will be processor and optimizer specific. cache

```
1   void WG14_N866_D(int n, int * restrict x, int * restrict y)
2   {
3   /*
4    * Let the number of elements in the pointed-to object be 3n.  The following
5    * modifies the lower-n elements through x, and modifies the upper-n
6    * elements through y.  The middle n elements are read through x and y.
7    */
```

```
8   for(int i=0; i<n; i++)
9      {
10     x[i]     += x[i+n];
11     y[i+2*n] += y[i+n];
12     }
13  }
14
15  void f(void)
16  {
17  int z[300];
18
19  WG14_N866_D(100, z, z);
20  }
```

In the following example the individual array elements pointed to by each of the restricted pointers are disjoint.

```
1   void WG14_N886_K(int n, char * restrict p, char * restrict q)
2   {
3   for(int i=0; i<n; i+=2)
4      {
5      ++p[i];
6      ++q[i];
7      }
8   }
9
10  void all_edge_affects(void)
11  {
12  int r[100];
13
14  WG14_N886_K(100, r, r+1);
15  }
```

---

restrict intended use

The intended use of the **restrict** qualifier (like the **register** storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior). 1491

Commentary

Rationale

The **restrict** type qualifier allows programs to be written so that translators can produce significantly faster executables. Anyone for whom this is not a concern can safely ignore this feature of the language.

alias analysis

A proof by Landi[10] brought the search for an optimal alias detection algorithm (determining whether two different identifiers refer to the same storage location at a particular point in a program) to an abrupt end (at least for programs containing **if** statements, loops, dynamic storage allocation, and manipulating at least a singly linked list). He showed that in general: (1) determining whether an alias occurs during some execution of a program is undecidable, and (2) determining whether an alias occurs during all executions of a program is uncomputable (simplifying the problem to (1) intraprocedural analysis where no use is made of dynamic storage allocation is NP-hard,[11] and (2) flow-insensitive, intraprocedural, may-alias analysis is NP-hard[8]).

The **restrict** qualifier promotes optimization by reducing some costs (to vendors in producing the optimizer and the developer in terms of resources needed to run the translator), at the expense of increasing another cost (developer time needed to deduce where the qualifier can be added).

restrict 1490 requires all accesses

It is possible that deleting all instances of the **restrict** qualifier will change the observable behavior of a program (because the requirement on accesses was not being met).

Other Languages

The implementations of some languages support pragmas that enable developers to communicate optimization information to the translator.

Common Implementations

Whenever a value is stored indirectly through a pointer, a translator has to assume that potentially all objects now hold a different values. A little analysis allows the set of potentially out-of-date object values to be reduced to those whose address is ever assigned to a pointer. Successively more sophisticated analyses can be used to reduce the size of this set. A store through a restricted pointer can be assumed to not affect the value of any other value accessed in the lifetime of that pointer object. This information helps other optimization techniques to do a better job, by making it possible for them to hold on longer to the information they have built up about the values and properties of objects.

The use of restricted pointers (rather than pointers without this qualification) is not itself an optimization technique. Use of a restricted pointer provides alias information that, for non-restricted pointer accesses, would otherwise have to be obtained through complete program analysis.

A number of algorithms for deducing the set of objects that a pointer could be pointing to, at any point in a program, have been proposed.[2, 12, 13] Unfortunately, for all but the smallest programs, they require large amounts of memory and processor time. An empirical evaluation[7] of the different algorithms used programs of under 7.1 KLOC for its comparisons (this figure is an over estimate since it is based on a line count of the entire source, not just the executable statements), with a single program of 29.6 KLOC exceeding available memory (200 M byte) for some of the measurements. Researchers have started to make use of the common cases seen in production code to design alternative algorithms that require fewer resources. Making use of such information enabled an analysis of Microsoft Word 97,[3] 1.4 MLOC of C++, to produce results close to those obtained by Andersen's algorithm.[2]

Hind[6] discusses pointer analysis issues that still remain open.

Coding Guidelines

The **`restrict`** qualifier also has uses outside of optimization. The use of this qualifier could be treated as an assertion that could to be checked by static analysis tools. The C99 Standard is still too new to know if this kind of usage will occur.

Example

processor
SIMD

The SIMD approach uses a fine grained model of executing portions of a program in parallel. Being able to automatically break a program into components that execute on different processors (i.e., executing different functions on different processors) has proved to be extremely difficult. However, at the time of this writing there are no commercial translators offering such functionality. Automatically deducing that calls to `walk_tree`, in the following example, can be distributed to different processors remains a very difficult problem.

```
1  #include <stddef.h>
2
3  struct data {
4               long important_value;
5             };
6  struct t_rec {
7                struct data information;
8                struct t_rec * restrict left,
9                             * restrict right;
10              };
11 typedef struct t_ref * restrict TREE;
12
13 extern void process_data(struct data);
14
15 void walk_tree(TREE root)
16 {
```

```
17  if (root->left != NULL)
18     walk_tree(root->left);
19  if (root->right != NULL)
20     walk_tree(root->right);
21
22  process_data(root->information);
23  }
```

---

qualified array type

If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. 1492

Commentary

The implications of this specification become apparent when arrays are converted to pointers to their first element. The declaration `const int arr[10];` declares `arr` to have type array of **`const int`**. If an occurrence of this object is implicitly converted to a pointer type, the resulting type is pointer to **`const int`**. If the qualifier had been associated with the array type, the converted pointer type would have been **`const`** pointer to **`int`**. The following example requires a more complicated chain of reasoning (based on a discussion by McDonald):

```
1  struct T {
2            int *mem_1[2];
3            int **mem_2;
4           };
5
6  const struct T s;  /* Members inherit const. */
7
8  void f(void)
9  {
10 TYPE_1 loc_1;
11 TYPE_2 loc_2;
12
13 loc_1 = &s.mem_1; /* Statement 1. */
14 loc_2 =  s.mem_1; /* Statement 2. */
15 }
```

struct qualified result qualified

What should the types `TYPE_1` and `TYPE_2` look like? The result of the dot selection operator includes any qualifiers in the type of its left operand. The two types needed are:

array converted to pointer

1. an expression having array type is not converted to *pointer to . . .* when it is the operand of the unary **&** operator. Thus, `&s.mem_1` has type pointer to array of const pointer to int, and the declaration of `loc_1` needs to be `int * const (*loc_1)[2];`.

array converted to pointer

2. in the second case `s.mem_1` is implicitly converted to a pointer type. However, does the conversion to pointer type occur before the above C rule on qualifiers is applied? The two possibilities are:

```
1  int *  const *  /* Type if qualified before conversion. */
2  int ** const    /* Type if qualified after conversion.  */
```

EXAMPLE 1497 const aggregate

Applying the qualifier before conversion is considered to be the preferred interpretation (it is also the behavior of the implementations tested by your author; also see elsewhere) because it prevents the elements of the array, `mem_1`, being modified (whereas the member `mem_2` is treated as having type `int ** const`).

qualified array of

A method of specifying qualifiers, in an array declaration, so they are associated with the array type was introduced in C99 and is discussed elsewhere.

Other Languages

Few, in any, languages implicitly convert arrays into pointers to their first element. So the distinction that occurs in C, because of this implicit conversion, does not arise.

Coding Guidelines

Because of the implicit conversion that occurs for parameters declared to have an array type, it is possible for them to be assigned to (because qualifiers applies to the element type). Assigning to an object defined, in the source, using an array type might be considered suspicious with or without a qualifier. However, the usage is rare and these coding guidelines are silent on the issue.

array converted to pointer

Example

```
1  void f(const int arr_1[10], const int arr_2[20])
2  {
3  arr_1 = arr_2;
4  }
```

1493 If the specification of a function type includes any type qualifiers, the behavior is undefined.[116)]

Commentary

A function type describes a function with specified return type. It would be meaningless to say that a function returned a volatile-qualified type. It only gets to return a single value. A function type having a const-qualified return type could be interpreted to mean that the value returned was always the same. The C committee choose not to give this interpretation to such a construct.

function type

C++

In fact, if at any time in the determination of a type a cv-qualified function type is formed, the program is ill-formed. 8.3.5p4

A C++ translator will issue a diagnostic for the appearance of a qualifier in a function type, while a C translator may silently ignore it.

The C++ Standard allows *cv-qualifiers* to appear in a function declarator. The syntax is:

D1 **(** *parameter-declaration-clause* **)** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$ 8.3.5p1

the *cv-qualifier* occurs after what C would consider to be the function type.

Common Implementations

The most commonly seen behavior is to ignore the type qualifiers.

Coding Guidelines

Function types whose return type includes a qualifier can occur through the use of typedef names. It would be consistent for a functions return type to use the same typedef name as the objects appearing in its **return** statement. While it might be unusual for a const-qualified object to appear in a **return** statement, the usage appears harmless and is not common.

Example

A footnote in the standard also provides some examples.

footnote 137

```
1  typedef const int C_I;
2
3  C_I f(int p_1)
4  {
5  return p_1 + 1;
6  }
```

---

qualified type to be compatible

For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; 1494

Commentary

simple assignment constraints

The requirements on the types of the operands of the assignment operators ignore any qualifiers on the type of the right operand.

C++

compatible type if

The C++ Standard does not define the term *compatible type*. However, the C++ Standard does define the terms *layout-compatible* (3.9p11) and *reference-compatible* (8.5.3p4). However, *cv-qualifiers* are not included in the definition of these terms.

---

the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type. 1495

Commentary

type qualifier 1476 syntax

Measurements of qualifier ordering are discussed elsewhere.

C++

The C++ Standard does not specify any ordering dependency on *cv-qualifiers* within a *decl-specifier*.

Coding Guidelines

declaration ?? specifier ordering

The guideline recommendation dealing with the ordering of keywords within type specifiers is discussed elsewhere.

---

const volatile EXAMPLE

EXAMPLE 1 An object declared 1496

```
extern const volatile int real_time_clock;
```

may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

Commentary

Such an object may be in a memory mapped region of storage that is effectively read-only, from the programs perspective (in the sense that any writes to objects in that area of storage do not modify the values held there).

EXAMPLE const aggregate

EXAMPLE 2 The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type: 1497

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  // the object ncs is modifiable
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;

ncs = cs;      // valid
cs = ncs;      // violates modifiable lvalue constraint for =
pi = &ncs.mem; // valid
pi = &cs.mem;  // violates type constraints for =
pci = &cs.mem; // valid
pi = a[0];     // invalid: a[0] has type "const int *"
```

Commentary

The terms *valid* and *invalid* are commonly used by developers. However, the C Standard does not define them. As used in this example, the term *valid* might be interpreted as "does not affect the conformance status of the program". And the term *invalid* might be interpreted as "will cause a translator to issue a diagnostic message".

Coding Guidelines

Some developers might be surprised that in the following declarations:

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  // the object ncs is modifiable
```

the **const** qualifier in the first declaration only applies to the type of `cs` (i.e., the tag `s` is not defined to refer to a const-qualified type). Type qualifiers are not part of any of the syntactic constructs used to define tags.

struct tag; struct-or-union identifier not visible

1498 114) A **volatile** declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function.

footnote 114

Commentary

These are probably the two most common uses of the **volatile** qualifier.

C++

The C++ Standard does not make this observation.

1499 Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

Commentary

This is a requirement on the implementation. Accessing an object declared with a volatile-qualified type is a side effect. The result of this side effect is likely to be unknown to the implementation, which is likely to play safe.

side effect

C++

*[Note: **volatile** is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of **volatile** are intended to be the same in C++ as they are in C.]*

7.1.5.1p8

Example

In the function `f` below, a translator may optimize the evaluation of the expression appearing in the first return statement. The value 0 can be unconditionally returned after a read access to `glob_2` (no multiplication operation is performed). The expression in the second **return** statement may exhibit undefined behavior. It depends on whether accessing `glob_2` causes its value to be modified. An implementation could choose to unconditionally return the value 0 after two read accesses to `glob_2` (no subtraction operation is performed). The undefined behavior, if an access does modify `glob_2`, being that the new values do not affect the value returned.

object modified once between sequence points

```
extern int glob_1;
extern volatile int glob_2;

int f(void)
{
if (glob_1 == 1)
   return glob_2 * 0;        /* First return.  */
```

```
8   else
9      return (glob_2 - glob_2); /* Second return. */
10  }
```

---

footnote 115

115) For example, a statement that assigns a value returned by **malloc** to a single pointer establishes this association between the allocated object and the pointer. 1500

Commentary

lvalue converted to value

Taking the address of an object with static or automatic storage duration is not an access to that object. Such an address may also be assigned to a restricted pointer to establish this association.

---

footnote 116

116) Both of these can occur through the use of **typedef**s. 1501

Commentary

```
1   typedef char A[10];
2
3   volatile A v_a;
4
5   typedef int F(void);
6
7   const F *cp_f;
```

C++

The C++ Standard does not make this observation, but it does include the following example:

8.3.5p4 *[Example:*

```
typedef void F();
struct S {
const F f;     // ill-formed:
               // not equivalent to: void f() const;
};
```

—end example]

References

1. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *SIGPLAN Conference on Programming Language Design and Implementation PLDI'03*, pages 129–140, June 2003.
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
3. M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, volume 35.5 of *ACM SIGPLAN Notices*, pages 35–46, N.Y., June 18–21 2000. ACM Press.
4. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.
5. J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 192–203, 1999.
6. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 54–61. ACM, June 2001.
7. M. Hind and A. Pioli. Which pointer analysis should I use? *International Symposium on Software Testing and Analysis (ISSTA 2000), Aug 2000*, 2000.
8. S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, Jan. 1997.
9. D. Koes, M. Budiu, G. Venkataramani, and S. C. Goldstein. Programmer specified pointer independence. Technical Report CMU-CS-03-128, Carnegie Mellon University, Apr. 2003.
10. W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.
11. W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the* 18^{th} *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, 1991.
12. J. Lu. *Interprocedural Pointer Analysis for C*. PhD thesis, Rice University, Apr. 1998.
13. X.-X. S. Zhang. *Practical pointer aliasing analysis*. PhD thesis, New Brunswick Rutgers, The State University of New Jersey, Oct. 1998.