

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

3.1

access

access

(execution-time action) to read or modify the value of an object

Commentary

While the behavior for most kinds of access are simple and easy to deduce, this innocent looking definition hides several dark corners. For instance, does an expression that multiplies an object by 1 modify that object? Yes. Does accessing a bit-field that shares a storage unit with another bit-field also cause an access to that other bit-field? WG14 are looking into creating a more rigorous specification of what it means to access an object. A Technical Report may be published in due course (it will take at least two years). At the time of this writing WG14 has decided to wait until the various dark corners have been more fully investigated and the issues resolved and documented before making a decision on the form of publication.

modify³⁷
includes cases

The term *reference* is also applied to *objects*. The distinction between the two terms is that programs reference objects but access their values. An unevaluated expression may contain references to objects, but it never accesses them. The term *designate an object* is used in the standard. This term can involve a reference, an access, or both, or neither. In the following:

reference
object

```
1 int *p = 0;
2 (**p);
```

the lvalue `(**p)` both accesses and references an object, namely the pointer object `p`, but it does not designate any object. The term *designate* also focuses attention on a particular object under discussion. For instance, in the expression `a[n]` there are the references `a`, `n`, and `a[n]`; the expression may or may not access any or all of `a`, `n`, and `a[n]`. But the discussion is likely to refer to the object designated by `a[n]`, not its component parts.

&*

C++

In C++ the term *access* is used primarily in the sense of *accessibility*; that is, the semantic rules dealing with when identifiers declared in different classes and namespaces can be referred to. The C++ Standard has a complete clause (Clause 11, Member access control) dealing with this issue. While the C++ Standard also uses *access* in the C sense (e.g., in 1.8p1), this is not the primary usage.

Common Implementations

Many of these corner cases involve potential optimizations that translator vendors might like to perform. Some translators containing nontrivial optimizers provide options, selectable by developers, that control the degree to which optimizations are attempted. Optimizations that may change the behavior of a construct (from one choice of undefined, or unspecified behavior to another one) are usually only enabled at the higher, try harder levels of optimization.

Example

In the following:

```
1 int f(int param)
2 {
3     struct {
4         volatile unsigned int mem1:2;
5         unsigned int mem2:4;
6         volatile unsigned int mem3:6;
7         unsigned int mem4:1;
8     } mmap;
9
10    return mmap.mem2 += param;
11 }
```

the member mem2 may be placed in the same storage unit as mem1 and/or mem3. If the processor does not have bit-field extraction instructions, it will be necessary to generate code to load one or more bytes to obtain the value of mem2. If the member mem1 is contained in one of those bytes, does an access of mem2 also constitute an access to mem1?

Another issue is the number of accesses to a bit-field. Obtaining the value of mem3 requires a single, read, access. But how many accesses are needed to modify mem3? One to read the value, which is modified; a store back into mem3 may require another read (to obtain the value of the other members stored in the same storage unit; perhaps mem4 in the preceding example); the modified value of mem3 is inserted into the bit pattern read and the combined value written back into the storage unit. An alternative implementation may hang on to the value of mem4 so that the second read access is not needed.

36 NOTE 1 Where only one of these two actions is meant, “read” or “modify” is used.

37 NOTE 2 “Modify” includes the case where the new value being stored is the same as the previous value.

modify
includes cases

Commentary

This specification only needs to be followed exactly when there is more than one access between sequence points, or for objects declared with the volatile storage class. As far as the developer is concerned, a modification occurs. As long as the implementation delivers the expected result, it is free to do whatever it likes.

Example

```

1  extern int glob;
2  extern volatile int reg;
3
4  void g(void)
5  {
6  reg *= 1; /*
7             * Value of reg looks as if it is unchanged, but because
8             * it is volatile qualified an access to the object is
9             * required. This access may cause its value to change.
10            */
11 /*
12  * The following cannot be optimized to a single assignment.
13  */
14 reg=glob;
15 reg=glob;
16
17 glob += 0; /* Value of glob unchanged but it is still modified. */
18 /*
19  * glob modified twice between sequence points -> undefined behavior
20  */
21 glob = (glob += 0) + 4;
22 }
```

38 NOTE 3 Expressions that are not evaluated do not access objects.

Commentary

The term *not evaluated* here means *not required to be evaluated by the standard*. Implementations do not have complete freedom to decide what not to evaluate. Reasons why an expression may not be evaluated include:

- being part of a statement which is not executed,

- being part of a subexpression whose evaluation is conditional on other subexpressions within a full expression, and
- being an operand of the **sizeof** operator.

If an implementation can deduce that the evaluation of an expression causes no side-effects, it can use the as-if rule to optimize away the generation of machine code (to evaluate the expression).

Common Implementations

The inverse of this rule is one area where translators that perform optimizations have to be careful. In:

```
1  a = b + c;
```

`b` and `c` are accessed to obtain their values for the `+` operator. An optimizer might, for instance, deduce that their sum had already been calculated and is still available in a register. However, using the value held in that register is only possible if it can be shown that not accessing `b` and `c` will not change the behavior of the program. If either were declared with the `volatile` qualifier, for instance, such an optimization could not be performed.

Coding Guidelines

Accessing an object only becomes important if there are side effects associated with that access; for instance, if the object is declared with the **volatile** qualifier. This issue is covered by guideline recommendations discussed elsewhere.

Example

```
1  extern int i, j;
2  extern volatile int k;
3
4  void f(void)
5  {
6  /*
7   * Side effect of access to k occurs if the left operand
8   * of the && operator evaluates to nonzero.
9   */
10 if ((i == 2) && (j == k))
11     /* ... */ ;
12 /*
13  * In the following expression k appears to be read twice.
14  * Only one of the subexpressions will ever be evaluated.
15  * There is no unspecified or undefined behavior.
16  */
17 i = (j < 3 ? (k - j) : (j - k));
18 }
```

sizeof
operand evaluated
sizeof
operand not
evaluated

as-if rule

sequence ??
points
all orderings
give same value
code ??
shall affect output

References