The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

3.4.4

unspecified behavior

unspecified behavior

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

Commentary

implementation-defined behavior correct program

The difference between unspecified (implementation-defined) and undefined is that the former applies to a correct program construct and correct data while the latter applies to an erroneous program construct or erroneous data. Are there any restrictions on possible translator behaviors? The following expresses the Committee's intent:

Rationale

This latitude does not extend as far as failing to translate the program.

The wording was changed by the response to DR #247.

C90

Behavior, for a correct program construct and correct data, for which this International Standard imposes no requirements.

The C99 wording more clearly describes the intended meaning of the term *unspecified behavior*, given the contexts in which it is used.

C++

1.3.13

behavior, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behavior occurs. [Note: usually, the range of possible behaviors is delineated by this International Standard.]

This specification suggests that there exist possible unspecified behaviors that are not delineated by the standard, while the wording in the C Standard suggests that all possible unspecified behaviors are mentioned.

Common Implementations

Because there is no requirement to document the behavior in these cases, and the frequent difficulty of being able to say anything specific, most vendors say nothing.

Coding Guidelines

For implementation-defined behaviors, it is possible to read the documented behavior. For unspecified behaviors vendors do not usually document possible behaviors.

All nontrivial programs contain unspecified behavior; for instance, in the expression `b+c`, it is unspecified whether the object `b` is accessed before or after the object `c`. But, unless both objects are defined using the **`volatile`** qualifier, the order has no affect on a program's output.

type qualifier syntax

A blanket guideline recommendation prohibiting the use of any construct whose behavior is unspecified would be counterproductive. If the behavior of a construct is unspecified, but the behavior of the program containing it is identical for all of the different possibilities, the usage should be regarded as acceptable. How possible, different unspecified behaviors might be enumerated and the effects these behaviors have on the output of a program is left to developers to deduce. This issue is also covered by a guideline recommendation discussed elsewhere.

sequence ?? points all orderings give same value

Example

In an optimizing compiler the order of evaluation of expressions is likely to depend on context: What registers are free to store intermediate values; What registers contain previously calculated values that can be reused in the current expression? Documenting the behavior would entail describing all of the flow analysis, expression tree structure, and code optimization algorithms. In theory the order of evaluation, for each expression, could be deduced from this information. However, it would be impractical to carry out.

```
1  #include <stdio.h>
2
3  static int glob_1, glob_2;
4
5  int main(void)
6  {
7  if ((glob_1++, (glob_1+glob_2)) == (glob_2++, (glob_1+glob_2)))
8     printf("We may print this\n");
9  else
10    printf("or we may print this\n");
11 }
```

Usage

Annex J.1 lists 50 unspecified behaviors.

---

50 EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

References