ISO/IEC JTC 1/SC 22/OWGV N0051

22 November 2006

Contribution from:
Derek Jones, Some proposed language vulnerability guidelines, 20 November 2006.

Some proposed language vulnerability guidelines

Submitted to the December 2006 Washington D.C. meeting of the OWG

Derek M. Jones
derek@knosof.co.uk

1 Introduction

This paper is an attempt to list all of the issues that need to be covered by a set of general language usage coding guidelines.

Issues have been placed in the following categories:

- Language definition
- Implementation behavior
- Human factors

2 Language definition

The definition of most languages does not exactly define the behavior of every language construct under every circumstance. The different kinds of variations in behavior are often grouped into a small number of categories which are called out in the language definition.

An obvious way of addressing these constructs whose variation in behavior might be a source of faults in programs is to recommend against their use. However, these constructs often provide useful functionality and in some cases it is possible to restrict usage so that the behavior is always fully defined.

Some languages are defined by a particular implementation, e.g., Perl and PHP. In these cases it is still possible for unspecified, implementation defined and undefined behaviors to occur. Every port of the implementation to a different platform is a different implementation.

Do we recommend that languages defined by an implementation should not be used, or do we require that all of the behaviors that would normally appear in a language definition document be documented?

2.1 Unspecified behavior

Unspecified behavior occurs when the language definition specifies more than one possible behavior for a construct. For instance, in

```
A = B ;
```

the C Standard allows an implementation to evaluate `B` followed by the evaluation of `A`, or to evaluate `A` followed by the evaluation of `B`.

The important point with unspecified behavior is not that more than one sequence of operations is possible, but whether the different sequences always produce the same result or whether the result is different for some sequences.

Recommending against the use of assignment statements would solve the problem of the unspecified behavior that occurs in the evaluation of their operands. However, assignment statements are very useful and it is worth making sure that their use always produces the same answer.

In the following example:

```
i = printf("Hello") + printf(" world\n");
```

the two calls to `printf` will occur in one of two possible orders, each producing a different output. The code can be rewritten to ensure that the output is always the same, as in:

```
i = printf("Hello world\n");
```

Cg .1 Use of a construct having unspecified behavior shall produce a result is the same for all of the possible behaviors defined by the language definition.

A C binding to this guideline might expand it to include the following guidelines:

- The result of evaluating an expression, shall not depend on the order of evaluation of subexpressions or the order in which side effects take place.
- The output of a program shall not depend on the order in which the arguments to function calls are evaluated.

2.2 Implementation defined behavior

Implementation defined behavior is behavior that can vary between implementations. The term implementation need not apply to products from different companies, it can also apply to different versions of the same compiler.

The authors of a language definition can have a number of reasons for wanting to explicitly call out that the behavior of a construct may vary between implementations, including the following:

- when specifying the behavior of an implementation that may in practice have no effect. For instance, in C use of the **`register`** storage class has implementation defined behavior (it is a hint to the compiler that an object is frequently used and the code might execute faster if the object's value were kept in a machine register). In practice most processors hold the value of objects in registers for a period of time, irrespective of whether the **`register`** storage class is used.
- when specifying the behavior for which implementations are known to vary and no agreement can be reached on a single definition (e.g., whether the type char could be treated as a signed or unsigned type).

Use of implementation defined constructs may provide many benefits and programs rarely have to be portable to all possible implementations. If the behavior across implementations used to build a program is fixed, or known to vary within prescribed limits, use of an implementation defined construct may have a cost that is significantly lower than the benefit.

A software developer may chose to specify that a particular implementation will always be used, or that an implementation having specific characteristics will always be used.

Like constructs having unspecified behavior the important issue occurs when changes in behavior change the output.

Cg .2 Use of a construct having implementation defined behavior shall produce a result is the same for all of the possible behaviors likely to occur in the implementations used.

2.3 Undefined behavior

A particular use of a construct is often specified as having undefined behavior because it is possible to occur and the authors of the language definition did not want to specify a behavior. A common motivation is that the usage represents behavior that is considered erroneous. An example of undefined behavior in C is accessing an object whose value is indeterminate (i.e., an uninitialised local variable).

Given the lack of any specification of behavior use of any construct that causes undefined behavior is likely to lead to unexpected program output, and is to be avoided.

Cg .3 Any use of a construct having undefined behavior shall not occur in a program.

Accessing storage outside of an object

Indexing an array out of bounds...

Copying past the end of an object (buffer overflow).

3 Implementation behavior

Ideally the behavior of an implementation is completely encompassed by the appropriate language definition. In practice there are implementations defined behaviors that are not specified in a language definition. For instance few languages define the representation that should be used for integer types (the C Standard specifies that one of three possibilities be used).

The behavior of implementations will be a consideration when considering the extent to which implementation defined behavior is

3.1 Limits

All implementations have limits of one sort or another. Some of these might include:

- the maximum size of an object that can be allocated storage,
- the maximum nesting of constructs (e.g., nesting of parenthesis in an expression, or nesting of compound statements),
- the maximum number of identifiers that can be declared.

Some language definitions, for instance C, specify minimum limits on the number of constructs that conforming implementations are required to support.

Exceeding an implementation limit may not be a source of faults in that the result is to fail to build an executable program.

The extent to which commonly used implementations have known limits may be something that writers of language specific guidelines may want to address.

3.2 Representation information

Making use of knowledge of the pattern of bits used to represent some values has its costs and benefits. The cost is incurred when faults are introduced because the pattern changes (which changes the behavior of the program) or when a developer is unaware of the representation related operations and their implications.

Cg .4 Algorithms or operations shall not make use of knowledge of representation details.

Examples of making use of representation information include using bitwise operations to perform arithmetic operations, shifting right to perform a divide,...

4 Human factors

The impact of human factors on faults in software can be divided into two broad categories: knowledge failures and performance failures.

4.1 Knowledge failures

Knowledge failures occur when a developer believes they know the result produced by a construct, but that belief is incorrect. Recommendations against knowledge failures requires information on those constructs about which developers often have incorrect beliefs and an estimation of the knowledge a developer is expected to have.

There are two possible ways of dealing with incorrect beliefs. Use of the construct can be recommended against, or an alternative usage can be recommended that mitigates the incorrect belief.

An example of a knowledge failure is developer beliefs on the relative precedence of binary operators. There are a number of possible ways of mitigating this situation, including:

- recommending that parenthesis always be used for those combinations that developers frequently get wrong,
- recommending that expressions be split such that certain pairs of binary operators are not present in expressions.

Implicit conversions of a value from one type to another are an example of a construct whose use may create an opportunity for both a knowledge and a performance failure. Developers may be unaware of the conversions specified in the language definition or may simply fail to apply the knowledge they have.

Analysis of the accuracy and stability of algorithms that contain floating point calculations requires a great deal of expertise and most developers are unaware of what needs to be done.

Cg .5 No assumptions about the accuracy of the result of any expression that contains operands having a floating-point type shall be made unless they are backed by a mathematical analysis.

4.2 Performance failures

A performance failure occurs when a developer has correct knowledge but applies it incorrectly. For instance, in the controlling expression of a C *if-statement* the operator = might be mistaken for the operator ==.

Like knowledge failures there are two possible ways of dealing with performance failures. Use of the construct can be recommended against, or an alternative usage can be recommended that mitigates the failure.

The =/== case occurs because of an assumption about the common case and a failure to play sufficient attention to what appears in the source. Another such example of this kind of failure occurs when the character l is treated as the digit 1 rather the letter l (or vice versa).

Other performance failures may be non-local. For instance, when deleting the declaration a local variable that has the same name as a variable at file scope it is also necessary to delete all references to the local variable in the body of the function, otherwise any reference will go unnoticed and refer to the variable at file scope.

Some languages contain constructs which *introduce* variables into a scope without any obvious declaration being visible. For instance, argument dependent lookup and multiple inheritance in C++.

Cg .6 A declaration of an identifier shall not occur in a scope where an identifier with the same name is accessible.

An alternative guideline is:

Cg .7 Adding/deleting a declaration shall not cause a program to continue to successfully compile.

Performance mistakes can also occur when a great deal of cognitive effort is needed to analyse the behavior of a construct. The term *complex* or *complicated* is sometimes used to describe such constructs. However, there is no generally accepted, and experimentally verified, definition of what constitutes complexity.

Signals, exceptions and interrupts introduce a degree of non-determinacy into the behavior of a program.