EXP30-C. Do not depend on the order of evaluation for side effects

Evaluation of an expression may produce side effects. At specific points during execution, known as sequence points, all side effects of previous evaluations are complete, and no side effects of subsequent evaluations have yet taken place. Do not depend on the order of evaluation for side effects unless there is an intervening sequence point.

The C Standard, 6.5, paragraph 2 [ISO/IEC 9899:2011], states

> *If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.*

This requirement must be met for each allowable ordering of the subexpressions of a full expression; otherwise, the behavior is undefined. (See undefined behavior 35.)

The following sequence points are defined in the C Standard, Annex C [ISO/IEC 9899:2011]:

- Between the evaluations of the function designator and actual arguments in a function call and the actual call
- Between the evaluations of the first and second operands of the following operators:
    - Logical AND: &&
    - Logical OR: `||`
    - Comma: `,`
- Between the evaluations of the first operand of the conditional `?:` operator and whichever of the second and third operands is evaluated
- The end of a full declarator
- Between the evaluation of a full expression and the next full expression to be evaluated; the following are full expressions:
    - An initializer that is not part of a compound literal
    - The expression in an expression statement
    - The controlling expression of a selection statement (`if` or `switch`)
    - The controlling expression of a `while` or do statement
    - Each of the (optional) expressions of a `for` statement
    - The (optional) expression in a `return` statement
- Immediately before a library function returns
- After the actions associated with each formatted input/output function conversion specifier
- Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call

This rule means that statements such as

```
i = i + 1;
a[i] = i;
```

have defined behavior, and statements such as the following do not:

```
/* i is modified twice between sequence points */
i = ++i + 1;

/* i is read other than to determine the value to be stored */
a[i++] = i;
```

Not all instances of a comma in C code denote a usage of the comma operator. For example, the comma between arguments in a function call is not a sequence point. However, according to the C Standard, 6.5.2.2, paragraph 10 [ISO/IEC 9899:2011]

> *Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.*

This rule means that the order of evaluation for function call arguments is unspecified and can happen in any order.

Noncompliant Code Example

Programs cannot safely rely on the order of evaluation of operands between sequence points. In this noncompliant code example, `i` is evaluated twice without an intervening sequence point, so the behavior of the expression is undefined:

```
#include <stdio.h>

void func(int i, int *b) {
  int a = i + b[++i];
  printf("%d, %d", a, i);
}
```

Compliant Solution

These examples are independent of the order of evaluation of the operands and can be interpreted in only one way:

```
#include <stdio.h>

void func(int i, int *b) {
  int a;
  ++i;
  a = i + b[i];
  printf("%d, %d", a, i);
}
```

Alternatively:

```
#include <stdio.h>

void func(int i, int *b) {
  int a = i + b[i + 1];
  ++i;
  printf("%d, %d", a, i);
}
```

Noncompliant Code Example

The call to `func()` in this noncompliant code example has undefined behavior because there is no sequence point between the argument expressions:

```
extern void func(int i, int j);

void f(int i) {
  func(i++, i);
}
```

The first (left) argument expression reads the value of `i` (to determine the value to be stored) and then modifies `i`. The second (right) argument expression reads the value of `i` between the same pair of sequence points as the first argument, but not to determine the value to be stored in `i`. This additional attempt to read the value of `i` has undefined behavior.

Compliant Solution

This compliant solution is appropriate when the programmer intends for both arguments to `func()` to be equivalent:

```
extern void func(int i, int j);

void f(int i) {
  i++;
  func(i, i);
}
```

This compliant solution is appropriate when the programmer intends for the second argument to be 1 greater than the first:

```
extern void func(int i, int j);

void f(int i) {
  int j = i++;
  func(j, i);
}
```

Noncompliant Code Example

The order of evaluation for function arguments is unspecified. This noncompliant code example exhibits unspecified behavior but not undefined behavior:

```
extern void c(int i, int j);
int glob;

int a(void) {
  return glob + 10;
}

int b(void) {
  glob = 42;
  return glob;
}

void func(void) {
  c(a(), b());
}
```

It is unspecified what order `a()` and `b()` are called in; the only guarantee is that both `a()` and `b()` will be called before `c()` is called. If `a()` or `b()` rely on shared state when calculating their return value, as they do in this example, the resulting arguments passed to `c()` may differ between compilers or architectures.

Compliant Solution

In this compliant solution, the order of evaluation for `a()` and `b()` is fixed, and so no unspecified behavior occurs:

```
extern void c(int i, int j);
int glob;

int a(void) {
  return glob + 10;
}
int b(void) {
  glob = 42;
  return glob;
}

void func(void) {
  int a_val, b_val;

  a_val = a();
  b_val = b();

  c(a_val, b_val);
}
```

Risk Assessment

Attempting to modify an object multiple times between sequence points may cause that object to take on an unexpected value, which can lead to unexpected program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP30-C	Medium	Probable	Medium	**P8**	**L2**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**evaluation-order** **multiple-volatile-accesses**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-EXP30**	
Clang	3.9	`-Wunsequenced`	Detects simple violations of this rule, but does not diagnose unsequenced function call arguments.
Compass/ROSE			Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator
Coverity	2017.07	**EVALUATION_ORDER**	Can detect the specific instance where a statement contains multiple side effects on the same value with an undefined evaluation order because, with different compiler flags or different compilers or platforms, the statement may behave differently
ECLAIR	1.2	**CC2.EXP30**	Fully implemented
GCC	4.3.5		Can detect violations of this rule when the `-Wsequence-point` flag is used
Klocwork	2018	**PORTING.VAR.EFFECTS**	
LDRA tool suite	9.7.1	**35 D, 1 Q, 9 S, 30 S, 134 S**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-EXP30-a** **CERT_C-EXP30-b** **CERT_C-EXP30-c** **CERT_C-EXP30-d**	The value of an expression shall be the same under any order of evaluation that the standard permits Don't write code that depends on the order of evaluation of function arguments Don't write code that depends on the order of evaluation of function designator and function arguments Don't write code that depends on the order of evaluation of expression that involves a function call
Polyspace Bug Finder	R2019a	CERT C: Rule EXP30-C	Checks for situations when expression value depends on order of evaluation or of side effects (rule fully covered)
PRQA QA-C	9.5	**0400 [U], 0401 [U], 0402 [U],** **0403 [U], 0403 [U], 0403 [U],** **0404 [U], 0405 [U]**	Fully implemented
PVS-Studio	6.23	**V567**	
RuleChecker	19.04	**evaluation-order** **multiple-volatile-accesses**	Fully checked
Splint	3.1.1		
SonarQube C/C++ Plugin	3.11	**IncAndDecMixedWithOtherOperators**	
TrustInSoft Analyzer	1.38	**separated**	Exhaustively verified (see one compliant and one non-compliant example).

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT C	EXP50-CPP. Do not depend on the order of evaluation for side effects	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT Oracle Secure Coding Standard for Java	EXP05-J. Do not follow a write by a subsequent write or read of the same object within an expression	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Side-effects and Order of Evaluation [SAM]	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 13.2 (required)	CERT cross-reference in MISRA C: 2012 – Addendum 3
CWE 2.11	CWE-758	2017-07-07: CERT: Rule subset of CWE

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-758 and EXP30-C

Independent(INT34-C, INT36-C, MEM30-C, MSC37-C, FLP32-C, EXP33-C, EXP30-C, ERR34-C, ARR32-C)

CWE-758 = Union(EXP30-C, list) where list =

- Undefined behavior that results from anything other than reading and writing to a variable twice without an intervening sequence point.

Bibliography

[ISO/IEC 9899:2011]	6.5, "Expressions" 6.5.2.2, "Function Calls" Annex C, "Sequence Points"
[Saks 2007]	
[Summit 2005]	Questions 3.1, 3.2, 3.3, 3.3b, 3.7, 3.8, 3.9, 3.10a, 3.10b, and 3.11