DCL36-C. Do not declare an identifier with conflicting linkage classifications

Linkage can make an identifier declared in different scopes or declared multiple times within the same scope refer to the same object or function. Identifiers are classified as *externally linked*, *internally linked*, or *not linked*. These three kinds of linkage have the following characteristics [Kirch-Prinz 2002]:

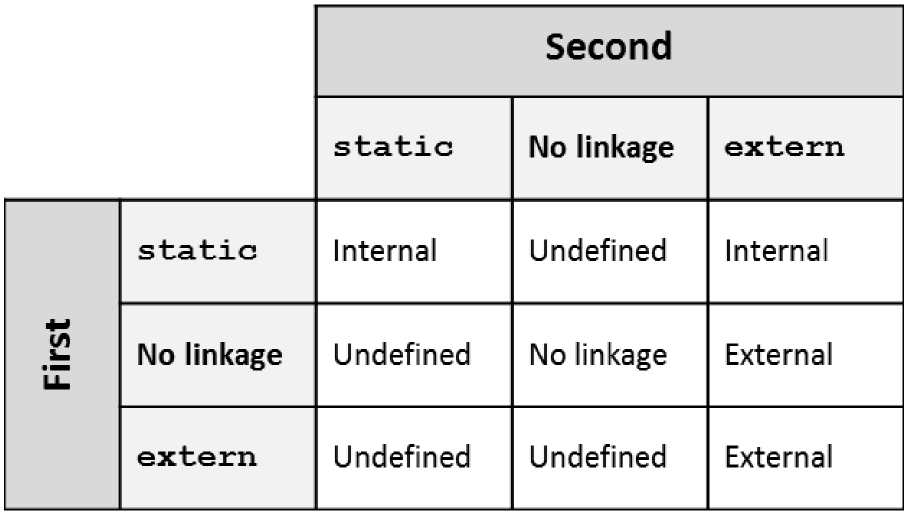
- **External linkage:** An identifier with external linkage represents the same object or function throughout the entire program, that is, in all compilation units and libraries belonging to the program. The identifier is available to the linker. When a second declaration of the same identifier with external linkage occurs, the linker associates the identifier with the same object or function.
- **Internal linkage:** An identifier with internal linkage represents the same object or function within a given translation unit. The linker has no information about identifiers with internal linkage. Consequently, these identifiers are internal to the translation unit.
- **No linkage:** If an identifier has no linkage, then any further declaration using the identifier declares something new, such as a new variable or a new type.

According to the C Standard, 6.2.2 [ISO/IEC 9899:2011], linkage is determined as follows:

> *If the declaration of a file scope identifier for an object or a function contains the storage class specifier `static`, the identifier has internal linkage.*
>
> *For an identifier declared with the storage-class specifier `extern` in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.*
>
> *If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.*
>
> *The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier `extern`.*

Use of an identifier (within one translation unit) classified as both internally and externally linked is undefined behavior. (See also undefined behavior 8.) A translation unit includes the source file together with its headers and all source files included via the preprocessing directive `#include`.

The following table identifies the linkage assigned to an object that is declared twice in a single translation unit. The column designates the first declaration, and the row designates the redeclaration.



		Second		
		`static`	No linkage	`extern`
First	`static`	Internal	Undefined	Internal
	No linkage	Undefined	No linkage	External
	`extern`	Undefined	Undefined	External

Noncompliant Code Example

In this noncompliant code example, `i2` and `i5` are defined as having both internal and external linkage. Future use of either identifier results in undefined behavior.

```
int i1 = 10;         /* Definition, external linkage */
static int i2 = 20;  /* Definition, internal linkage */
extern int i3 = 30;  /* Definition, external linkage */
int i4;              /* Tentative definition, external linkage */
static int i5;       /* Tentative definition, internal linkage */

int i1;  /* Valid tentative definition */
int i2;  /* Undefined, linkage disagreement with previous */
int i3;  /* Valid tentative definition */
int i4;  /* Valid tentative definition */
int i5;  /* Undefined, linkage disagreement with previous */

int main(void) {
  /* ... */
  return 0;
}
```

Implementation Details

Microsoft Visual Studio 2013 issues no warnings about this code, even at the highest diagnostic levels.

The GCC compiler generates a fatal diagnostic for the conflicting definitions of `i2` and `i5`.

Compliant Solution

This compliant solution does not include conflicting definitions:

```
int i1 = 10;         /* Definition, external linkage */
static int i2 = 20;  /* Definition, internal linkage */
extern int i3 = 30;  /* Definition, external linkage */
int i4;              /* Tentative definition, external linkage */
static int i5;       /* Tentative definition, internal linkage */

int main(void) {
  /* ... */
  return 0;
}
```

Risk Assessment

Use of an identifier classified as both internally and externally linked is undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL36-C	Medium	Probable	Medium	**P8**	**L2**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**static-function-declaration** **static-object-declaration**	Partially checked
Axivion Bauhaus Suite	6.9.0	**CertC-DCL36**	Fully implemented
Coverity	2017.07	**PW.LINKAGE_CONFLICT**	Implemented
ECLAIR	1.2	**CC2.DCL36**	Fully implemented
GCC	4.3.5		
Klocwork	2018	**MISRA.FUNC.STATIC.REDECL**	
LDRA tool suite	9.7.1	**461 S, 575 S, 2 X**	Fully implemented
Splint	3.1.1		

Parasoft C /C++test	10.4.2	**CERT_C-DCL36-a**	Identifiers shall not simultaneously have both internal and external linkage in the same translation unit
Polyspace Bug Finder	R2018a	MISRA C:2012 Rule 8.2 MISRA C:2012 Rule 8.4 MISRA C:2012 Rule 8.8 MISRA C:2012 Rule 17.3	Function types shall be in prototype form with named parameters A compatible declaration shall be visible when an object or function with external linkage is defined The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage A function shall not be declared implicitly
PRQA QA-C	9.5	**0625 (U)**	Fully implemented
RuleChecker	19.04	**static-function-declaration** **static-object-declaration**	Partially checked
TrustInSoft Analyzer	1.38	**non-static declaration follows static declaration**	Partially verified.

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
MISRA C:2012	Rule 8.2 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 8.4 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 8.8 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 17.3 (mandatory)	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[Banahan 2003]	Section 8.2, "Declarations, Definitions and Accessibility"
[ISO/IEC 9899:2011]	6.2.2, "Linkages of Identifiers"
[Kirch-Prinz 2002]	