DCL40-C. Do not create incompatible declarations of the same function or object

Two or more incompatible declarations of the same function or object must not appear in the same program because they result in undefined behavior. The C Standard, 6.2.7, mentions that two types may be distinct yet compatible and addresses precisely when two distinct types are compatible.

The C Standard identifies four situations in which undefined behavior (UB) may arise as a result of incompatible declarations of the same function or object:

UB	Description	Code
15	*Two declarations of the same object or function specify types that are not compatible (6.2.7).*	All noncompliant code in this guideline
31	*Two identifiers differ only in nonsignificant characters (6.4.2.1).*	Excessively Long Identifiers
37	*An object has its stored value accessed other than by an lvalue of an allowable type (6.5).*	Incompatible Object Declarations Incompatible Array Declarations
41	*A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).*	Incompatible Function Declarations Excessively Long Identifiers

Although the effect of two incompatible declarations simply appearing in the same program may be benign on most implementations, the effects of invoking a function through an expression whose type is incompatible with the function definition are typically catastrophic. Similarly, the effects of accessing an object using an lvalue of a type that is incompatible with the object definition may range from unintended information exposure to memory overwrite to a hardware trap.

Noncompliant Code Example (Incompatible Object Declarations)

In this noncompliant code example, the variable `i` is declared to have type `int` in file `a.c` but defined to be of type `short` in file `b.c`. The declarations are incompatible, resulting in undefined behavior 15. Furthermore, accessing the object using an lvalue of an incompatible type, as shown in function `f()`, is undefined behavior 37 with possible observable results ranging from unintended information exposure to memory overwrite to a hardware trap.

```
/* In a.c */
extern int i;   /* UB 15 */

int f(void) {
  return ++i;   /* UB 37 */
}

/* In b.c */
short i;   /* UB 15 */
```

Compliant Solution (Incompatible Object Declarations)

This compliant solution has compatible declarations of the variable `i`:

```
/* In a.c */
extern int i;

int f(void) {
  return ++i;
}

/* In b.c */
int i;
```

Noncompliant Code Example (Incompatible Array Declarations)

In this noncompliant code example, the variable a is declared to have a pointer type in file a.c but defined to have an array type in file b.c. The two declarations are incompatible, resulting in undefined behavior 15. As before, accessing the object in function f() is undefined behavior 37 with the typical effect of triggering a hardware trap.

```
/* In a.c */
extern int *a;   /* UB 15 */

int f(unsigned int i, int x) {
  int tmp = a[i];   /* UB 37: read access */
  a[i] = x;         /* UB 37: write access */
  return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 };   /* UB 15 */
```

Compliant Solution (Incompatible Array Declarations)

This compliant solution declares a as an array in a.c and b.c:

```
/* In a.c */
extern int a[];

int f(unsigned int i, int x) {
  int tmp = a[i];
  a[i] = x;
  return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 };
```

Noncompliant Code Example (Incompatible Function Declarations)

In this noncompliant code example, the function f() is declared in file a.c with one prototype but defined in file b.c with another. The two prototypes are incompatible, resulting in undefined behavior 15. Furthermore, invoking the function is undefined behavior 41 and typically has catastrophic consequences.

```
/* In a.c */
extern int f(int a);   /* UB 15 */

int g(int a) {
  return f(a);   /* UB 41 */
}

/* In b.c */
long f(long a) {   /* UB 15 */
  return a * 2;
}
```

Compliant Solution (Incompatible Function Declarations)

This compliant solution has compatible prototypes for the function f():

```
/* In a.c */
extern int f(int a);

int g(int a) {
  return f(a);
}

/* In b.c */
int f(int a) {
  return a * 2;
}
```

Noncompliant Code Example (Incompatible Variadic Function Declarations)

In this noncompliant code example, the function `buginf()` is defined to take a variable number of arguments and expects them all to be signed integers with a sentinel value of `-1`:

```
/* In a.c */
void buginf(const char *fmt, ...) {
   /* ... */
}

/* In b.c */
void buginf();
```

Although this code appears to be well defined because of the prototype-less declaration of `buginf()`, it exhibits undefined behavior in accordance with the C Standard, 6.7.6.3, paragraph 15 [ISO/IEC 9899:2011],

> *For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions.*

Compliant Solution (Incompatible Variadic Function Declarations)

In this compliant solution, the prototype for the function `buginf()` is included in the scope in the source file where it will be used:

```
/* In a.c */
void buginf(const char *fmt, ...) {
   /* ... */
}

/* In b.c */
void buginf(const char *fmt, ...);
```

Noncompliant Code Example (Excessively Long Identifiers)

In this noncompliant code example, the length of the identifier declaring the function pointer `bash_groupname_completion_function()` in the file `bashline.h` exceeds by 3 the minimum implementation limit of 31 significant initial characters in an external identifier. This introduces the possibility of colliding with the `bash_groupname_completion_funct` integer variable defined in file `b.c`, which is exactly 31 characters long. On an implementation that exactly meets this limit, this is undefined behavior 31. It results in two incompatible declarations of the same function. (See undefined behavior 15.) In addition, invoking the function leads to undefined behavior 41 with typically catastrophic effects.

```
/* In bashline.h */
/* UB 15, UB 31 */
extern char * bash_groupname_completion_function(const char *, int);

/* In a.c */
#include "bashline.h"

void f(const char *s, int i) {
  bash_groupname_completion_function(s, i);  /* UB 41 */
}

/* In b.c */
int bash_groupname_completion_funct;  /* UB 15, UB 31 */
```

NOTE: The identifier `bash_groupname_completion_function` referenced here was taken from GNU Bash, version 3.2.

Compliant Solution (Excessively Long Identifiers)

In this compliant solution, the length of the identifier declaring the function pointer `bash_groupname_completion()` in `bashline.h` is less than 32 characters. Consequently, it cannot clash with `bash_groupname_completion_funct` on any compliant platform.

```
/* In bashline.h */
extern char * bash_groupname_completion(const char *, int);

/* In a.c */
#include "bashline.h"

void f(const char *s, int i) {
  bash_groupname_completion(s, i);
}

/* In b.c */
int bash_groupname_completion_funct;
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL40-C	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**type-compatibility** **type-compatibility-link** **distinct-extern**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-DCL40**	Fully implemented
CodeSonar	5.1p0	**LANG.STRUCT.DECL.IF** **LANG.STRUCT.DECL.IO**	Inconsistent function declarations Inconsistent object declarations
Coverity	2017.07	**MISRA C 2012 Rule 8.4**	Implemented
LDRA tool suite	8.5.4	**1 X, 17 D**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-DCL40-a** **CERT_C-DCL40-b**	A declaration shall be visible when an object or function with external linkage is defined If objects or functions are declared more than once their types shall be compatible
Parasoft Insure++			Runtime analysis

Polyspace Bug Finder	R2018a	Declaration mismatch MISRA C:2012 Rule 5.1 MISRA C:2012 Rule 8.3	Mismatch between function or variable declarations External identifiers shall be distinct All declarations of an object or function shall use the same names and type qualifiers
PRQA QA-C	9.5	**0776, 0778, 0779, 0789, 1510**	Fully implemented
PRQA QA-C++	4.3	**1510**	
RuleChecker	19.04	**type-compatibility** **type-compatibility-link** **distinct-extern**	Fully checked
TrustInSoft Analyzer	1.38	**incompatible declaration**	Exhaustively verified.

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
ISO/IEC TS 17961	Declaring the same function or object in incompatible ways [funcdecl]	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 8.4 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[Hatton 1995]	Section 2.8.3
[ISO/IEC 9899:2011]	6.7.6.3, "Function Declarators (including Prototypes)" J.2, "Undefined Behavior"