DCL31-C. Declare identifiers before using them

The C11 Standard requires type specifiers and forbids implicit function declarations. The C90 Standard allows implicit typing of variables and functions. Consequently, some existing legacy code uses implicit typing. Some C compilers still support legacy code by allowing implicit typing, but it should not be used for new code. Such an implementation may choose to assume an implicit declaration and continue translation to support existing programs that used this feature.

Noncompliant Code Example (Implicit int)

C no longer allows the absence of type specifiers in a declaration. The C Standard, 6.7.2 [ISO/IEC 9899:2011], states

\[
\text{At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name.}
\]

This noncompliant code example omits the type specifier:

```c
extern foo;
```

Some C implementations do not issue a diagnostic for the violation of this constraint. These nonconforming C translators continue to treat such declarations as implying the type int.

Compliant Solution (Implicit int)

This compliant solution explicitly includes a type specifier:

```c
extern int foo;
```

Noncompliant Code Example (Implicit Function Declaration)

Implicit declaration of functions is not allowed; every function must be explicitly declared before it can be called. In C90, if a function is called without an explicit prototype, the compiler provides an implicit declaration.

The C90 Standard [ISO/IEC 9899:1990] includes this requirement:

\[
\text{If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration extern int identifier(); appeared.}
\]

If a function declaration is not visible at the point at which a call to the function is made, C90-compliant platforms assume an implicit declaration of extern int identifier().

This declaration implies that the function may take any number and type of arguments and return an int. However, to conform to the current C Standard, programmers must explicitly prototype every function before invoking it. An implementation that conforms to the C Standard may or may not perform implicit function declarations, but C does require a conforming implementation to issue a diagnostic if it encounters an undeclared function being used.

In this noncompliant code example, if malloc() is not declared, either explicitly or by including stdlib.h, a compiler that conforms only to C90 may implicitly declare malloc() as int malloc(). If the platform’s size of int is 32 bits, but the size of pointers is 64 bits, the resulting pointer would likely be truncated as a result of the implicit declaration of malloc(), returning a 32-bit integer.

```c
#include <stddef.h>
/* #include <stdlib.h> is missing */

int main(void) {
    for (size_t i = 0; i < 100; ++i) {
        /* int malloc() assumed */
        char *ptr = (char *)malloc(0x10000000);
        *ptr = 'a';
    }
    return 0;
}
```

Implementation Details
When compiled with Microsoft Visual Studio 2013 for a 64-bit platform, this noncompliant code example will eventually cause an access violation when dereferencing `ptr` in the loop.

**Compliant Solution (Implicit Function Declaration)**

This compliant solution declares `malloc()` by including the appropriate header file:

```c
#include <stdlib.h>

int main(void) {
    for (size_t i = 0; i < 100; ++i) {
        char *ptr = (char *)malloc(0x10000000);
        *ptr = 'a';
    }
    return 0;
}
```

For more information on function declarations, see DCL07-C. Include the appropriate type information in function declarators.

**Noncompliant Code Example (Implicit Return Type)**

Do not declare a function with an implicit return type. For example, if a function returns a meaningful integer value, declare it as returning `int`. If it returns no meaningful value, declare it as returning `void`.

```c
#include <limits.h>
#include <stdio.h>

void foo(void) {
    return UINT_MAX;
}

int main(void) {
    long long int c = foo();
    printf("%lld\n", c);
    return 0;
}
```

Because the compiler assumes that `foo()` returns a value of type `int` for this noncompliant code example, `UINT_MAX` is incorrectly converted to 1.

**Compliant Solution (Implicit Return Type)**

This compliant solution explicitly defines the return type of `foo()` as `unsigned int`. As a result, the function correctly returns `UINT_MAX`.

```c
#include <limits.h>
#include <stdio.h>

unsigned int foo(void) {
    return UINT_MAX;
}

int main(void) {
    long long int c = foo();
    printf("%lld\n", c);
    return 0;
}
```

**Risk Assessment**

Because implicit declarations lead to less stringent type checking, they can introduce unexpected and erroneous behavior. Occurrences of an omitted type specifier in existing code are rare, and the consequences are generally minor, perhaps resulting in abnormal program termination.
Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>19.04</td>
<td>type-specifier</td>
<td>Fully checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function-return-type</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>implicit-function-declaration</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>undeclared-parameter</td>
<td></td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-DCL31</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>-Wimplicit-int</td>
<td></td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>MISRA C 2012 Rule 8.1</td>
<td>Implemented</td>
</tr>
<tr>
<td>ECLAIR</td>
<td>1.2</td>
<td>CC2.DCL31</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>GCC</td>
<td>4.3.5</td>
<td></td>
<td>Can detect violations of this rule when the -Wimplicit and -Wreturn-type flags are used</td>
</tr>
<tr>
<td>Klocwork</td>
<td>2018</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CWARN.IMPLICITINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FUNCRET.IMPLICIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MISRA.DECL.NO_TYPE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MISRA.FUNC.NOPROT.CALL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETVOID.IMPLICIT</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>24 D, 41 D, 20 S, 326 S, 496 S</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-DCL31-a</td>
<td>All functions shall be declared before use</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule DCL31-C</td>
<td>Checks for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Types not explicitly specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Implicit function declaration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule fully covered</td>
<td></td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0434 (C)</td>
<td>Fully implemented</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2050</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2051</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3335</td>
<td></td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.07</td>
<td>V1031</td>
<td></td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>S819, S820</td>
<td>Partially implemented; implicit return type not covered.</td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>type-specifier</td>
<td>Fully checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function-return-type</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>implicit-function-declaration</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>undeclared-parameter</td>
<td></td>
</tr>
<tr>
<td>TrustInSoft Analyzer</td>
<td>1.38</td>
<td>type specifier missing</td>
<td>Partially verified (exhaustively detects undefined behavior).</td>
</tr>
</tbody>
</table>

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)

Bibliography

[ISO/IEC 9899:1990]


[Jones 2008]