DCL20-C. Explicitly specify void when a function accepts no arguments


An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.

Subclause 6.11.6 states that

The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

Consequently, functions that accept no arguments should explicitly declare a void parameter in their parameter list. This holds true in both the declaration and definition sections (which should match).

Defining a function with a void argument list differs from declaring it with no arguments because, in the latter case, the compiler will not check whether the function is called with parameters at all [TIGCC, void usage]. Consequently, function calling with arbitrary parameters will be accepted without a warning at compile time.

Failure to declare a void parameter will result in

- An ambiguous functional interface between the caller and callee.
- Sensitive information outflow.

A similar recommendation deals with parameter type in a more general sense: DCL07-C. Include the appropriate type information in function declarators.

Noncompliant Code Example (Ambiguous Interface)

In this noncompliant code example, the caller calls foo() with an argument of 3. The caller expects foo() to accept a single int argument and to output the argument as part of a longer message. Because foo() is declared without the void parameter, the compiler will not perform any caller check. It is therefore possible that the caller may not detect the error. In this example, for instance, foo() might output the value 3 as expected.

Because no function parameter has the same meaning as an arbitrary parameter, the caller can provide an arbitrary number of arguments to the function.

```c
/* In foo.h */
void foo();

/* In foo.c */
void foo() {
    int i = 3;
    printf("i value: \d\n", i);
}

/* In caller.c */
#include "foo.h"

foo(3);
```

Compliant Solution (Ambiguous Interface)

In this compliant solution, void is specified explicitly as a parameter in the declaration of foo's prototype:

```c
/* In foo.h */
void foo(void);

/* In foo.c */
void foo() {
    int i = 3;
    printf("i value: \d\n", i);
}

/* In caller.c */
#include "foo.h"

foo();
```
/* In foo.h */
void foo(void);

/* In foo.c */
void foo(void) {
    int i = 3;
    printf("i value: \d\n", i);
}

/* In caller.c */
#include "foo.h"
#include "foo.h"

Implementation Details (Ambiguous Interface)

When the compliant solution is used and foo(3) is called, the GCC compiler issues the following diagnostic, which alerts the programmer about the misuse of the function interface:

    error: too many arguments to function "foo"

Noncompliant Code Example (Information Outflow)

Another possible vulnerability is the leak of privileged information. In this noncompliant code example, a user with high privileges feeds some secret input to the caller that the caller then passes to foo(). Because of the way foo() is defined, we might assume there is no way for foo() to retrieve information from the caller. However, because the value of i is really passed into a stack (before the return address of the caller), a malicious programmer can change the internal implementation and copy the value manually into a less privileged file.

    /* Compile using gcc4.3.3 */
    void foo() {
        /*
        * Use assembly code to retrieve i
        * implicitly from caller
        * and transfer it to a less privileged file.
        */
    }
    ...
    /* Caller */
    foo(i); /* i is fed from user input */

Compliant Solution (Information Outflow)

    void foo(void) {
        int i = 3;
        printf("i value: \d\n", i);
    }

Again, the simplest solution is to explicitly specify void as the only parameter.

Risk Assessment

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL20-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Low</td>
<td>P12</td>
<td>L1</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>Version</td>
<td>Rule</td>
<td>Violation Type</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------</td>
<td>-----------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Astrée</td>
<td>19.04</td>
<td>empty-parameter-list</td>
<td>Fully checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-DCL20</td>
<td></td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>LANG_FUNCS_PROT</td>
<td>Incomplete function prototype</td>
</tr>
<tr>
<td>Klocwork</td>
<td>2018</td>
<td>MISRA_FUNC_NO_PARAMS</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>63 S</td>
<td>Fully Implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-DCL20-a</td>
<td>The number of arguments passed to a function shall match the number of parameters</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>3001, 3007</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>empty-parameter-list</td>
<td>Fully checked</td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>S929</td>
<td></td>
</tr>
</tbody>
</table>

### Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](https://cert.org).

### Related Guidelines

In C++, `foo()` and `foo(void)` have exactly the same meaning and effect, so this rule doesn't apply to C++. However, `foo(void)` should be declared explicitly instead of `foo()` to distinguish it from `foo(...)`, which accepts an arbitrary number and type of arguments.

MISRA C:2012 Rule 8.2 (required)

### Bibliography

<table>
<thead>
<tr>
<th>Source</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subclause 6.11.6, &quot;Function Declarators&quot;</td>
</tr>
<tr>
<td>[TIGCC, void usage]</td>
<td>Manual, &quot;C Language Keywords&quot;: void</td>
</tr>
</tbody>
</table>