DCL03-C. Use a static assertion to test the value of a constant expression

Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see MSC11-C. Incorporate diagnostic tests using assertions). The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls a bork(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems.

Static assertion is a new facility in the C Standard. It takes the form

\`
static_assert(constant-expression, string-literal);
\`


> The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, except that characters not in the basic source character set are not required to appear in the message.

It means that if constant-expression is true, nothing will happen. However, if constant-expression is false, an error message containing string-literal will be output at compile time.

\`
/* Passes */
static_assert(
    sizeof(int) <= sizeof(void*),
    "sizeof(int) <= sizeof(void*)"
);

/* Fails */
static_assert(
    sizeof(double) <= sizeof(int),
    "sizeof(double) <= sizeof(int)"
);
\`

Static assertion is not available in C99.

Noncompliant Code Example

This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly:

\`
#include <assert.h>

struct timer {
    unsigned char MODE;
    unsigned int DATA;
    unsigned int COUNT;
};

int func(void) {
    assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));
}
\`

Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed.

Compliant Solution

For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution:
Using `#error` directives allows for clear diagnostic messages. Because this approach evaluates assertions at compile time, there is no runtime penalty.

### Compliant Solution

This portable compliant solution uses `static_assert`:

```c
#include <assert.h>

struct timer {
    unsigned char MODE;
    unsigned int DATA;
    unsigned int COUNT;
};

static_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),
              "Structure must not have any padding Quo";
```

Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Because the assertion is performed at compile time, no runtime cost in space or time is incurred. An assertion can be used at file or block scope, and failure results in a meaningful and informative diagnostic error message.

Other uses of static assertion are shown in STR07-C. Use the bounds-checking interfaces for string manipulation and FIO34-C. Distinguish between characters read from a file and EOF or WEOF.

### Risk Assessment

Static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities at compile time. The absence of static assertions, however, does not mean that code is incorrect.

<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>DCL03-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>High</td>
<td>P1</td>
<td>L3</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>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-DCL03</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>misc-</td>
<td>Checked by clang-tidy</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.3p0</td>
<td>(customization)</td>
<td>Users can implement a custom check that reports uses of the assert() macro</td>
</tr>
<tr>
<td>Compass /ROSE</td>
<td></td>
<td></td>
<td>Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation</td>
</tr>
<tr>
<td>ECLAIR</td>
<td>1.2</td>
<td>CC2, DCL03</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>44 S</td>
<td>Fully implemented</td>
</tr>
</tbody>
</table>

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

Related Guidelines

| C++ Secure Coding Standard | VOID DCL03-CPP. Use a static assertion to test the value of a constant expression |

Bibliography

[Becker 2008]

[Eckel 2007]


[Jones 2010]

[Klarer 2004]

[Saks 2005]

[Saks 2008]