FLP07-C. Cast the return value of a function that returns a floating-point type

Cast the return value of a function that returns a floating point type to ensure predictable program execution.


If a return statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.

This paragraph is annotated (footnote 160) as follows:

The return statement is not an assignment. The overlap restriction of subclause 6.5.16.1 does not apply to the case of function return. The representation of floating-point values may have wider range or precision than implied by the type; a cast may be used to remove this extra range and precision.

Conversion as if by assignment to the type of the function is required if the return expression has a different type than the function but not if the return expression has a wider value only because of wide evaluation. This allows seemingly inconsistent and confusing behavior. Consider the following code as an example:

```c
float f(float x) {
    return x * 0.1f;
}
float g(float x) {
    return x * 0.1;
}
```

Function `f()` is allowed to return a value wider than `float`, but function `g()` (which uses the wider constant) is not.

Although the standard does not require narrowing return expressions of the same type as the function, it does not clearly state what is allowed. Is it allowed to narrow the result? Is it allowed to narrow the result sometimes but not always? Is it allowed to partially narrow the result (for example, if the application binary interface [ABI] returns floats in double format, but a float function has a float return expression evaluated to wider than double)? An aggressive implementation could argue yes to all these questions, though the resulting behavior would complicate debugging and error analysis.

Footnote 160 in the C Standard says a cast may be used to remove extra range and precision from the return expression. This means that a predictable program must have casts on all function calls that return floating-point values (except where the function directly feeds an operator-like assignment that implies the conversion). With type-generic math (tgmath.h), the programmer has to reason through the tgmath.h resolution rules to determine which casts to apply. These are significant obstacles to writing predictable code.

The C Standard, subclause F.6 [ISO/IEC 9899:2011], states:

> If the return expression is evaluated in a floating-point format different from the return type, the expression is converted as if by assignment\(^{362}\) to the return type of the function and the resulting value is returned to the caller.

362 Assignment removes any extra range and precision.

This applies only to implementations that conforms to the optional Annex F, "IEC 60559 Floating-Point Arithmetic." The macro `__STDC_IEC_559__` can be used to determine whether an implementation conforms to Annex F.

Noncompliant Code Example

This noncompliant code example fails to cast the result of the expression in the return statement and thereby guarantee that the range or precision is no wider than expected. The uncertainty in this example is introduced by the constant `0.1f`. This constant may be stored with a range or precision that is greater than that of `float`. Consequently, the result of `x * 0.1f` may also have a range or precision greater than that of `float`. As described previously, this range or precision may not be reduced to that of a `float`, so the caller of `calcPercentage()` may receive a value that is more precise than expected. This may lead to inconsistent program execution across different platforms.
float calc_percentage(float value) {
    return value * 0.1f;
}

void float_routine(void) {
    float value = 99.0f;
    long double percentage;

    percentage = calc_percentage(value);
}

Compliant Solution (within the Function)

This compliant solution casts the value of the expression in the return statement. It forces the return value to have the expected range and precision, as described in subclause 5.2.4.2.2, paragraph 9, of the C Standard [ISO/IEC 9899:2011].

float calc_percentage(float value) {
    return (float)(value * 0.1f);
}

void float_routine(void) {
    float value = 99.0f;
    long double percentage;

    percentage = calc_percentage(value);
}

Forcing the range and precision inside the `calcPercentage()` function is a good way to fix the problem once without having to apply fixes in multiple locations (every time `calcPercentage()` is called).

Compliant Solution (Outside the Function)

Source code to the called function may not always be available. This compliant solution casts the return value of the `calcPercentage()` function to `float` to force the correct range and precision when the source of the called function cannot be modified.

void float_routine(void) {
    float value = 99.0f;
    long double percentage;

    percentage = (float) calc_percentage(value);
}

Risk Assessment

Failure to follow this guideline can lead to inconsistent results across different platforms.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLP07-C</td>
<td>Low</td>
<td>Probable</td>
<td>Medium</td>
<td>P4</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-FLP07</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Annex F.6, &quot;The return Statement&quot;</td>
</tr>
<tr>
<td>[WG14/N1396]</td>
<td></td>
</tr>
</tbody>
</table>