FLP03-C. Detect and handle floating-point errors

Errors during floating-point operations are often neglected by programmers who instead focus on validating operands before an operation. Errors that occur during floating-point operations are admittedly difficult to determine and diagnose, but the benefits of doing so often outweigh the costs. This recommendation suggests ways to capture errors during floating-point operations.

The following code exhibits undefined behavior:

```c
int j = 0;
int iResult = 1 / j;
```

On most implementations, integer division by zero is a terminal error, commonly printing a diagnostic message and aborting the program:

```c
double x = 0.0;
double dResult = 1 / x;
```

Floating-point division by zero also results in undefined behavior, although most implementations do not treat it as a terminal error. If additional precautions are not taken, it results in a silent error.

The most portable way to determine if a floating-point exceptional condition has occurred is to use the floating-point exception facilities provided by C in `fenv.h`.

However, the C floating-point exception functions are not without problems. The following caveats exist regarding the interaction between floating-point exceptions and conversions:

- Conversion from floating-point to integer may cause an "invalid" floating-point exception. If this occurs, the value of that integer is undefined and should not be used.
- Most implementations fail to raise "invalid" for conversions from any negative or "large" positive floating-point values to unsigned integer types or to `signed char`. (See `tfl2int.c`.)
- When a noninteger floating-point value is converted to an integer, the "inexact" floating-point exception is raised.

For information regarding floating-point number conversions, see FLP34-C. Ensure that floating-point conversions are within range of the new type.

The C Standard does not require all implementations to support floating-point exceptions. Each exception macro in `fenv.h` is defined if, and only if, the corresponding exception is supported. Only implementations that use IEC 60559 (formerly IEEE-754) floating-point arithmetic are required to support all five exceptions defined by C (see the C Standard, subclause 7.6.2 [ISO/IEC 9899:2011]). Nevertheless, these functions are the most portable solution for handling floating-point exceptions.

A less portable but potentially more secure solution is to use the capabilities provided by the underlying implementation. If this approach is taken, the caveats of that system must be well understood. The following table provides a starting point for some common operating systems:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>How to Handle Floating-Point Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Use the C floating-point exception functions</td>
</tr>
<tr>
<td>Solaris 10</td>
<td></td>
</tr>
<tr>
<td>AIX 5.3</td>
<td></td>
</tr>
<tr>
<td>HP-UX 11.31</td>
<td></td>
</tr>
<tr>
<td>Mac OS X 10.5</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>Use either the C floating-point exception functions or structured exception handling through <code>_fpiece_flt</code> [MSDN]</td>
</tr>
</tbody>
</table>

Noncompliant Code Example

In this noncompliant code example, floating-point operations are performed without checking for errors. Note that range checking has been intentionally omitted because the intent is to detect errors following the floating-point operation.

```c
void fpOper_noErrorChecking(void) {
    /* ... */
    double a = 1e-40, b, c = 0.1;
    float x = 0, y;
    /* Inexact and underflows */
    y = a;
    /* Divide-by-zero operation */
    b = y / x;
    /* Inexact (loss of precision) */
    c = sin(30) * a;
    /* ... */
}
```
However, exceptional conditions (as indicated by the comments) occur that may lead to unexpected arithmetic results.

Compliant Solution (C)

This compliant solution uses C Standard functions to handle floating-point errors:

```c
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

void fpOper_fenv(void) {
    double a = 1e-40, b, c = 0.1;
    float x = 0, y;
    int fpeRaised;
    /* ... */

    feclearexcept(FE_ALL_EXCEPT);
    /* Store a into y is inexact and underflows: */
    y = a;
    fpeRaised = fetestexcept(FE_ALL_EXCEPT);
    /* fpeRaised has FE_INEXACT and FE_UNDERFLOW */

    feclearexcept(FE_ALL_EXCEPT);
    /* Divide-by-zero operation */
    b = y / x;
    fpeRaised = fetestexcept(FE_ALL_EXCEPT);
    /* fpeRaised has FE_DIVBYZERO */

    feclearexcept(FE_ALL_EXCEPT);
    c = sin(30) * a;
    fpeRaised = fetestexcept(FE_ALL_EXCEPT);
    /* fpeRaised has FE_INEXACT */

    feclearexcept(FE_ALL_EXCEPT);
    /* ... */
}
```

Compliant Solution (Windows)

Microsoft Visual Studio 2008 and earlier versions do not support C functions to handle floating-point errors. Windows provides an alternative method using `_statusfp()`, `_statusfp2()`, and `_clearfp()`.

```c
void fpOper_usingStatus(void) {
    /* ... */
    double a = 1e-40, b, c;
    float x = 0, y;
    unsigned int rv = _clearfp();

    /* Store into y is inexact and underflows: */
    y = a;
    rv = _clearfp(); /* rv has _SW_INEXACT and _SW_UNDERFLOW */

    /* Zero-divide */
    b = y / x; rv = _clearfp(); /* rv has _SW_ZERO_DIVIDE */

    /* Inexact */
    c = sin(30) * a; rv = _clearfp(); /* rv has _SW_INEXACT */
    /* ... */
}
```

Compliant Solution (Windows SEH)

Microsoft Visual Studio 2008 also uses structured exception handling (SEH) to handle floating-point operations. SEH provides more information about the error and allows the programmer to change the results of the floating-point operation that caused the error condition.
void fp_usingSEH(void) {
    /* ... */
    double a = 1e-40, b, c = 0.1;
    float x = 0, y;
    unsigned int rv;

    unmask_fpsr();

    _try {
        /* Store into y is inexact and underflows: */
        y = a;

        /* Divide-by-zero operation */
        b = y / x;

        /* Inexact */
        c = sin(30) * a;
    }

    _except (_fpieee_flt(
        GetExceptionCode(),
        GetExceptionInformation(),
        fpieee_handler)) {
        {
            printf("fpieee_handler: EXCEPTION_EXECUTE_HANDLER");
        }
    }

    /* ... */
}

void unmask_fpsr(void) {
    unsigned int u;
    unsigned int control_word;
    _controlfp_s(&control_word, 0, 0);
    u = control_word & ~(_EM_INVALID
        | _EM_DENORMAL
        | _EM_ZERODIVIDE
        | _EM_OVERFLOW
        | _EM_UNDERFLOW
        | _EM_INEXACT);
    _controlfp_s(&control_word, u, _MCW_EM);
    return;
}

int fpieee_handler(_FPIEEE_RECORD *ieee) {
    /* ... */

    switch (ieee->RoundingMode) {
        case _FpRoundNearest:
            /* ... */
            break;

        /*
            * Other RMs include _FpRoundMinusInfinity,
            * _FpRoundPlusInfinity, _FpRoundChopped.
            */

            /* ... */
    }

    switch (ieee->Precision) {
        case _FpPrecision24:
            /* ... */
            break;

            /* Other Ps include _FpPrecision53 */

            /* ... */
    }

    switch (ieee->Operation) {
case _FpCodeAdd:
    /* ... */
    break;

    /*
    * Other Ops include _FpCodeSubtract, _FpCodeMultiply,
    * _FpCodeDivide, _FpCodeSquareRoot, _FpCodeCompare,
    */
    /* ... */
}

    /*
    * Process the bitmap ieee->Cause.
    * Process the bitmap ieee->Enable.
    * Process the bitmap ieee->Status.
    * Process the Operand ieee->Operand1,
    * evaluate format and Value.
    * Process the Operand ieee->Operand2,
    * evaluate format and Value.
    * Process the Result ieee->Result,
    * evaluate format and Value.
    * The result should be set according to the operation
    * specified in ieee->Cause and the result formatted as
    * specified in ieee->Result.
    */
    /*
    /* ... */
}

Risk Assessment

Undetected floating-point errors may result in lower program efficiency, inaccurate results, or software vulnerabilities. Most processors stall for a significant duration when an operation incurs a NaN (not a number) value.

<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>FLP03-C</td>
<td>Low</td>
<td>Probable</td>
<td>High</td>
<td>P2</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>Astrée</td>
<td>19.04</td>
<td>float-division-by-zero</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Compass /ROSE</td>
<td>9.7.1</td>
<td>43 D</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>10.4.2</td>
<td>CERT_C-FLP03-a, CERT_C-FLP03-b, CERT_C-FLP03-c, CERT_C-FLP03-d</td>
<td>Avoid division by zero, Avoid implicit conversions from wider to narrower floating type, Avoid implicit conversions from narrower to wider floating type, Avoid implicit conversions of floating point numbers from wider to narrower floating type</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>Runtime analysis</td>
<td></td>
</tr>
<tr>
<td>Parasoft Insure++</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rationale

The rationale for this rule is to ensure that floating-point operations are handled correctly and that NaN values are managed appropriately. This is crucial for maintaining the integrity and performance of software systems.
<table>
<thead>
<tr>
<th>Polyspace Bug Finder</th>
<th>R2019b</th>
<th>CERT C: Rec. FLP03-C</th>
<th>Checks for:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Float conversion overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Float overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Float division by zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rec. partially covered.</td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

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

**Related Guidelines**

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>VOID FLP03-CPP. Detect and handle floating point errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>MITRE CWE</td>
<td>CWE-369, Divide by zero</td>
</tr>
</tbody>
</table>

**Bibliography**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>[IEEE Std 1003.1:2013]</td>
<td>XBD, Headers, <code>&lt;fenv.h&gt;</code></td>
</tr>
<tr>
<td>[Intel 2001]</td>
<td></td>
</tr>
<tr>
<td>[Keil 2008]</td>
<td></td>
</tr>
<tr>
<td>[MSDN]</td>
<td><em>fpiee_flt (CRT)</em></td>
</tr>
<tr>
<td>[SecurityFocus 2007]</td>
<td></td>
</tr>
</tbody>
</table>