FLP32-C. Prevent or detect domain and range errors in math functions

The C Standard, 7.12.1 [ISO/IEC 9899:2011], defines three types of errors that relate specifically to math functions in `<math.h>`. Paragraph 2 states

A domain error occurs if an input argument is outside the domain over which the mathematical function is defined.

Paragraph 3 states

A pole error (also known as a singularity or infinitary) occurs if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit.

Paragraph 4 states

A range error occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

An example of a domain error is the square root of a negative number, such as \( \sqrt{-1.0} \), which has no meaning in real arithmetic. Contrastingly, 10 raised to the 1-millionth power, \( 10^{1e6} \), cannot be represented in many floating-point implementations because of the limited range of the type `double` and consequently constitutes a range error. In both cases, the function will return some value, but the value returned is not the correct result of the computation. An example of a pole error is \( \log(0.0) \), which results in negative infinity.

Programmers can prevent domain and pole errors by carefully bounds-checking the arguments before calling mathematical functions and taking alternative action if the bounds are violated.

Range errors usually cannot be prevented because they are dependent on the implementation of floating-point numbers as well as on the function being applied. Instead of preventing range errors, programmers should attempt to detect them and take alternative action if a range error occurs.

The following table lists the `double` forms of standard mathematical functions, along with checks that should be performed to ensure a proper input domain, and indicates whether they can also result in range or pole errors, as reported by the C Standard. Both `float` and `long double` forms of these functions also exist but are omitted from the table for brevity. If a function has a specific domain over which it is defined, the programmer must check its input values. The programmer must also check for range errors where they might occur. The standard math functions not listed in this table, such as `fabs()`, have no domain restrictions and cannot result in range or pole errors.

<table>
<thead>
<tr>
<th>Function</th>
<th>Domain</th>
<th>Range</th>
<th>Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(x)</td>
<td>(-1 \leq x &amp;&amp; x \leq 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>asin(x)</td>
<td>(-1 \leq x &amp;&amp; x \leq 1)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>atan(x)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>atan2(y, x)</td>
<td>x != 0 &amp;&amp; y != 0</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>acosh(x)</td>
<td>x \geq 1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>asinh(x)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>atanh(x)</td>
<td>(-1 &lt; x &amp;&amp; x &lt; 1)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cosh(x), sinh(x)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>exp(x), exp2(x), expm1(x)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ldexp(x, exp)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>log(x), log10(x), log2(x)</td>
<td>x \geq 0</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>log1p(x)</td>
<td>x \geq -1</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ilogb(x)</td>
<td>x != 0 &amp;&amp; !isinf(x) &amp;&amp; !isnan(x)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>logb(x)</td>
<td>x != 0</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>scalbn(x, n), scalbln(x, n)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>hypot(x, y)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>(x &gt; 0 || (x == 0 &amp;&amp; y &gt; 0) || (x &lt; 0 &amp;&amp; y \text{ is an integer}))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>x \geq 0</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>erf(x)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>erfc(x)</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Function</td>
<td>Domain and Pole Checking</td>
<td>Range Checking</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>lgamma(x), tgamma(x)</td>
<td>x ≠ 0, x is an integer</td>
<td>Yes, Yes</td>
<td></td>
</tr>
<tr>
<td>lrint(x), lround(x)</td>
<td>None</td>
<td>Yes, No</td>
<td></td>
</tr>
<tr>
<td>fmod(x, y), remainder(x, y), remquo(x, y, quo)</td>
<td>y ≠ 0</td>
<td>Yes, No</td>
<td></td>
</tr>
<tr>
<td>nextafter(x, y), nexttoward(x, y)</td>
<td>None</td>
<td>Yes, No</td>
<td></td>
</tr>
<tr>
<td>fdiv(x, y)</td>
<td>None</td>
<td>Yes, No</td>
<td></td>
</tr>
<tr>
<td>fma(x, y, z)</td>
<td>None</td>
<td>Yes, No</td>
<td></td>
</tr>
</tbody>
</table>

The most reliable way to handle domain and pole errors is to prevent them by checking arguments beforehand, as in the following exemplar:

```c
double safe_sqrt(double x) {
    if (x < 0) {
        fprintf(stderr, "sqrt requires a nonnegative argument\n");
        /* Handle domain / pole error */
    }
    return sqrt(x);
}
```

Range Checking

Programmers usually cannot prevent range errors, so the most reliable way to handle them is to detect when they have occurred and act accordingly.

The exact treatment of error conditions from math functions is tedious. The C Standard, 7.12.1 [ISO/IEC 9899:2011], defines the following behavior for floating-point overflow:

A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the function returns the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL according to the return type, with the same sign as the correct value of the function; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "overflow" floating-point exception is raised.

It is preferable not to check for errors by comparing the returned value against HUGE_VAL or 0 for several reasons:

- These are, in general, valid (albeit unlikely) data values.
- Making such tests requires detailed knowledge of the various error returns for each math function.
- Multiple results aside from HUGE_VAL and 0 are possible, and programmers must know which are possible in each case.
- Different versions of the library have varied in their error-return behavior.

It can be unreliable to check for math errors using `errno` because an implementation might not set `errno`. For real functions, the programmer determines if the implementation sets `errno` by checking whether `math_errhandling & MATH_ERRNO` is nonzero. For complex functions, the C Standard, 7.3.2, paragraph 1, simply states that "an implementation may set `errno` but is not required to." [ISO/IEC 9899:2011]

The obsolete System V Interface Definition (SVID3) [UNIX 1992] provides more control over the treatment of errors in the math library. The programmer can define a function named `matherr()` that is invoked if errors occur in a math function. This function can print diagnostics, terminate the execution, or specify the desired return value. The `matherr()` function has not been adopted by C or POSIX, so it is not generally portable.

The following error-handling template uses C Standard functions for floating-point errors when the C macro `math_errhandling` is defined and indicates that they should be used; otherwise, it examines `errno`:
See FLP03-C. Detect and handle floating-point errors for more details on how to detect floating-point errors.

Subnormal Numbers

A subnormal number is a nonzero number that does not use all of its precision bits [IEEE 754 2006]. These numbers can be used to represent values that are closer to 0 than the smallest normal number (one that uses all of its precision bits). However, the \texttt{asin()}, \texttt{asinh()}, \texttt{atan()}, \texttt{atanh()}, and \texttt{erf()} functions may produce range errors, specifically when passed a subnormal number. When evaluated with a subnormal number, these functions can produce an inexact, subnormal value, which is an underflow error. The C Standard, 7.12.1, paragraph 6 [ISO/IEC 9899:2011], defines the following behavior for floating-point underflow:

The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression \texttt{math_errno} \& MATH_ERRNO is nonzero, \texttt{errno} acquires the value \texttt{ERANGE} is implementation-defined; if the integer expression \texttt{math_errno} \& MATH_ERREXCEPT is nonzero, whether the “underflow” floating-point exception is raised is implementation-defined.

Implementations that support floating-point arithmetic but do not support subnormal numbers, such as IBM S/360 hex floating-point or nonconforming IEEE-754 implementations that skip subnormals (or support them by flushing them to zero), can return a range error when calling one of the following families of functions with the following arguments:

- \texttt{fmod(\texttt{min+subnorm}, \texttt{min})}
- \texttt{remainder(\texttt{min+subnorm}, \texttt{min})}
- \texttt{remquo(\texttt{min+subnorm}, \texttt{min}, \texttt{quo})}

where \texttt{min} is the minimum value for the corresponding floating point type and \texttt{subnorm} is a subnormal value.

If Annex F is supported and subnormal results are supported, the returned value is exact and a range error cannot occur. The C Standard, F.10.7.1 [ISO /IEC 9899:2011], specifies the following for the \texttt{fmod()}, \texttt{remainder()}, and \texttt{remquo()} functions:

When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

Annex F, subclause F.10.7.2, paragraph 2, and subclause F.10.7.3, paragraph 2, of the C Standard identify when subnormal results are supported.

Noncompliant Code Example (\texttt{sqrt()} )

This noncompliant code example determines the square root of \( x \):
```c
#include <math.h>
void func(double x) {
    double result;
    result = sqrt(x);
}
```

However, this code may produce a domain error if \( x \) is negative.

**Compliant Solution (sqrt())**

Because this function has domain errors but no range errors, bounds checking can be used to prevent domain errors:

```c
#include <math.h>
void func(double x) {
    double result;
    if (isless(x, 0.0)) {
        /* Handle domain error */
    }
    result = sqrt(x);
}
```

**Noncompliant Code Example (sinh(), Range Errors)**

This noncompliant code example determines the hyperbolic sine of \( x \):

```c
#include <math.h>
void func(double x) {
    double result;
    result = sinh(x);
}
```

This code may produce a range error if \( x \) has a very large magnitude.

**Compliant Solution (sinh(), Range Errors)**

Because this function has no domain errors but may have range errors, the programmer must detect a range error and act accordingly:
void func(double x) {
    double result;
    {
        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;
        result = sinh(x);
    
    if ((math_errhandling & MATH_ERRNO) && errno != 0) {
        /* Handle range error */
    } else if ((math_errhandling & MATH_ERREXCEPT) &&
        fetestexcept(FE_INVALID | FE_DIVBYZERO |
            FE_OVERFLOW | FE_UNDERFLOW) != 0) {
        /* Handle range error */
    }
    /* Use result... */
} 

Noncompliant Code Example (pow())

This noncompliant code example raises \(x\) to the power of \(y\):

```c
#include <math.h>
void func(double x, double y) {
    double result;
    result = pow(x, y);
} 
```

This code may produce a domain error if \(x\) is negative and \(y\) is not an integer value or if \(x\) is 0 and \(y\) is 0. A domain error or pole error may occur if \(x\) is 0 and \(y\) is negative, and a range error may occur if the result cannot be represented as a double.

Compliant Solution (pow())

Because the `pow()` function can produce domain errors, pole errors, and range errors, the programmer must first check that \(x\) and \(y\) lie within the proper domain and do not generate a pole error and then detect whether a range error occurs and act accordingly:
```c
#include <math.h>
#include <fenv.h>
#include <errno.h>

void func(double x, double y) {
    double result;
    if (((x == 0.0f) && islessequal(y, 0.0)) || isless(x, 0.0)) {
        /* Handle domain or pole error */
    } else {
        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;
        result = pow(x, y);
        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                                 FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }
    /* Use result... */
}
```

**Noncompliant Code Example (asin(), Subnormal Number)**

This noncompliant code example determines the inverse sine of x:

```c
#include <math.h>

void func(float x) {
    float result = asin(x);
    /* ... */
}
```

**Compliant Solution (asin(), Subnormal Number)**

Because this function has no domain errors but may have range errors, the programmer must detect a range error and act accordingly:
```c
#include <math.h>
#include <fenv.h>
#include <errno.h>

void func(float x) {
    float result;
    
    #pragma STDC FENV_ACCESS ON
    if (math_errhandling & MATH_ERREXCEPT) {
        feclearexcept(FE_ALL_EXCEPT);
    }
    errno = 0;
    result = asin(x);
    if ((math_errhandling & MATH_ERRNO) && errno != 0) {
        /* Handle range error */
    } else if ((math_errhandling & MATH_ERREXCEPT) &&
               fetestexcept(FE_INVALID | FE_DIVBYZERO |
                            FE_OVERFLOW | FE_UNDERFLOW) != 0) {
        /* Handle range error */
    }
    /* Use result... */
}
```

**Risk Assessment**

Failure to prevent or detect domain and range errors in math functions may cause unexpected results.

<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>FLP32-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Medium</td>
<td>P8</td>
<td>L2</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>20.10</td>
<td>stdlib-limits</td>
<td>Partially checked</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.4p0</td>
<td>MATH.DOMAIN.ATAN, MATH.DOMAIN.TOOHIGH, MATH.DOMAIN.TOLLOW, MATH.DOMAIN.MATH.RANGE, MATH.RANGE.GAMMA, MATH.DOMAIN.LOG, MATH.RANGE.LOG, MATH.DOMAIN, FE_INVALID, MATH.DOMAIN.POW, MATH.RANGE.COSH, MATH.DOMAIN.SQRT</td>
<td>Arctangent Domain Error, Argument Too High, Argument Too Low, Floating Point Domain Error, Floating Point Range Error, Gamma on Zero, Logarithm on Negative Value, Logarithm on Zero, Raises FE_INVALID, Undefined Power of Zero, cosh on High Number, cosh on Low Number, sqrt on Negative Value</td>
</tr>
<tr>
<td>Parasoft C /C++test</td>
<td>2020.2</td>
<td>CERT_C-FLP32-a</td>
<td>Validate values passed to library functions</td>
</tr>
<tr>
<td>PC-lint Plus</td>
<td>1.4</td>
<td>2423</td>
<td>Partially supported: reports domain errors for functions with the Semantics &quot;dom_1&quot;, &quot;dom.lt0&quot;, or &quot;dom.lt1&quot;, including standard library math functions</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2020a</td>
<td>CERT-C. Rule FLP32-C</td>
<td>Checks for invalid use of standard library floating point routine (rule partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>5025</td>
<td></td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>5033</td>
<td></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)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>FLP03-C. Detect and handle floating-point errors</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-682, Incorrect Calculation</td>
<td>2017-07-07: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes
Key here for mapping notes

CWE-391 and FLP32-C
Intersection( CWE-391, FLP32-C) =

- Failure to detect range errors in floating-point calculations

CWE-391 - FLP32-C

- Failure to detect errors in functions besides floating-point calculations

FLP32-C – CWE-391 =

- Failure to detect domain errors in floating-point calculations

CWE-682 and FLP32-C
Independent( INT34-C, FLP32-C, INT33-C) CWE-682 = Union( FLP32-C, list) where list =

- Incorrect calculations that do not involve floating-point range errors

Bibliography

7.12.1, "Treatment of Error Conditions"
F.10.7, "Remainder Functions"

[IEEE 754 2006 ]

[Plum 1985]  Rule 2-2

[Plum 1989]  Topic 2.10, "conv—Conversions and Overflow"

[UNIX 1992]  System V Interface Definition (SVID3)