C++ Interval Arithmetic Programming Reference

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-6146-10
October 2000, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Tables

Code Examples

Preface

This manual documents the C++ interface to the C++ interval arithmetic library provided with the Sun WorkShop™ 6 update 1 Compilers C++ (5.2).

Who Should Use This Book

This is a reference manual intended for programmers with a working knowledge of the C++ language, the Solaris™ operating environment, and UNIX commands.

What Is in This Book

This book contains the following chapters:

Chapter 1, “Using the Interval Arithmetic Library,” describes the C++ interval arithmetic support goals and provides code samples that interval programmers can use to learn more about the C++ interval features.

Chapter 2, “C++ Interval Arithmetic Library Reference,” describes the C++ interval arithmetic library interface.

“Glossary” contains definitions of interval terms.

What Is Not in This Book

This book is not an introduction to intervals and does not contain derivations of the interval innovations included in the interval arithmetic C++ library. For a list of sources containing introductory interval information, see the interval arithmetic readme.

What Typographic Changes Mean

The following table describes the typographic conventions used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	Code samples, the names of commands, files, and directories; on-screen computer output	`interval<double>("[4, 5]"))`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`math%` **`CC -xia test.cc`** `math%` **`a.out`** `x = [2.0,3.0]`
`AaBbCc123`	Placeholders for `interval` language elements	The `interval` affirmative order relational operators *op* ∈ {`lt`, `le`, `eq`, `ge`, `gt`} are equivalent to the mathematical operators op ∈ { <, ≤, =, ≥, > }.
AaBbCc123	Variables used in equations, book titles, new words or terms, or words to be emphasized	The C++ code equivalent of $X \cap Y$ is `intersect(X,Y)`
AaBbCc123	Command-line placeholder: replace with a real name or value	To invoke `CC` with class `interval` support, type `math%` **`CC -xia`** *source_file*`.cc`

Note – Examples use `math%` as the system prompt.

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your `PATH` environment variable:

1. **Display the current value of the `PATH` variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing `/opt/SUNWspro/bin/`.**

   If you find the paths, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your `PATH` environment variable by following the instructions in this section.

To determine if you need to set your MANPATH environment variable:

1. **Request the `workshop` man page by typing:**

```
% man workshop
```

2. **Review the output, if any.**

   If the `workshop`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in this section for setting your MANPATH environment variable.

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the `/opt` directory. Contact your system administrator if your Sun WorkShop software is not installed in `/opt`.

The PATH and MANPATH variables should be set in your home `.cshrc` file if you are using the C shell or in your home `.profile` file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your PATH variable:

  ```
  /opt/SUNWspro/bin
  ```

- To access Sun WorkShop man pages with the `man` command, add the following to your MANPATH variable:

  ```
  /opt/SUNWspro/man
  ```

For more information about the PATH variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the MANPATH variable, see the `man`(1) man page. For more information about setting your PATH and MANPATH variables to access this release, see the *Sun WorkShop 6 Installation Guide* or your system administrator.

Related Interval References

The interval literature is large and growing. Interval applications exist in various substantive fields. However, most interval books and journal articles either contain these algorithms, or are written for interval analysts who are developing new interval algorithms. There is not yet a book titled “Introduction to Intervals.”

The Sun WorkShop 6 C++ compiler is not the only source of C++ support for intervals. Readers interested in other well known sources can refer to the following books:

- R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo, M. Rauch, *C-XSC Class Library for Extended Scientific Computing*. Springer, 1993.
- R. Hammer, M. Hocks, U. Kulisch, D. Ratz, *Numerical Toolbox for Verified Computing I, Basic Numerical Problems*. Springer, 1993.

For a list of technical reports that establish the foundation for the interval innovations implemented in class `interval`, see "References" on page 74. See the Interval Arithmetic Readme for the location of the online versions of these references.

Online Resources

Additional interval information is available at various web sites and email mailing lists. For a list of online resources, refer to the interval arithmetic readme.

Web Sites

A detailed bibliography and interval FAQ can be obtained online at the URLs listed in the interval arithmetic readme.

Email

To discuss interval arithmetic issues or ask questions about using interval arithmetic, a mailing list has been constructed. Anyone can send questions to this list. Refer to the interval arithmetic readme for instructions on how to subscribe to this mailing list.

To report a suspected interval error, send email to the following address:

`sun-dp-comments@Sun.COM`

Include the following text in the Subject line of the email message:

`WORKSHOP "6.0` *`mm/dd/yy`*`" Interval`

where *mm/dd/yy* is the month, day, and year of the message.

Code Examples

All code examples in this book are contained in the following directory:

`http://www.sun.com/forte/cplusplus/interval`

The name of each file is `ce`*n*`-`*m*`.cc`, where *n* is the chapter in which the example occurs and *m* is the number of the example. Additional interval examples are also provided in this directory.

Related Non-Interval Sun WorkShop 6 Documentation

For more information about this product, see the following sources. (The names of our development tools have changed from Sun WorkShop™ to Forte™ Developer products; you might see both product names used.)

Note – If your Sun WorkShop 6 update 1 software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- **Man pages and readmes**. This documentation describes the new features, performance enhancements, problems and workarounds, and software corrections in this Sun WorkShop 6 update 1 release.

  You can access these documents in HTML on your local system or network by pointing your browser to `file:/opt/SUNWspro/docs/index.html`.

- **The Sun WorkShop and Sun WorkShop TeamWare online help**. The online help has been updated for the new features in this Sun WorkShop 6 update 1 release.

  You can access the online help on your local system or network by pointing your browser to `file:/opt/SUNWspro/docs/index.html`. You can access the online help from the Help menu in the Sun WorkShop products.

- **What's New in Sun WorkShop 6 update 1.** This book describes the new features in this Sun WorkShop 6 update 1 release and in the Sun WorkShop 6 release.

  You can access this book on your local system or network by pointing your browser to `file:/opt/SUNWspro/docs/index.html`. You can also access it by pointing your browser to `http://docs.sun.com` and searching for the Forte Developer 6 update 1 collection.

- **Sun WorkShop 6 manuals**. These manuals were provided with Sun WorkShop 6. Information in the Sun WorkShop 6 update 1 man pages, readmes, and online help supersedes information in the Sun WorkShop 6 manuals.

  You can access the manuals on your local system or network by pointing your browser to the Sun WorkShop 6 update 1 Documentation Index (`file:/opt/SUNWspro/docs/index.html`). You can also access them by pointing your browser to `http://docs.sun.com` and searching for the Forte C, Forte C++, Forte for High Performance Computing, and Forte TeamWare products.

  The following Sun WorkShop manuals are *only* accessible on your local system or network (by pointing your browser to `file:/opt/SUNWspro/docs/index.html`) and *not* through `http://docs.sun.com`:

  - *Sun WorkShop Memory Monitor User's Manual*
  - *Standard C++ Class Library Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
  - *Sun Performance Library Reference*

- **Sun WorkShop 6 update 1 supplements**. The supplements provide more detailed information on some of the major new features in this Sun WorkShop 6 update 1 release.

  You can access the supplements by pointing your browser to `http://docs.sun.com` and searching for the Forte Developer 6 update 1 collection.

- **Sun WorkShop 6 update 1 Release Notes.** These notes provide installation-related and late-breaking information about this Sun WorkShop 6 update 1 release. Information in the release notes supersedes information in any of the other documentation.

  The release notes are available as a text file on the Forte Developer 6 update 1 CD at `/cdrom/devpro_v8n1_platform/release_notes.txt`. They are also available in HTML on the Forte Developer Products Hot News page by pointing your browser at `http://www.sun.com/forte/developer/hotnews.html`.

CHAPTER 1

Using the Interval Arithmetic Library

1.1 What Is Interval Arithmetic?

Interval arithmetic is a system for computing with intervals of numbers. Because interval arithmetic always produces intervals that contain the set of all possible result values, interval algorithms have been developed to perform surprisingly difficult computations. For more information on interval applications, see the interval arithmetic readme.

1.2 C++ Interval Support Goal: Implementation Quality

The goal of `interval` support in C++ is to stimulate development of commercial interval solver libraries and applications by providing program developers with:

- Quality interval code
- Narrow-width interval results
- Rapidly executing interval code
- An easy-to-use software development environment

Support and features are components of implementation quality. Throughout this book, various quality of implementation opportunities are described. Additional suggestions from users are welcome.

1.2.1 Quality Interval Code

As a consequence of evaluating any interval expression, a valid interval-supporting compiler must produce an interval that contains the set of all possible results. The requirement to contain the set of all possible results is called the containment constraint of interval arithmetic. The failure to satisfy the containment constraint is a containment failure. A silent failure (with no warning or documentation) to satisfy the interval containment constraint is a fatal error in any interval computing system. By satisfying this single constraint, intervals provide unprecedented computing quality.

Given the containment constraint is satisfied, implementation quality is determined by the location of a point in the two-dimensional plane whose axes are *runtime* and *interval width*. On both axes, small is better. How to trade runtime for interval width depends on the application. Both runtime and interval width are obvious measures of interval-system quality. Because interval width and runtime are always available, measuring the accuracy of both interval algorithms and implementation systems is no more difficult than measuring their speed.

The Sun WorkShop 6 tools for performance profiling can be used to tune interval programs. However, in C++, no interval-specific tools exist to help isolate where an algorithm may gain unnecessary interval width. Quality of implementation opportunities include adding additional interval-specific code development and debugging tools.

1.2.2 Narrow-Width Interval Results

All the normal language and compiler quality of implementation opportunities exist for intervals, including rapid execution and ease of use.

Valid interval implementation systems include a new additional quality of implementation opportunity: Minimize the width of computed intervals while always satisfying the containment constraint.

If an interval's width is as narrow as possible, it is said to be *sharp*. For a given floating-point precision, an interval result is sharp if its width is as narrow as possible.

The following the following statements apply to the width of intervals produced by the `interval` class:

- Individual intervals are sharp approximations of their external representation.
- Individual interval arithmetic functions produce sharp results.
- Mathematical functions usually produce sharp results.

1.2.3 Rapidly Executing Interval Code

By providing compiler optimization and hardware instruction support, `interval` operations are not necessarily slower than their floating-point counterparts. The following can be said about the speed of interval operators and mathematical functions:

- Arithmetic operations are reasonably fast.
- The speed of `interval<double>` mathematical functions is generally less than half the speed of their `double` counterparts. `interval<float>` math functions are provided, but are not tuned for speed (unlike their `interval<double>` counterparts). The `interval<long double>` mathematical functions are not provided in this release. However, other `interval<long double>` functions are supported.

1.2.4 Easy-to-Use Development Environment

The C++ `interval` class facilitates interval code development, testing, and execution.

Sun WorkShop 6 C++ includes the following interval extensions:

- `interval` template specializations for intervals using `float`, `double`, and `long double` scalar types.
- `interval` arithmetic operations and mathematical functions that form a closed mathematical system. (This means that valid results are produced for any possible operator-operand combination, including division by zero and other indeterminate forms involving zero and infinities.)
- Three types of interval relational functions:
  - Certainly
  - Possibly
  - Set
- `interval`-specific functions, such as `intersect` and `interval_hull.`
- `interval`-specific functions, such as `inf`, `sup`, and `wid.`
- `interval` input/output, including single-number input/output.

For examples and more information on these and other interval functions, see CODE EXAMPLE 2-8 on page 46 through CODE EXAMPLE 2-10 on page 53 and Section 2.9.4 “Functions That Accept Interval Arguments” on page 68.

Chapter 2 describes these and other interval features.

1.2.5 The C++ Interval Class Compilation Interface

The compilation interface consists of the following:

- A new value, `interval`, for the `-library` flag, which expands to the appropriate libraries.
- A new value, `interval`, for the `-staticlib` flag, which at present is ignored because only static libraries are currently supported.
- A new flag, `-xia`, which expands to `-fsimple=0 -ftrap=%none -fns=no -library=interval`. This flag is the same flag that is used with the Fortran compilers, though the expansion is different.

To use the C++ interval arithmetic features, add the following header file to the code.

```
#include <suninterval.h>
```

An example of compiling code using the `-xia` command-line option is shown here.

```
math% CC -o filename -xia filename.cc
```

The C++ interval library supports the following common C++ compilation modes:

- Compatibility mode (ARM) using `-compat`
- Standard mode (ISO) with the standard library, which is the default
- Standard mode with the traditional iostream library (`-library=iostream`)

See the *C++ Migration Guide*, the *C++ Programming Guide*, and the *C++ User's Guide* for more information on these modes.

The following sections describe the ways that these compilation modes affect compilation of applications using the interval library.

1.2.5.1 `namespace SUNW_interval`

In standard mode only, all interval types and symbols are defined within the `namespace SUNW_interval`. To write applications that compile in both standard mode and compatibility mode, use the following code.

```
#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif
```

1.2.5.2 Boolean Return Values

Some interval functions return boolean values. Because compatibility mode does not support boolean types by default, these functions are defined returning a type `interval_bool`, which is a typedef to an `int` (compatibility mode) or a `bool` (standard mode). Client code should use whatever type appropriate for boolean values and rely on the appropriate conversions from `interval_bool` to the client's boolean type. The library does not support explicit use of `-features=bool` or `-features=no%bool`.

1.2.5.3 Input and Output

The interval library requires the I/O mechanisms supplied in one of the three compilation modes listed in Section 1.2.5 "The C++ Interval Class Compilation Interface" on page 12. In particular, the flag `-library=iostream` must be specified on all compile and link commands if the application is using the standard mode with the traditional iostream library.

1.3 Writing Interval Code for C++

The examples in this section are designed to help new interval programmers to understand the basics and to quickly begin writing useful interval code. Modifying and experimenting with the examples is strongly recommended.

1.3.1 Hello Interval World

CODE EXAMPLE 1-1 is the interval equivalent of "hello world."

CODE EXAMPLE 1-1 Hello Interval World

```
math% cat ce1-1.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {

cout <<"[2,3]+[4,5]="
         << (interval<double>("[2,3]") +
             interval<double>("[4,5]"));
    cout << endl;
}

math% CC  -xia -o ce1-1 ce1-1.cc
math% ce1-1
[2,3]+[4,5]=[0.6000000000000000E+001,0.8000000000000000E+001]
```

CODE EXAMPLE 1-1 uses standard output streams to print the labeled sum of the intervals [2, 3] and [4, 5].

1.3.2 `interval` External Representations

The integer and floating-point numbers that can be represented in computers are referred to as internal machine representable numbers. These numbers are a subset of the entire set of extended (including -∞ and +∞) real numbers. To make the distinction, machine representable numbers are referred to as internal and any number as external. Let x be an external (decimal) number or an interval endpoint that can be read or written in C++. Such a number can be used to represent either an external interval or an endpoint. There are three displayable forms of an external interval:

- `[X_inf, X_sup]` represents the mathematical interval $[\underline{x}, \bar{x}]$
- `[X]` represents the degenerate mathematical interval $[x, x]$, or $[x]$
- `X` represents the non-degenerate mathematical interval $[x] + [-1,+1]_{uld}$ (unit in the last digit). This form is the single-number representation, in which the last decimal digit is used to construct an interval. See Section 1.3.4 "interval Input/Output" on page 16 and Section 2.8.2 "Single-Number Output" on page 61.

In this form, trailing zeros are significant. Thus `0.10` represents interval `[0.09, 0.11]`, `100E-1` represents interval `[9.9, 10.1]`, and `0.10000000` represents the interval `[0.099999999, 0.100000001]`.

A positive or negative infinite interval endpoint is input/output as a case-insensitive string `inf` or `infinity` prefixed with a minus sign or an optional plus sign.

The empty interval is input/output as the case-insensitive string `empty` enclosed in square brackets, `[...]`. The string, `"empty"`, can be preceded or followed by blank spaces.

See Section 2.4.1 "Arithmetic Operators +, –, *, /" on page 39, for more details.

Note – If an invalid interval such as `[2,1]` is converted to an internal interval, `[inf, inf]` is stored internally.

1.3.3 Interval Declaration and Initialization

The `interval` declaration statement performs the same functions for `interval` data items as the `double` and `int` declarations do for their respective data items.

CODE EXAMPLE 1-2 uses `interval` variables and initialization to perform the same operation as CODE EXAMPLE 1-1.

CODE EXAMPLE 1-2 Hello Interval World With `interval` Variables

```
math% cat ce1-2.cc

#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {

    interval<double> X("[2,3]");
    interval<double> Y("3");  // interval [2,4] is represented
    cout <<"[2,3]+[2,4]=" << X + Y;
    cout << endl;
}

math% CC  -xia -o ce1-2 ce1-2.cc
math% ce1-2
[2,3]+[2,4]=[0.40000000000000000E+001,0.70000000000000000E+001]
```

Variables `X` and `Y` are declared to be of type `interval<double>` variables and are initialized to [2, 3] and [2, 4], respectively. The standard output stream is used to print the labeled interval sum of `X` and `Y`.

Note – To facilitate code-example readability, all interval variables are shown as uppercase characters. Interval variables can be uppercase or lowercase in code.

1.3.4 `interval` Input/Output

Full support for reading and writing intervals is provided. Because reading and interactively entering interval data can be tedious, a *single-number* interval format is introduced. The single-number convention is that any number not contained in brackets is interpreted as an interval whose lower and upper bounds are constructed by subtracting and adding 1 unit to the last displayed digit.

Thus

2.345 = [2.344, 2.346],

2.34500 = [2.34499, 2.34501],

and

23 = [22, 24].

Symbolically,

$[2.34499, 2.34501] = 2.34500 + [-1, +1]_{uld}$

where $[-1, +1]_{uld}$ means that the interval [-1, +1] is added to the last digit of the preceding number. The subscript, uld, is a mnemonic for “unit in the last digit.”

To represent a degenerate interval, a single number can be enclosed in square brackets. For example,

[2.345] = [2.345, 2.345] = 2.345000000000.....

This convention is used both for input and constructing intervals out of an external character string representation. Thus, type **`[0.1]`** to indicate the input value is an exact decimal number, even though 0.1 is not machine representable.

For example, during input to a program, `[0.1,0.1]` = `[0.1]` represents the *point*, 0.1, while using single-number input/output, `0.1` represents the interval

$0.1 + [-1, +1]_{uld} = [0, 0.2]$.

In C++ the input conversion process constructs a sharp interval that contains the input decimal value. If the value is machine representable, the internal machine approximation is degenerate. If the value is not machine representable, an interval having width of 1-ulp (unit-in-the-last-place of the mantissa) is constructed.

Note – A uld and an ulp are different. A uld is a unit in the last displayed decimal digit of an external number. An ulp is the smallest possible increment or decrement that can be made to an internal machine number.

The simplest way to read and print `interval` data items is with standard stream input and output.

CODE EXAMPLE 1-3 is a simple tool to help users become familiar with interval arithmetic and single-number `interval` input/output using streams.

Note – The interval containment constraint requires that directed rounding be used during both input and output. With single-number input followed immediately by single-number output, a decimal digit of accuracy can appear to be lost. In fact, the width of the input interval is increased by at most 1-ulp, when the input value is not machine representable. See Section 1.3.5 “Single-Number Input/Output” on page 19 and CODE EXAMPLE 1-6 on page 21.

CODE EXAMPLE 1-3 Interval Input/Output

```
math% cat ce1-3.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X, Y;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<" X,Y=?";
    cin >>X >>Y;
```

CODE EXAMPLE 1-3 Interval Input/Output *(Continued)*

```
    for (;;){
       cout <<endl <<"For X =" <<X <<endl<<", and Y=" <<Y <<endl;

        cout <<"X+Y=" << (X+Y) <<endl;

        cout <<"X-Y=" << (X-Y) <<endl;

        cout <<"X*Y=" << (X*Y) <<endl;

        cout <<"X/Y=" << (X/Y) <<endl;

        cout <<"pow(X,Y)=" << pow(X,Y) <<endl;

        cout <<" X,Y=?";

        cin >>X>>Y;
    }
}

math% CC ce1-3.cc -xia -o ce1-3
math% ce1-3
Press Control/C to terminate!
 X,Y=?[1,2][3,4]

For X =[0.1000000000000000E+001,0.2000000000000000E+001]
, and Y=[0.3000000000000000E+001,0.4000000000000000E+001]
X+Y=[0.4000000000000000E+001,0.6000000000000000E+001]
X-Y=[-.3000000000000000E+001,-.1000000000000000E+001]
X*Y=[0.3000000000000000E+001,0.8000000000000000E+001]
X/Y=[0.2500000000000000E+000,0.6666666666666668E+000]
pow(X,Y)=[0.1000000000000000E+001,0.1600000000000000E+002]
 X,Y=?[1,2] -inf

For X =[0.1000000000000000E+001,0.2000000000000000E+001]
, and Y=[              -Infinity,-.1797693134862315E+309]
X+Y=[              -Infinity,-.1797693134862315E+309]
X-Y=[0.1797693134862315E+309,               Infinity]
X*Y=[              -Infinity,-.1797693134862315E+309]
X/Y=[-.1112536929253602E-307,0.0000000000000000E+000]
pow(X,Y)=[0.0000000000000000E+000,              Infinity]
 X,Y=? <Control-C>
```

1.3.5 Single-Number Input/Output

One of the most frustrating aspects of reading interval output is comparing interval infima and suprema to count the number of digits that agree. For example, CODE EXAMPLE 1-4 and CODE EXAMPLE 1-5 shows the interval output of a program that generates different random-width `interval` data.

Note – Only program output is shown in CODE EXAMPLE 1-4 and CODE EXAMPLE 1-5. The code that generates the output is included with the examples located in the `http://sun.com/forte/cplusplus/interval` directory.

CODE EXAMPLE 1-4 [inf, sup] Interval Output

```
math% a.out
Press Control/C to terminate!
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output: 5 4 0
x4=[0.14680409E-014,0.14976984E-014]
x4=[-.16254538E+039,-.15932665E+039]
x4=[0.14542469E-034,       Infinity]
x4=[0.28025969E-044,0.28025970E-044]
x4=[-.54349165E-034,-.54338293E-034]
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output: 5 8 0
x8=[0.8671171289369087E+049,0.8671176773501073E+049]
x8=[-.2405178593145946E-124,-.2403657905522831E-124]
x8=[-.8474166174941822E-255,-.8474166169582288E-255]
x8=[0.1084305636204888E+266,0.1084327322534477E+266]
x8=[0.3117107160903294E-298,0.3117107180617612E-298]
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output: 5 16 0
x16=[0.490699395899333384569390820255623 9E+0287,0.49069942693388
911617983641918 39492E+0287]
x16=[0.588687654519598638072992624236009 5E-0193,
0.588805403825433141256568331482775 3E-0193]
x16=[0.597200657326918243731116187669226 5E-0288,
0.597200657704621103311113367175830 3E-0288]
x16=[0.000000000000000000000000000000000E+0000,0.00000000000000
000000000000000000000E+0000]
x16=[-.916438095738104375452873031923729 9E+0143,
-.916432299692998919691619015996529 1E+0143]
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output:<Control-C>
```

Compare the output readability in CODE EXAMPLE 1-4 with CODE EXAMPLE 1-5.

CODE EXAMPLE 1-5 Single-Number Output

```
math% a.out
Press Control/C to terminate!
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output: 5 4 1
             0.15  E-014
            -0.16  E+039
[0.1454E-034,   Infinity]
             0.2802E-044
            -0.5434E-034
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output: 5 8 1
              0.86711E+049
             -0.240  E-124
             -0.84741E-255
              0.10843E+266
              0.31171E-298
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output: 5 16 1
                    0.4906994  E+287
                    0.588      E-193
                    0.597200657E-288
[                   0.000000000E+000]
                   -0.91643    E+143
Enter number of intervals, 4 - for float, 8 - for double, or
16 - for long double, and 1 - for single-number output:<Control-C>
```

In the single-number display format, trailing zeros are significant. See Section 2.8 “Input and Output” on page 60 for more information.

Intervals can always be entered and displayed using the traditional [*inf, sup*] display format. In addition, a single number in square brackets denotes a point. For example, on input, `[0.1]` is interpreted as the number 1/10. To guarantee containment, directed rounding is used to construct an internal approximation that is known to contain the number 1/10.

CODE EXAMPLE 1-6 Character Input With Internal Data Conversion

```
math% cat ce1-6.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    char BUFFER[128];
    cout << "Press Control/C to terminate!"<< endl;
    cout << "X=?";
    cin >>BUFFER;
    for(;;) {
        interval<double> X(BUFFER);
        cout << endl << "Your input was:" <<BUFFER << endl;
      cout << "Resulting stored interval is:" << endl << X << endl;
        cout << "Single number interval output is: ";
        single_number_output(X, cout);
        cout <<endl <<"X=?" ;
        cin >>BUFFER;
    }
}
math% CC -xia ce1-6.cc -o ce1-6
math% ce1-6
Press Control/C to terminate!
X=?1.37

Your input was:1.37
Resulting stored interval is:
[0.1359999999999999E+001,0.1380000000000001E+001]
Single number interval output is:              0.13   E+001
X=?1.444

Your input was:1.444
Resulting stored interval is:
[0.1442999999999999E+001,0.1445000000000001E+001]
Single number interval output is:              0.144  E+001
X=? <Control-C>
```

CODE EXAMPLE 1-6 notes:

- Single numbers in square brackets represent degenerate intervals.
- When a non-machine representable number is read using single-number input, conversion from decimal to binary (radix conversion) and the containment constraint force the number's interval width to be increased by 1-ulp (unit in the

last place of the mantissa). When this result is displayed using single-number output, it can appear that a decimal digit of accuracy has been lost. This is not so. To echo single-number interval inputs, use character input together with an interval constructor with a character string argument, as shown in CODE EXAMPLE 1-6 on page 21.

Note – The empty interval is supported in the `interval` class. The empty interval can be entered as `[empty]`. Infinite interval endpoints are also supported, as described in Section 1.3.2 “interval External Representations” on page 14.

1.3.6 Arithmetic Expressions

Writing arithmetic expressions that contain `interval` data items is simple and straightforward. Except for `interval`-specific functions and constructors, `interval` expressions look like floating-point arithmetic expressions, such as in CODE EXAMPLE 1-7.

CODE EXAMPLE 1-7 Simple `interval` Expression Example

```
math% cat ce1-7.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X1("[0.1]");
    interval <double> N(3);
    interval <double> A (5.0);
    X = X1 * A / N;
    cout << "[0.1]*[A]/[N]=" <<X <<endl;
}
math% CC -xia -o ce1-7 ce1-7.cc
math% ce1-7
[0.1]*[A]/[N]=[0.1666666666666666E+000,0.1666666666666668E+000]
```

Note – Not all mathematically equivalent `interval` expressions produce intervals having the same width. Additionally, it is often not possible to compute a sharp result by simply evaluating a single `interval` expression. In general, interval result width depends on the value of `interval` arguments and the form of the expression.

1.3.7 `interval`-Specific Functions

A variety of `interval`-specific functions are provided. See Section 2.9.4 “Functions That Accept Interval Arguments” on page 68. Use CODE EXAMPLE 1-8 to explore how specific `interval` functions behave.

CODE EXAMPLE 1-8 `interval`-Specific Functions

```
math% cat ce1-8.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<"X=?";
    cin >>X;
    for(;;){
        cout <<endl << "For X =" <<X << endl;
        cout <<"mid(X)=" << (mid(X)) <<endl;
        cout <<"mig(X)=" << (mig(X)) <<endl;
        cout <<"mag(X)=" << (mag(X)) <<endl;
        cout <<"wid(X)=" << (wid(X)) <<endl;
        cout <<"X=?";
        cin >>X;
    }
}
math% CC  -xia -o ce1-8 ce1-8.cc
math% ce1-8
Press Control/C to terminate!
X=? [1.23456,1.234567890]
For X =[0.1234559999999999E+001,0.1234567890000001E+001]
mid(X)=1.23456
mig(X)=1.23456
mag(X)=1.23457
wid(X)=7.89e-06
X=? [1,10]
For X =[0.1000000000000000E+001,0.1000000000000000E+002]
mid(X)=5.5
mig(X)=1
mag(X)=10
wid(X)=9
X=? <Control-C>
```

1.3.8 Interval Versions of Standard Functions

Use CODE EXAMPLE 1-9 to explore how some standard mathematical functions behave.

CODE EXAMPLE 1-9 Interval Versions of Mathematical Functions

```
math% cat ce1-9.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<"X=?";
    cin >>X;
    for (;;) {
        cout <<endl << "For X =" <<X << endl;

        cout <<"abs(X)=" << (fabs(X)) <<endl;

        cout <<"log(X)=" << (log(X)) <<endl;

        cout <<"sqrt(X)=" << (sqrt(X)) <<endl;

        cout <<"sin(X)=" << (sin(X)) <<endl;

        cout <<"acos(X)=" << (acos(X)) <<endl;

        cout <<"X=?";
        cin >>X;
    }
}
math% CC  -xia -o ce1-9 ce1-9.cc
math% ce1-9
Press Control/C to terminate!
X=? [1.1,1.2]
For X =[0.1099999999999999E+001,0.1200000000000001E+001]
abs(X)=[0.1099999999999999E+001,0.1200000000000001E+001]
log(X)=[0.9531017980432472E-001,0.1823215567939548E+000]
sqrt(X)=[0.1048808848170151E+001,0.1095445115010333E+001]
sin(X)=[0.8912073600614351E+000,0.9320390859672266E+000]
acos(X)=[EMPTY                                          ]
```

CODE EXAMPLE 1-9 Interval Versions of Mathematical Functions *(Continued)*

```
X=? [-0.5,0.5]
For X =[-.5000000000000000E+000,0.5000000000000000E+000]
abs(X)=[0.0000000000000000E+000,0.5000000000000000E+000]
log(X)=[              -Infinity,-.6931471805599452E+000]
sqrt(X)=[0.0000000000000000E+000,0.7071067811865476E+000]
sin(X)=[-.4794255386042031E+000,0.4794255386042031E+000]
acos(X)=[0.1047197551196597E+001,0.2094395102393196E+001]
X=? <Control-C>
```

1.4 Code Development Tools

Information on interval code development tools is available online. See the interval arithmetic readme for a list of interval web sites and other online resources.

To report a suspected interval error, send email to the following address:

`sun-dp-comments@Sun.COM`

Include the following text in the Subject line of the email message:

`WORKSHOP "6.0` *mm/dd/yy*`" Interval`

where *mm/dd/yy* is the month, day, and year of the message.

1.4.1 Debugging Support

In Sun WorkShop 6, interval data types are supported by `dbx` to the following extent:

- The values of individual `interval` variables can be printed using the `print` command.
- The value of all `interval` variables can be printed using the `dump` command.
- New values can be assigned to `interval` variables using the `assign` command.
- Printing the value of `interval` expressions is not supported.
- All generic functionality that is not data type specific should work.

For additional details on `dbx` functionality, see *Debugging a Program With* `dbx`.

CHAPTER 2

C++ Interval Arithmetic Library Reference

This chapter is a reference for the syntax and semantics of the `interval` arithmetic library implemented in Sun WorkShop 6 C++. The sections can be read in any order.

2.1 Character Set Notation

Throughout this document, unless explicitly stated otherwise, integer and floating-point constants mean *literal* constants. Literal constants are represented using strings, because class types do not support literal constants. Section 2.1.1 “String Representation of an Interval Constant (SRIC)” on page 28.

TABLE 2-1 shows the character set notation used for code and mathematics.

TABLE 2-1 Font Conventions

Character Set	Notation
C++ code	`interval<double> DX;`
Input to programs and commands	`Enter X: ? [2.3,2.4]`
Placeholders for constants in code	`[a,b]`
Scalar mathematics	$x(a + b) = xa + xb$
Interval mathematics	$X(A + B) \subseteq XA + XB$

Note – Pay close attention to font usage. Different fonts represent an interval’s exact, external mathematical value and an interval’s machine-representable, internal approximation.

2.1.1 String Representation of an Interval Constant (SRIC)

In C++, it is possible to define variables of a class type, but not literal constants. So that a literal interval constant can be represented, the C++ interval class uses a string to represent an interval constant. A string representation of an interval constant (SRIC) is a character string containing one of the following:

- A single integer or real decimal number enclosed in square brackets, `"[3.5]"`.
- A pair of integer or real decimal numbers separated by a comma and enclosed in square brackets, `"[3.5 E-10, 3.6 E-10]"`.
- A single integer or decimal number. This form is the single-number representation, in which the last decimal digit is used to construct an interval. See Section 1.3.2 "interval External Representations" on page 14.

Quotation marks delimit the string. If a degenerate interval is not machine representable, directed rounding is used to round the exact mathematical value to an internal machine representable interval known to satisfy the containment constraint.

A SRIC, such as `"[0.1]"` or `"[0.1,0.2]"`, is associated with the two values: its external value and its internal approximation. The numerical value of a SRIC is its internal approximation. The external value of a SRIC is always explicitly labelled as such, by using the notation ev(SRIC). For example, the SRIC `"[1, 2]"` and its external value ev(`"[1, 2]"`) are both equal to the mathematical value [1, 2]. However, while ev(`"[0.1, 0.2]"`) = [0.1, 0.2], `interval<double>("[0.1, 0.2]")` is only an internal machine approximation containing [0.1, 0.2], because the numbers 0.1 and 0.2 are not machine representable.

Like any mathematical constant, the external value of a SRIC is invariant.

Because intervals are opaque, there is no language requirement to use any particular interval storage format to save the information needed to internally approximate an interval. Functions are provided to access the infimum and supremum of an interval. In a SRIC containing two interval endpoints, the first number is the infimum or lower bound, and the second is the supremum or upper bound.

If a SRIC contains only one integer or real number in square brackets, the represented interval is degenerate, with equal infimum and supremum. In this case, an internal interval approximation is constructed that is guaranteed to contain the SRIC's single decimal external value. If a SRIC contains only one integer or real number *without* square brackets, single number conversion is used. See Section 2.8.1 "Input" on page 60.

A valid interval must have an infimum that is less than or equal to its supremum. Similarly, a SRIC must also have an infimum that is less than or equal to its supremum. For example, the following code fragment must evaluate to *true*:

```
inf(interval<double>("[0.1]") <= sup(interval<double>("[0.1]"))
```

CODE EXAMPLE 2-1 contains examples of valid and invalid SRICs.

For additional information regarding SRICs, see the supplementary paper [4] cited in Section 2.11 “References” on page 74.

CODE EXAMPLE 2-1 Valid and Invalid `interval` External Representations

```
math% cat ce2-1.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X1("[1,2]");
    interval <double> X2("[1]");
    interval <double> X3("1");
    interval <double> X4("[0.1]");
    interval <double> X5("0.1");
    interval <double> X6("0.10");
    interval <double> X7("0.100");
    interval <double> X8("[2,1]");
    cout << "X1=" << X1 << endl;
    cout << "X2=" << X2 << endl;
    cout << "X3=" << X3 << endl;
    cout << "X4=" << X4 << endl;
    cout << "X5=" << X5 << endl;
    cout << "X6=" << X6 << endl;
    cout << "X7=" << X7 << endl;
    cout << "X8=" << X8 << endl;

}
math% CC -xia -o ce2-1 ce2-1.cc
math% ce2-1
X1=[0.1000000000000000E+001,0.2000000000000000E+001]
X2=[0.1000000000000000E+001,0.1000000000000000E+001]
X3=[0.0000000000000000E+000,0.2000000000000000E+001]
X4=[0.9999999999999999E-001,0.1000000000000001E+000]
X5=[0.0000000000000000E+000,0.2000000000000001E+000]
X6=[0.8999999999999999E-001,0.1100000000000001E+000]
X7=[0.9899999999999999E-001,0.1010000000000001E+000]
X8=[              -Infinity,               Infinity]
```

Constructing an interval approximation from a SRIC is an inefficient operation that should be avoided, if possible. In CODE EXAMPLE 2-2, the `interval<double>` constant `Y` is constructed only once at the start of the program, and then its internal representation is used thereafter.

CODE EXAMPLE 2-2 Efficient Use of the String-to-Interval Constructor

```
math% cat ce2-2.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

const interval<double> Y("[0.1]");
const int limit = 100000;

int main()
{
   interval<double>  RESULT(0.0);
   clock_t t1= clock();
   if(t1==clock_t(-1)){cerr<< "sorry, no clock\n"; exit(1);}

   for (int i = 0; i < limit; i++){
            RESULT += Y;
   }
   clock_t t2= clock();
  if(t2==clock_t(-1)){cerr<< "sorry, clock overflow\n"; exit(2);}
   cout << "efficient loop took " <<
           double(t2-t1)/CLOCKS_PER_SEC << " seconds" << endl;
   cout << "result" << RESULT << endl ;
   t1= clock();
  if(t1==clock_t(-1)){cerr<< "sorry, clock overflow\n"; exit(2);}
   for (int i = 0; i < limit; i++){
            RESULT += interval<double>("[0.1]");
   }
   t2= clock();
  if(t2==clock_t(-1)){cerr<< "sorry, clock overflow\n"; exit(2);}
   cout << "inefficient loop took " <<
           double(t2-t1)/CLOCKS_PER_SEC << " seconds" << endl;
    cout << "result" << RESULT << endl ;
}
```

CODE EXAMPLE 2-2 Efficient Use of the String-to-Interval Constructor *(Continued)*

```
math% CC -xia ce2-2.cc -o ce2-2
math% ce2-2
efficient loop took 0.16 seconds
result[0.9999999999947978E+004,0.1000000000003054E+005]
inefficient loop took 5.59 seconds
result[0.1999999999980245E+005,0.2000000000013270E+005]
```

2.1.2 Internal Approximation

The internal approximation of a floating-point constant does not necessarily equal the constant's external value. For example, because the decimal number 0.1 is not a member of the set of binary floating-point numbers, this value can only be *approximated* by a binary floating-point number that is close to 0.1. For floating-point data items, the approximation accuracy is unspecified in the C++ standard. For `interval` data items, a pair of floating-point values is used that is known to contain the set of mathematical values defined by the decimal numbers used to symbolically represent an `interval` constant. For example, the mathematical interval [0.1, 0.2] is represented by a string "`[0.1,0.2]`".

Just as there is no C++ language requirement to accurately approximate floating-point constants, there is also no language requirement to approximate an interval's external value with a narrow width `interval` internal representation. There is a requirement for an `interval` internal representation to *contain* its external value.

ev(`inf(interval<double>("[0.1,0.2]"))`) ≤
inf(ev("`[0.1,0.2]`")) = inf([0.1, 0.2])

and

sup("[0.1, 0.2]") = sup(ev("`[0.1,0.2]`")) ≤
ev(`sup(interval<double>("[0.1,0.2]"))`)

Note – The arguments of ev( ) are always code expressions that produce mathematical values. The use of different fonts for code expressions and mathematical values is designed to make this distinction clear.

C++ `interval` internal representations are sharp. This is a quality of implementation feature.

2.2 `interval` Constructor

The following interval constructors are supported:

```
explicit interval( const char* ) ;
explicit interval( const interval<float>& ) ;
explicit interval( const interval<double>& ) ;
explicit interval( const interval<long double>& ) ;
explicit interval( int ) ;
explicit interval( long long ) ;
explicit interval( float ) ;
explicit interval( double ) ;
explicit interval( long double ) ;
interval( int, int ) ;
interval( long long, long long ) ;
interval( float, float ) ;
interval( double, double ) ;
interval( long double, long double ) ;
```

Only the interval constructor with interval arguments

```
interval( const interval<float>& ) ;
interval( const interval<double>& )
interval( const interval<long double>& ) ;
```

guarantees containment. In this case the argument interval is rounded outward, if necessary.

The interval constructor with non-interval arguments returns `[-inf,inf]` if either the second argument is less then the first, or if either argument is not a mathematical real number, such as when one or both arguments is a `NaN`.

Interval constructors with floating-point or integer arguments might not return an interval that contains the external value of constant arguments.

For example, use `interval<double>("[1.1,1.3]")` to sharply contain the mathematical interval [1.1, 1.3]. However, `interval<double>(1.1,1.3)` might not contain [1.1, 1.3], because the internal values of floating-point literal constants are approximated with unknown accuracy.

CODE EXAMPLE 2-3 `interval` Constructor With Floating-Point Arguments

```
math% cat ce2-3.cc
#include <limits.h>
#include <strings.h>
#include <sunmath.h>
#include <stack>
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main()
{
   //Compute 0.7-0.1-0.2-0.3-0.1 == 0.0

   interval<double>  correct_result;
   const interval<double> x1("[0.1]"),
   x2("[0.2]"),x3("[0.3]"),x7("[0.7]");

    cout << "Exact result:" <<  0.0 << endl ;

    cout << "Incorrect evaluation:" <<
   interval<double>(0.7-0.1-0.2-0.3-0.1, 0.7-0.1-0.2-0.3-0.1) <<
   endl ;

    correct_result = x7-x1-x2-x3-x1;

    cout << "Correct evaluation:" <<  correct_result << endl ;

}
math% CC -xia -o ce2-3 ce2-3.cc
math% ce2-3.cc
Exact result:0
Incorrect evaluation:
[-.2775557561562892E-016,-.2775557561562891E-016]
Correct evaluation:
[-.1942890293094024E-015,0.1526556658859591E-015]
```

The result value of an interval constructor is always a valid interval.

The `interval_hull` function can be used with an interval constructor to construct an interval containing two floating-point numbers, as shown in CODE EXAMPLE 2-4.

CODE EXAMPLE 2-4 Using the `interval_hull` Function With Interval Constructor

```
math% cat ce2-4.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <float> X;
    long double a,b;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<" a,b =?";
    cin >>a >>b;
    for(;;){
        cout <<endl << "For a =" << a << ", and b =" <<b<< endl;
             X = interval <float>(
                     interval_hull(interval<long double>(a),
                                   interval<long double>(b)));
        if(in(a,X) && in(b,X)){
                cout << "Check" << endl ;
                cout << "X=" << X << endl ;
           }
        cout <<" a,b =?";
        cin >>a >>b;
    }
}
math% CC -xia ce2-4.cc -o ce2-4
math% ce2-4
Press Control/C to terminate!
 a,b =?1.0e+400 -0.1

For a =1e+400, and b =-0.1
Check
X=[-.10000001E+000,       Infinity]
 a,b =? <Control-C>
```

2.2.1 `interval` Constructor Examples

The three examples in this section illustrate how to use the `interval` constructor to perform conversions from floating-point to `interval`-type data items.
CODE EXAMPLE 2-5 shows that floating-point expression arguments of the `interval` constructor are evaluated using floating-point arithmetic.

CODE EXAMPLE 2-5 `interval` Conversion

```
math% cat ce2-5.cc

#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <float> X, Y;
    interval <double> DX, DY;
    float R = 0.1f, S = 0.2f, T = 0.3f;
    double R8 = 0.1, T1, T2;

    Y = interval <float>(R,R);
    X = interval <float>(0.1f);          //note 1
    if (X == Y)
        cout <<"Check1"<< endl;
    X = interval <float>(0.1f, 0.1f);
    if (X == Y)
        cout <<"Check2"<< endl;
    T1 = R + S;
    T2 = T + R8;
    DY = interval <double>(T1, T2);
    DX = interval <double>(R+S, T+R8);   //note 2
    if (DX == DY)
        cout <<"Check3"<< endl;
    DX = interval <double>(Y);          //note 3
    if (ceq(DX,interval <double>(0.1f, 0.1f)))
        cout <<"Check4"<< endl;
}
math% CC  -xia -o ce2-5 ce2-5.cc
math% ce2-5
Check1
Check2
Check3
Check4
```

CODE EXAMPLE 2-5 notes:

- **Note 1.** Interval `X` is assigned a degenerate interval with both endpoints equal to the internal representation of the real constant 0.1.
- **Note 2.** Interval `DX` is assigned an interval with left and right endpoints equal to the result of floating-point expressions `R+S` and `T+R8` respectively.
- **Note 3.** Interval `Y` is converted to a containing `interval<double>`.

CODE EXAMPLE 2-6 shows how the `interval` constructor can be used to create the smallest possible interval, `Y`, such that the endpoints of `Y` are *not* elements of a given interval, `X`.

CODE EXAMPLE 2-6 Creating a Narrow Interval That Contains a Given Real Number

```
math% cat ce2-6.cc
#include <suninterval.h>
#include <values.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif


int main() {
   interval <double> X("[10.E-10,11.E-10]");
   interval <double> Y;
   Y = interval<double>(-MINDOUBLE, MINDOUBLE) + X;
   cout << "X is " <<
   ((!in_interior(X,Y))? "not": "")<< "in interior of Y" <<endl;
}
math% CC ce2-6.cc -o ce2-6 -xia
math% ce2-6
X is  in interior of Y
```

Given an interval `X`, a sharp interval `Y` satisfying the condition `in_interior(X,Y)` is constructed. For information on the interior set relation, Section 2.6.3 “Interior: in_interior(X,Y)” on page 50.

CODE EXAMPLE 2-7 on page 37 illustrates when the `interval` constructor returns the interval [`-inf`, `inf`] and [`max_float`, `inf`].

CODE EXAMPLE 2-7 `interval(NaN)`

```
math% cat ce2-7.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
   interval <double> DX;
   float R=0.0, S=0.0, T;
   T = R/S;                                  //note  1
   cout<< T <<endl;
   cout<< interval<double>(T,S)<<endl;   //note  2
   cout<< interval<double>(T,T)<<endl;
   cout<< interval<double>(2.,1.)<<endl; //note  3
   cout<< interval<double>(1./R)<<endl;  //note  4
}
math% CC  -xia -o ce2-7 ce2-7.cc
math% ce2-7
NaN
[              -Infinity,               Infinity]
[              -Infinity,               Infinity]
[              -Infinity,               Infinity]
[0.1797693134862315E+309,               Infinity]
```

CODE EXAMPLE 2-7 notes:

- **Note 1.** Variable `T` is assigned a `NaN` value.
- **Note 2.** Because one of the arguments of the interval constructor is a NaN, the result is the interval `[-inf, inf]`.
- **Note 3.** The interval `[-inf, inf]` is constructed instead of an invalid interval [2,1].
- **Note 4.** The interval [`max_float`, `inf`] is constructed, which contains the interval [`inf`, `inf`]. See the supplementary paper [8] cited in Section 2.11 “References” on page 74 for a discussion of the chosen intervals to represent internally.

2.3 `interval` Arithmetic Expressions

`interval` arithmetic expressions are constructed from the same arithmetic operators as other numerical data types. The fundamental difference between `interval` and non-`interval` (point) expressions is that the result of any possible `interval` expression is a valid `interval` that satisfies the containment constraint of interval arithmetic. In contrast, point expression results can be any approximate value.

2.4 Operators and Functions

TABLE 2-2 lists the operators and functions that can be used with intervals. InTABLE 2-2, `X` and `Y` are intervals.

TABLE 2-2 Operators and Functions

Operator	Operation	Expression	Meaning
*	Multiplication	`X*Y`	Multiply `X` and `Y`
/	Division	`X/Y`	Divide `X` by `Y`
+	Addition	`X+Y`	Add `X` and `Y`
+	Identity	`+X`	Same as `X` (without a sign)
-	Subtraction	`X-Y`	Subtract `Y` from `X`
-	Numeric Negation	`-X`	Negate `X`
Function			**Meaning**
`interval_hull(X,Y)`			Interval hull of `X` and `Y`
`intersect(X,Y)`			Intersect `X` and `Y`
`pow(X,Y)`			Power function

Some `interval`-specific functions have no point analogs. These can be grouped into three categories: set, certainly, and possibly, as shown in TABLE 2-3. A number of unique set-operators have no certainly or possibly analogs.

TABLE 2-3 `interval` Relational Functions and Operators

Operators	==	!=				
Set Relational Functions	`superset(X,Y)`	`proper_superset(X,Y)`				
	`subset(X,Y)`	`proper_subset(X,Y)`				
	`in_interior(X,Y)`	`disjoint(X,Y)`				
	`in(r,Y)`					
	`seq(X,Y)`	`sne(X,Y)`	`slt(X,Y)`	`sle(X,Y)`	`sgt(X,Y)`	`sge(X,Y)`
Certainly Relational Functions	`ceq(X,Y)`	`cne(X,Y)`	`clt(X,Y)`	`cle(X,Y)`	`cgt(X,Y)`	`cge(X,Y)`
Possibly Relational Functions	`peq(X,Y)`	`pne(X,Y)`	`plt(X,Y)`	`ple(X,Y)`	`pgt(X,Y)`	`pge(X,Y)`

Except for the `in` function, `interval` relational functions can only be applied to two `interval` operands with the same type.

The first argument of the `in` function is of any integer or floating-point type. The second argument can have any interval type.

All the interval relational functions and operators return an `interval_bool`-type result.

2.4.1 Arithmetic Operators +, –, *, /

Formulas for computing the endpoints of interval arithmetic operations on finite floating-point intervals are motivated by the requirement to produce the narrowest interval that is guaranteed to contain the set of all possible point results. Ramon Moore independently developed these formulas and more importantly, was the first to develop the analysis needed to apply interval arithmetic. For more information, see *Interval Analysis* by R. Moore (Prentice-Hall, 1966).

The set of all possible values was originally defined by performing the operation in question on any element of the operand intervals. Therefore, given finite intervals, [a, b] and [c, d], with $\text{op} \in \{+, -, \times, \div\}$,

$$[a, b] \text{ op } [c, d] \supseteq \{x \text{ op } y \mid x \in [a, b] \text{ and } y \in [c, d]\},$$

with division by zero being excluded. Implementation formulas, or their logical equivalent, are:

$$[a, b] + [c, d] = [a + c\ ,\ b + d]$$

$$[a, b] - [c, d] = [a - d\ ,\ b - c]$$

$$[a, b] \times [c, d] = [\min(a \times c, a \times d, b \times c, b \times d)\ ,\ \max(a \times c, a \times d, b \times c, b \times d)]$$

$$[a, b] / [c, d] = \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right], \text{ if } 0 \notin [c, d]$$

Directed rounding is used when computing with finite precision arithmetic to guarantee the set of all possible values is contained in the resulting interval.

The set of values that any interval result must contain is called the containment set of the operation or expression.

To include extended intervals (with infinite endpoints) and division by zero, containment sets cannot directly depend on the value of arithmetic operations on real values. For extended intervals, containment sets are required for operations on points that are normally undefined. Undefined operations include the indeterminate forms $1 \div 0$, $0 \times \infty$, $0 \div 0$, and $\infty \div \infty$.

The containment-set closure identity solves the problem of identifying the value of containment sets of expressions at singular and indeterminate points. The identity states that containment sets are closures. The closure of a function at a point on the boundary of its domain includes all limit or accumulation points. For details, see the Glossary and the supplementary papers [1], [3], [10], and [11] cited in Section 2.11 “References” on page 74.

Symbolically, $\text{cset}(x \text{ op } y, \{(x_0, y_0)\}) = \{x_0\}\ \overline{\text{op}}\ \{y_0\}$, where $\overline{\text{op}}$ denotes the closure of the operation, op, and $\{x_0\}$ denotes the singleton set with only one element, x_0. The subscript 0 is used to symbolically distinguish a particular value, x_0, of the variable, x, from the variable itself. For example, with $x_0 = 1$, op = $\div$, and $y_0 = 0$, $x_0 \div y_0$ is undefined, but the closure, $\{1\}\ \bar{\div}\ \{0\} = \{-\infty, +\infty\}$.

This result is obtained using the sequences

$$\{y_j\} = \left\{\frac{-1}{j}\right\} \text{ or } \{y_j\} = \left\{\frac{1}{j}\right\},$$

both of whose limits are

$$\lim_{j \to \infty} y_j = 0$$

Using the two sequences, $\{y_j\}$, above, the closure of the division operator at $x_0 = 1$ and $y_0 = 0$ is:

$$\begin{aligned} 1 \bar{\div} 0 &= \lim_{j \to \infty} \frac{1}{y_j} \\ &= \lim_{j \to \infty} j \text{ or } \lim_{j \to \infty} (-j) \\ &= \{-\infty, +\infty\} \end{aligned}$$

The following tables, TABLE 2-4 through TABLE 2-8, display containment sets for the four basic arithmetic operations, wherein $\Re^* = [-\infty, +\infty]$, the entire set of extended real numbers, or $\Re^* = \Re \cup [-\infty, +\infty]$.

TABLE 2-4 Containment Set for Addition: cset($x + y$, $\{(x_0, y_0)\}$)

cset(x + y, {(x_0, y_0)})	**{-∞}**	**{real: y_0}**	**{+∞}**
{-∞}	{-∞}	{-∞}	$\Re^*$
{real: x_0}	{-∞}	$\{x_0 + y_0\}$	{+∞}
{+∞}	$\Re^*$	{+∞}	{+∞}

TABLE 2-5 Containment Set for Subtraction: cset($x - y$, $\{(x_0, y_0)\}$)

cset(x − y, {(x_0, y_0)})	**{-∞}**	**{real: y_0}**	**{+∞}**
{-∞}	$\Re^*$	{-∞}	{-∞}
{real: x_0}	{+∞}	$\{x_0 - y_0\}$	{-∞}
{+∞}	{+∞}	{+∞}	$\Re^*$

TABLE 2-6 Containment Set for Multiplication: cset($x \times y$, $\{(x_0, y_0)\}$)

cset(x × y, {(x_0, y_0)})	**{-∞}**	**{real: $y_0 < 0$}**	**{0}**	**{real: $y_0 > 0$}**	**{+∞}**
{-∞}	{+∞}	{+∞}	$\Re^*$	{-∞}	{-∞}
{real: $x_0 < 0$}	{+∞}	$\{x \times y\}$	{0}	$\{x \times y\}$	{-∞}
{0}	$\Re^*$	{0}	{0}	{0}	$\Re^*$
{real: $x_0 > 0$}	{-∞}	$x \times y$	{0}	$x \times y$	{+∞}
{+∞}	{-∞}	{-∞}	$\Re^*$	{+∞}	{+∞}

TABLE 2-7 Containment Set for Division: cset($x \div y$, $\{(x_0, y_0)\}$)

cset(x ÷ y, {(x_0, y_0)})	**{-∞}**	**{real: $y_0 < 0$}**	**{0}**	**{real: $y_0 > 0$}**	**{+∞}**
{-∞}	[0, +∞]	{+∞}	{-∞, +∞}	{-∞}	[-∞, 0]
{real: $x_0 \neq 0$}	{0}	$\{x \div y\}$	{-∞, +∞}	$\{x \div y\}$	{0}
{0}	{0}	{0}	$\Re^*$	{0}	{0}
{+∞}	[-∞, 0]	{-∞}	{-∞, +∞}	{+∞}	[0, +∞]

All inputs in the tables are shown as singletons. Results are shown as singletons, sets, or intervals. To avoid ambiguity, customary notation, such as $(-\infty) + (+\infty) = -\infty$, $(-\infty) + y = -\infty$, and $(-\infty) + (+\infty) = \Re^*$, is not used. These tables show the results for singleton-set inputs to each operation. Results for general set (or interval) inputs are the union of the results of the single-point results as they range over the input sets (or intervals).

In one case, division by zero, the result is not an interval, but the set, $\{-\infty, +\infty\}$. In this case, the narrowest interval in the current system that does not violate the containment constraint of interval arithmetic is the interval $[-\infty, +\infty] = \Re^*$.

Sign changes produce the expected results.

To incorporate these results into the formulas for computing interval endpoints, it is only necessary to identify the desired endpoint, which is also encoded in the rounding direction. Using $\downarrow$ to denote rounding down (towards -∞) and $\uparrow$ to denote rounding up (towards +∞),

$\downarrow (+\infty) \div (+\infty) = 0$ and $\uparrow (+\infty) \div (+\infty) = +\infty$.

$\downarrow 0 \times (+\infty) = -\infty$ and $\uparrow 0 \times (+\infty) = +\infty$.

Similarly, because $\text{hull}(\{-\infty, +\infty\}) = [-\infty, +\infty]$,

$\downarrow x \div 0 = -\infty$ and $\uparrow x \div 0 = +\infty$.

Finally, the empty interval is represented in C++ by the character string `[empty]` and has the same properties as the empty set, denoted $\varnothing$ in the algebra of sets. Any arithmetic operation on an empty interval produces an empty interval result. For additional information regarding the use of empty intervals, see the supplementary papers [6] and [7] cited in Section 2.11 “References” on page 74.

Using these results, C++ implements the closed interval system. The system is closed because all arithmetic operations and functions always produce valid interval results. See the supplementary papers [2] and [8] cited in Section 2.11 “References” on page 74.

2.4.2 Power Function pow(X,n) and pow(X,Y)

The power function can be used with integer or continuous exponents. With a continuous exponent, the power function has indeterminate forms, similar to the four arithmetic operators.

In the integer exponents case, the set of all values that an enclosure of X^n must contain is $\text{cset}(x^n,\{x\}) = \{z \mid z \in \text{cset}(x^n,\{x\}) \text{ and } x \in X\}$.

Monotonicity can be used to construct a sharp interval enclosure of the integer power function. When $n = 0$, $\text{cset}(x_n, \{x_0\}) = 1$ for all $x \in [-\infty, +\infty]$, and `pow(interval<double>("[empty]"),n)` is empty for all `n`.

In the continuous exponents case, the set of all values that an interval enclosure of X^Y must contain is

$$\text{cset}(\exp(Y\ln(X)), \{(Y_0, X_0)\}) = \{z \mid z \in \text{cset}(\exp(y\ln(x)), \{(y_0, x_0)\}), y \in Y_0, x \in X_0\}$$

where $\exp(y\ln(x)), \{(Y_0, X_0)\}$ is the containment set of the expression $\exp(y\ln(x))$. The function exp(y ln (x)) makes explicit that only values of $x \geq 0$ need be considered, and is consistent with the definition of `pow(x,y)` with floating-point arguments in C++.

The result is empty if either `interval` argument is empty, or if `sup(X)` < 0.

TABLE 2-8 displays the containment sets for all the singularities and indeterminate forms of cset(exp($y$ln(x)), {(y_0, x_0)}).

TABLE 2-8 cset(exp($y$ln(x)), {(y_0, x_0)})

x_0	y_0	**cset(exp($y$ln(x)), {(y_0, x_0)})**
0	$y_0 < 0$	$+\infty$
1	$-\infty$	$[0,+\infty]$
1	$+\infty$	$[0,+\infty]$
$+\infty$	0	$[0,+\infty]$
0	0	$[0,+\infty]$

The results in TABLE 2-8 can be obtained in two ways:

- Directly compute the closure of the composite expression exp($y$ln(x)) for the values of x_0 and y_0 for which the expression is undefined.
- Use the containment-set evaluation theorem to bound the set of values in a containment set.

For most compositions, the second option is much easier. If sufficient conditions are satisfied, the closure of a composition can be computed from the composition of its closures. That is, the closure of each sub-expression can be used to compute the closure of the entire expression. In the present case,

$$\text{cset}(\exp(y\ln(x)), \{x_0, y_0\}) = \overline{\exp}(\{y_0\} \overline{\times} \overline{\ln}(\{x_0\})).$$

It is always the case that

$$\text{cset}(\exp(y\ln(x)), \{x_0, y_0\}) \subseteq \overline{\exp}(\{y_0\} \overline{\times} \overline{\ln}(\{x_0\})).$$

Note that this is exactly how interval arithmetic operates on intervals. The needed closures of the ln and exp functions are:

$$\begin{aligned}
\overline{\ln}\{0\} &= \{-\infty\} \\
\overline{\ln}(\{+\infty\}) &= \{+\infty\} \\
\overline{\exp}(\{-\infty\}) &= \{0\} \\
\overline{\exp}(\{+\infty\}) &= \{+\infty\}
\end{aligned}$$

A necessary condition for closure-composition equality is that the expression must be a *single-use expression* (or SUE), which means that each independent variable can appear only once in the expression.

In the present case, the expression is clearly a SUE.

The entries in TABLE 2-8 follow directly from using the containment set of the basic multiply operation in TABLE 2-6 on the closures of the ln and exp functions. For example, with $x_0 = 1$ and $y_0 = -\infty$, $\ln(x_0) = 0$. For the closure of multiplication on the values $-\infty$ and 0 in TABLE 2-6 on page 41, the result is $[-\infty, +\infty]$. Finally, $\exp([-\infty, +\infty]) = [0, +\infty]$, the second entry in TABLE 2-8. Remaining entries are obtained using the same steps. These same results are obtained from the direct derivation of the containment set of $\exp(y\ln(x))$. At this time, sufficient conditions for closure-composition equality of any expression have not been identified. Nevertheless, the following statements apply:

- The containment-set evaluation theorem guarantees that a containment failure can never result from computing a composition of closures instead of a closure.
- An expression must be a SUE for closure-composition equality to be true.

2.5 Set Theoretic Functions

C++ supports the following set theoretic functions for determining the interval hull and intersection of two intervals.

CODE EXAMPLE 2-8 on page 46 demonstrates the use of the `interval`-specific functions listed in TABLE 2-9.

TABLE 2-9 Interval-Specific Functions

Function	Name	Mathematical Symbol
`interval_hull(X,Y)`	Interval Hull	$\underline{\cup}$
`intersect(X,Y)`	Intersection	$\cap$
`disjoint(X,Y)`	Disjoint	$A \cap B = \emptyset$
`in(r,Y)`	Element	$\in$
`in_interior(X,Y)`	Interior	See Section 2.6.3 “Interior: in_interior(X,Y)” on page 50.
`proper_subset(X,Y)`	Proper Subset	$\subset$
`proper_superset(X,Y)`	Proper Superset	$\supset$
`subset(X,Y)`	Subset	$\subseteq$
`superset(X,Y)`	Superset	$\supseteq$

CODE EXAMPLE 2-8 Set Operators

```
math% cat ce2-8.cc

#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X, Y;
    double R;
    R = 1.5;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<"X,Y=?";
    cin >>X >>Y;
    for(;;){
      cout <<endl << "For X =" <<X <<",  and" << endl << "Y =" <<Y<<
        endl;

        cout <<"interval_hull(X,Y)=" << endl <<
                interval_hull(X,Y) <<endl;

        cout <<"intersect(X,Y)="<< intersect(X,Y) <<endl;

       cout <<"disjoint(X,Y)=" << (disjoint(X,Y) ?"T":"F") <<endl;

        cout <<"in(R,Y)=" << (in(R,Y) ?"T":"F") <<endl;

        cout <<"in_interior(X,Y)=" <<
                (in_interior(X,Y) ?"T":"F") <<endl;

        cout <<"proper_subset(X,Y)=" <<
                (proper_subset(X,Y) ?"T":"F") <<endl;

        cout <<"proper_superset(X,Y)=" <<
                (proper_superset(X,Y) ?"T":"F") <<endl;

        cout <<"subset(X,Y)=" << (subset(X,Y) ?"T":"F") <<endl;

       cout <<"superset(X,Y)=" << (superset(X,Y) ?"T":"F") <<endl;

        cout <<"X,Y=?";
        cin >>X>>Y;
    }
}
```

CODE EXAMPLE 2-8 Set Operators *(Continued)*

```
math%CC  -xia -o ce2-8 ce2-8.cc
math%ce2-8
Press Control/C to terminate!
X,Y=? [1] [2]
For X =[0.1000000000000000E+001,0.1000000000000000E+001], and Y
=[0.2000000000000000E+001,0.2000000000000000E+001]
interval_hull(X,Y)=[0.1000000000000000E+001,0.2000000000000000E+
001]
intersect(X,Y)=[EMPTY                                           ]
disjoint(X,Y)=T
in(R,Y)=F
in_interior(X,Y)=F
proper_subset(X,Y)=F
proper_superset(X,Y)=F
subset(X,Y)=F
superset(X,Y)=F
X,Y=? [1,2] [1,3]
For X =[0.1000000000000000E+001,0.2000000000000000E+001], and Y
=[0.1000000000000000E+001,0.3000000000000000E+001]
interval_hull(X,Y)=[0.1000000000000000E+001,0.3000000000000000E+
001]
intersect(X,Y)=[0.1000000000000000E+001,0.2000000000000000E+001]
disjoint(X,Y)=F
in(R,Y)=T
in_interior(X,Y)=F
proper_subset(X,Y)=T
proper_superset(X,Y)=F
subset(X,Y)=T
superset(X,Y)=F
X,Y=? <Control-C>
```

2.5.1 Hull: $X \underline{\cup} Y$ or `interval_hull(X,Y)`

Description: Interval hull of two intervals. The interval hull is the smallest interval that contains all the elements of the operand intervals.

Mathematical definitions:

$$\texttt{interval\_hul(X,Y)} \equiv [\inf(X \cup Y), \sup(X \cup Y)]$$

$$= \begin{cases} Y, \text{ if } X = \varnothing, \\ X, \text{ if } Y = \varnothing, \text{ and} \\ [\min(\underline{x}, \underline{y}), \max(\bar{x}, \bar{y})], \text{ otherwise.} \end{cases}$$

Arguments: `X` and `Y` must be intervals with the same type.

Result type: Same as `X`.

2.5.2 Intersection: $X \cap Y$ or `intersect(X,Y)`

Description: Intersection of two intervals.

Mathematical and operational definitions:

$$\texttt{intersect(X,Y)} \equiv \{z \mid z \in X \text{ and } z \in Y\}$$

$$= \begin{cases} \varnothing, \text{ if } (X = \varnothing) \text{ or } (Y = \varnothing) \text{ or } (\min(\bar{x}, \bar{y}) < \max(\underline{x}, \underline{y})) \\ [\max(\underline{x}, \underline{y}), \min(\bar{x}, \bar{y})], \text{ otherwise.} \end{cases}$$

Arguments: `X` and `Y` must be intervals with the same type.

Result type: Same as `X`.

2.6 Set Relations

C++ provides the following set relations that have been extended to support intervals.

2.6.1 Disjoint: $X \cap Y = \emptyset$ or `disjoint(X,Y)`

Description: Test if two intervals are disjoint.

Mathematical and operational definitions:

$$\begin{aligned}\texttt{disjoint(X,Y)} \equiv\ & (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X, \forall y \in Y : x \neq y)) \\ =\ & (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and} \\ & (Y \neq \emptyset) \text{ and } ((\bar{y} < \underline{x}) \text{ or } (\bar{x} < \underline{y})))\end{aligned}$$

Arguments: X and `Y` must be intervals with the same type.

Result type: `interval_bool`.

2.6.2 Element: $r \in Y$ or `in(r,Y)`

Description: Test if the number, `r`, is an element of the interval, `Y`.

Mathematical and operational definitions:

$$\begin{aligned}r \in Y \equiv\ & (\exists\, y \in Y : y = r) \\ =\ & (Y \neq \emptyset) \text{ and } (\underline{y} \leq r) \text{ and } (r \leq \bar{y})\end{aligned}$$

Arguments: The type of `r` is an integer or floating-point type, and the type of `Y` is `interval`.

Result type: `interval_bool`.

The following comments refer to the $r \in Y$ set relation:

- If `r` is `NaN` (Not a Number), `in(r, y)` is unconditionally *false*.
- If `Y` is empty, `in(r, y)` is unconditionally *false*.

2.6.3 Interior: `in_interior(X,Y)`

Description: Test if `X` is in interior of `Y`.

The interior of a set in topological space is the union of all open subsets of the set.

For intervals, the function `in_interior(X,Y)` means that X is a subset of Y, and both of the following relations are *false:*

- $\inf(Y) \in X$, or in C++: `in_interior(inf(Y), X)`
- $\sup(Y) \in X$, or in C++: `in_interior(sup(Y), X)`

Note also that, $\emptyset \notin \emptyset$, but `in_interior([empty],[empty])` = *true*

The empty set is open and therefore is a subset of the interior of itself.

Mathematical and operational definitions:

$$\texttt{in\_interior(X,Y)} \equiv (X = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X, \exists y' \in Y, \exists y'' \in Y : y' < x < y''))$$
$$= (X = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{y} < \underline{x}) \text{ and } (\bar{x} < \bar{y}))$$

Arguments: `X` and `Y` must be intervals with the same type.

Result type: `interval_bool`.

2.6.4 Proper Subset: $X \subset Y$ or `proper_subset(X,Y)`

Description: Test if `X` is a proper subset of `Y`

Mathematical and operational definitions:

$$\texttt{proper\_subset(X,Y)} \equiv (X \subseteq Y) \text{ and } (X \neq Y)$$
$$= ((X = \emptyset) \text{ and } (Y \neq \emptyset)) \text{ or}$$
$$(X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{y} \leq \underline{x}) \text{ and } (\bar{x} < \bar{y}) \text{ or}$$
$$(\underline{y} < \underline{x}) \text{ and } (\bar{x} \leq \bar{y})$$

Arguments: `X` and `Y` must be intervals with the same type.

Result type: `interval_bool`.

2.6.5 Proper Superset: $X \supset Y$ or `proper_superset(X,Y)`

Description: See proper subset with $X \leftrightarrow Y$.

2.6.6 Subset: $X \subseteq Y$ or `subset(X,Y)`

Description: Test if `X` is a subset of `Y`

Mathematical and operational definitions:

$$\texttt{subset(X,Y)} \equiv (X = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X, \exists y' \in Y, \exists y'' \in Y : y' \leq x \leq y''))$$

$$= (X = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{y} \leq \underline{x}) \text{ and } (\bar{x} \leq \bar{y}))$$

Arguments: `X` and `Y` must be intervals with the same type.

Result type: `interval_bool`.

2.6.7 Superset: $X \supseteq Y$ or `superset(X,Y)`

Description: See subset with $X \leftrightarrow Y$.

2.7 Relational Functions

2.7.1 Interval Order Relations

Ordering intervals is more complicated than ordering points. Testing whether 2 is less than 3 is unambiguous. With intervals, while the interval `[2,3]` is certainly less than the interval `[4,5]`, what should be said about `[2,3]` and `[3,4]`?

Three different classes of `interval` relational functions are implemented:

- Certainly
- Possibly
- Set

For a certainly-relation to be *true,* every element of the operand intervals must satisfy the relation. A possibly-relation is *true* if it is satisfied by any elements of the operand intervals. The set-relations treat intervals as sets. The three classes of `interval` relational functions converge to the normal relational functions on points if both operand intervals are degenerate.

To distinguish the three function classes, the two-letter relation mnemonics (`lt`, `le`, `eq`, `ne`, `ge`, and `gt`) are prefixed with the letters `c`, `p`, or `s`. The functions `seq(X,Y)` and `sne(X,Y)` correspond to the operators `==` and `!=`. In all other cases, the relational function class must be explicitly identified, as for example in:

- `clt(X,Y)` certainly less than
- `plt(X,Y)` possibly less than
- `slt(X,Y)` set less than

See Section 2.4 “Operators and Functions” on page 38 for the syntax and semantics of all `interval` functions.

The following program demonstrates the use of a set-equality test.

CODE EXAMPLE 2-9 Set-Equality Test

```
math% cat ce2-9.cc

#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X("[2,3]");
    interval <double> Y("[4,5]");
    if (X+Y == interval <double>("[6,8]"))
        cout << "Check." <<endl;
}

math% CC  -xia -o ce2-9 ce2-9.cc
math% ce2-9
Check.
```

CODE EXAMPLE 2-9 uses the set-equality test to verify that `X+Y` is equal to the interval [6, 8] using the == operator.

Use CODE EXAMPLE 2-10 and CODE EXAMPLE 2-8 on page 46 to explore the result of `interval`-specific relational functions.

CODE EXAMPLE 2-10 Interval Relational Functions

```
math% cat ce2-10.cc

#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X, Y;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<" X,Y =?";
    cin >>X >>Y;
```

CODE EXAMPLE 2-10 Interval Relational Functions *(Continued)*

```
    for(;;){
        cout <<endl << "For X =" <<X << ", and Y =" <<Y<< endl;

        cout <<"ceq(X,Y),peq(X,Y),seq(X,Y)="
             << (ceq(X,Y) ?"T ":"F ")
             << (peq(X,Y) ?"T ":"F ")
             <<(seq(X,Y) ?"T ":"F ") <<endl;

        cout <<"cne(X,Y),pne(X,Y),sne(X,Y)="
             << (cne(X,Y) ?"T ":"F ")
             << (pne(X,Y) ?"T ":"F ")
             <<(sne(X,Y) ?"T ":"F ") <<endl;

        cout <<"cle(X,Y),ple(X,Y),sle(X,Y)="
             << (cle(X,Y) ?"T ":"F ")
             << (ple(X,Y) ?"T ":"F ")
             <<(sle(X,Y) ?"T ":"F ") <<endl;

        cout <<"clt(X,Y),plt(X,Y),slt(X,Y)="
             << (clt(X,Y) ?"T ":"F ")
             << (plt(X,Y) ?"T ":"F ")
             <<(slt(X,Y) ?"T ":"F ") <<endl;
        cout <<"cge(X,Y),pge(X,Y),sge(X,Y)="
             << (cge(X,Y) ?"T ":"F ")
             << (pge(X,Y) ?"T ":"F ")
             <<(sge(X,Y) ?"T ":"F ") <<endl;

        cout <<"cgt(X,Y),pgt(X,Y),sgt(X,Y)="
             << (cgt(X,Y) ?"T ":"F ")
             << (pgt(X,Y) ?"T ":"F ")
             <<(sgt(X,Y) ?"T ":"F ") <<endl;

        cout <<" X,Y =?";
        cin >>X>>Y;
    }
}
```

CODE EXAMPLE 2-10 Interval Relational Functions *(Continued)*

```
math% CC  -xia -o ce2-10 ce2-10.cc
math% ce2-10

Press Control/C to terminate!
 X,Y =? [2] [3]
For X =[0.2000000000000000E+001,0.2000000000000000E+001], and Y
=[0.3000000000000000E+001,0.3000000000000000E+001]
ceq(X,Y),peq(X,Y),seq(X,Y)=F F F
cne(X,Y),pne(X,Y),sne(X,Y)=T T T
cle(X,Y),ple(X,Y),sle(X,Y)=T T T
clt(X,Y),plt(X,Y),slt(X,Y)=T T T
cge(X,Y),pge(X,Y),sge(X,Y)=F F F
cgt(X,Y),pgt(X,Y),sgt(X,Y)=F F F
 X,Y =? 2 3
For X =[0.1000000000000000E+001,0.3000000000000000E+001], and Y
=[0.2000000000000000E+001,0.4000000000000000E+001]
ceq(X,Y),peq(X,Y),seq(X,Y)=F T F
cne(X,Y),pne(X,Y),sne(X,Y)=F T T
cle(X,Y),ple(X,Y),sle(X,Y)=F T T
clt(X,Y),plt(X,Y),slt(X,Y)=F T T
cge(X,Y),pge(X,Y),sge(X,Y)=F T F
cgt(X,Y),pgt(X,Y),sgt(X,Y)=F T F
 X,Y =? <Control-C>
```

An `interval` relational function, denoted *qop*, is composed by concatenating both of the following:

- An operator prefix, *q* ∈ {`c`, `p`, `s`}, where `c`, `p`, and `s` stand for certainly, possibly, and set, respectively
- A relational function suffix, *op* ∈ {`lt`, `le`, `eq`, `ne`, `gt`, `ge`}

In place of `seq(X,Y)` and `sne(X,Y)`, == and != operators are accepted. To eliminate code ambiguity, all other `interval` relational functions must be made explicit by specifying a prefix.

Letting "*nop*" stand for the complement of the operator *op*, the certainly and possibly functions are related as follows:

c*op* ≡ !(p*nop*)

p*op* ≡ !(c*nop*)

Note – This identity between certainly and possibly functions holds unconditionally if *op* ∈ {`eq`, `ne`}, and otherwise, only if neither argument is empty. Conversely, the identity does not hold if *op* ∈ {`lt`, `le`, `gt`, `ge`} and either operand is empty.

Assuming neither argument is empty, TABLE 2-10 contains the C++ operational definitions of all `interval` relational functions of the form:

qop(X,Y), given X = [$\underline{x}$,$\overline{x}$] and Y = [$\underline{y}$,$\overline{y}$]).

The first column contains the value of the prefix, and the first row contains the value of the operator suffix. If the tabled condition holds, the result is *true*.

TABLE 2-10 Operational Definitions of Interval Order Relations

	lt	le	eq	ge	gt	ne
s	$\underline{x} < \underline{y}$ *and* $\overline{x} < \overline{y}$	$\underline{x} \le \underline{y}$ *and* $\overline{x} \le \overline{y}$	$\underline{x} = \underline{y}$ *and* $\overline{x} = \overline{y}$	$\underline{x} \ge \underline{y}$ *and* $\overline{x} \ge \overline{y}$	$\underline{x} > \underline{y}$ *and* $\overline{x} > \overline{y}$	$\underline{x} \ne \underline{y}$ *or* $\overline{x} \ne \overline{y}$
c	$\overline{x} < \underline{y}$	$\overline{x} \le \underline{y}$	$\overline{y} \le \underline{x}$ *and* $\overline{x} \le \underline{y}$	$\underline{x} \ge \overline{y}$	$\underline{x} > \overline{y}$	$\underline{x} > \overline{y}$ *or* $\underline{y} > \overline{x}$
p	$\underline{x} < \overline{y}$	$\underline{x} \le \overline{y}$	$\underline{x} \le \overline{y}$ *and* $\underline{y} \le \overline{x}$	$\overline{x} \ge \underline{y}$	$\overline{x} > \underline{y}$	$\overline{y} > \underline{x}$ *or* $\overline{x} > \underline{y}$

2.7.2 Set Relational Functions

For an affirmative order relation with

op ∈ {`lt`, `le`, `eq`, `ge`, `gt`} and

op ∈ { <, ≤, =, ≥, > },

between two points x and y, the mathematical definition of the corresponding set-relation, S*op*, between two non-empty intervals X and Y is:

$$Sop(X, Y) \equiv (\forall x \in X, \exists y \in Y : x \text{ op } y) \text{ and } (\forall y \in Y, \exists x \in X : x \text{ op } y).$$

For the relation ≠ between two points x and y, the corresponding set relation, `sne(X,Y)`, between two non-empty intervals X and Y is:

$$\text{sne}(X, Y) \equiv (\exists \text{ x} \in X, \forall y \in Y : x \ne \text{y}) \text{ or } (\exists \text{ y} \in Y, \forall x \in X : x \ne \text{y}).$$

Empty intervals are explicitly considered in each of the following relations. In each case:

Arguments: X and Y must be intervals with the same type.

Result type: `interval_bool`.

2.7.2.1 Set-equal: $X = Y$ or `seq(X,Y)`

Description: Test if two intervals are set-equal.

Mathematical and operational definitions:

$$\begin{aligned}\texttt{seq(X,Y)} \equiv\ & (X \cup Y = \emptyset) \text{ or } (X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and} \\ & (\forall x \in X,\ \exists y \in Y : x = y) \text{ and } (\forall y \in Y,\ \exists x \in X : x = y)) \\ =\ & ((X = \emptyset) \text{ and } (Y = \emptyset)) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{x} = \underline{y}) \text{ and } (\bar{y} = \bar{x}))\end{aligned}$$

Any interval is set-equal to itself, including the empty interval. Therefore, `seq([a,b],[a,b])` is *true*.

2.7.2.2 Set-greater-or-equal: `sge(X,Y)`

Description: See set-less-or-equal with $X \leftrightarrow Y$.

2.7.2.3 Set-greater: `sgt(X,Y)`

Description: See set-less with $X \leftrightarrow Y$.

2.7.2.4 Set-less-or-equal: `sle(X,Y)`

Description: Test if one interval is set-less-or-equal to another.

Mathematical and operational definitions:

$$\begin{aligned}\texttt{sle(X,Y)} \equiv\ & (X \cup Y = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and} \\ & (\forall x \in X,\ \exists y \in Y : x \leq y) \text{ and } (\forall y \in Y,\ \exists x \in X : x \leq y)) \\ =\ & ((X = \emptyset) \text{ and } (Y = \emptyset)) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and} \\ & (\underline{x} \leq \underline{y}) \text{ and } (\bar{x} \leq \bar{y}))\end{aligned}$$

Any interval is set-equal to itself, including the empty interval. Therefore `sle([X,X])` is *true.*

2.7.2.5 Set-less: `slt(X,Y)`

Description: Test if one interval is set-less than another.

$$\begin{aligned}\texttt{slt(X,Y)} \equiv\ & (X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and} \\ & (\forall x \in X,\ \exists y \in Y : x < y) \text{ and } (\forall y \in Y,\ \exists x \in X : x < y) \\ =\ & (X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{x} < \underline{y}) \text{ and } (\bar{x} < \bar{y})\end{aligned}$$

2.7.2.6 Set-not-equal: $X \neq Y$ or `sne(X,Y)`

Description: Test if two intervals are not set-equal.

Mathematical and operational definitions:

$$\begin{aligned}\texttt{sne(X,Y)} \equiv\ & ((X = \emptyset) \text{ and } (Y \neq \emptyset)) \text{ or } ((X \neq \emptyset) \text{ and } (Y = \emptyset)) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } ((\exists x \in X,\ \forall y \in Y : x \neq y) \text{ or} \\ & (\exists y \in Y,\ \forall x \in X : x \neq y))) \\ =\ & ((X = \emptyset) \text{ and } (Y \neq \emptyset)) \text{ or } ((X \neq \emptyset) \text{ and } (Y = \emptyset)) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } ((\underline{x} \neq \underline{y}) \text{ or } (\bar{x} \neq \bar{y})))\end{aligned}$$

Any interval is set-equal to itself, including the empty interval. Therefore `sne([X,X])`is *false.*

2.7.3 Certainly Relational Functions

The certainly relational functions are true if the underlying relation is true for every element of the operand intervals. For example, `clt([a,b],[c,d])` is true if $x < y$ for all $x \in [a, b]$ and $y \in [c, d]$. This is equivalent to $b < c$.

For an affirmative order relation with

$op \in$ {`lt`, `le`, `eq`, `ge`, `gt`} and

op $\in$ { `<`, $\leq$, `=`, $\geq$, `>` },

between two points x and y, the corresponding certainly-*true* relation `c`*op* between two intervals, `X` and `Y`, is

$$\text{cop}(X, Y) \equiv (X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X,\ \forall y \in Y : x \text{ op } y)\ .$$

With the exception of the anti-affirmative certainly-not-equal relation, if either operand of a certainly relation is empty, the result is *false*. The one exception is the certainly-not-equal relation, `cne(X,Y)`, which is *true* in this case.

Mathematical and operational definitions `cne(X,Y)`:

$$\begin{aligned}\texttt{cne(X,Y)} \equiv\ & (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and} \\ & (\forall x \in X,\ \forall y \in Y : x \neq y)) \\ =\ & (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and} \\ & ((\underline{x} > \bar{y}) \text{ or } (\bar{y} > \bar{x})))\end{aligned}$$

For each of the certainly relational functions:

Arguments: `X` and `Y` must be intervals with the same type.

Result type: `interval_bool`.

2.7.4 Possibly Relational Functions

The possibly relational functions are true if any element of the operand intervals satisfy the underlying relation. For example, `plt([X,Y])` is true if there exists an $x \in [X]$ and a $y \in [Y]$ such that $x < y$. This is equivalent to $\underline{x} < \bar{y}$.

For an affirmative order relation with

op ∈ {`lt`, `le`, `eq`, `ge`, `gt`} and

op ∈ { <, ≤, =, ≥, > } ,

between two points x and y, the corresponding possibly-true relation *Pop* between two intervals X and Y is defined as follows:

$$p\text{op}(x, y) \equiv (X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\exists x \in X, \exists y \in Y : x \text{ op } y) \quad .$$

If the empty interval is an operand of a possibly relation then the result is *false*. The one exception is the anti-affirmative possibly-not-equal relation, `pne(X,Y)`, which is *true* in this case.

Mathematical and operational definitions `pne(X,Y)`:

$$\begin{aligned} \texttt{pne(X,Y)} \equiv\ & (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\exists x \in X, \exists y \in Y : x \neq y)) \\ =\ & (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } ((\bar{x} > \underline{y}) \text{ or } (\bar{y} > \underline{x}))) \end{aligned}$$

For each of the possibly relational functions:

Arguments: `X` and `Y` must be `intervals` with the same type.

Result type: `interval_bool`.

2.8 Input and Output

The process of performing `interval` stream input/output is the same as for other non-`interval` data types.

Note – Floating-point stream manipulations do not influence interval input/output.

2.8.1 Input

When using the single-number form of an interval, the last displayed digit is used to determine the interval's width. See Section 2.8.2 "Single-Number Output" on page 61. For more detailed information, see M. Schulte, V. Zelov, G.W. Walster, D. Chiriaev, "Single-Number Interval I/O," *Developments in Reliable Computing*, T. Csendes (ed.), (Kluwer 1999).

If an infimum is not internally representable, it is rounded down to an internal approximation known to be less than the exact value. If a supremum is not internally representable, it is rounded up to an internal approximations known to be greater than the exact input value. If the degenerate interval is not internally representable, it is rounded down and rounded up to form an internal `interval` approximation known to contain the exact input value. These results are shown in CODE EXAMPLE 2-11.

CODE EXAMPLE 2-11 Single-Number Output Examples

```
math% cat ce2-11.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

main() {
    interval<double> X[8];
    for (int i = 0; i < 8 ; i++) {
      cin >> X[i];
      cout << X[i] << endl;
    }
}
```

CODE EXAMPLE 2-11 Single-Number Output Examples *(Continued)*

```
math% CC -xia ce2-11.cc -o ce2-11
math% ce2-11
1.234500
[0.1234498999999999E+001,0.1234501000000001E+001]
[1.2345]
[0.1234499999999999E+001,0.1234500000000001E+001]
[-inf,2]
[              -Infinity,0.2000000000000000E+001]
[-inf]
[              -Infinity,-.1797693134862315E+309]
[EMPTY]
[EMPTY                                           ]
[1.2345,1.23456]
[0.1234499999999999E+001,0.1234560000000001E+001]
```

2.8.2 Single-Number Output

The function `single_number_output()` is used to display intervals in the single-number form and has the following syntax, where `cout` is an output stream.

```
single_number_output(interval<float> X, ostream& out=cout)
single_number_output(interval<double> X, ostream& out=cout)
single_number_output(interval<long double> X, ostream& out=cout)
```

If the external `interval` value is not degenerate, the output format is a floating-point or integer literal (`X` without square brackets, "`[`"..."`]`"). The external value is interpreted as a non-degenerate mathematical interval $[x] + [-1,1]_{uld}$.

The single-number interval representation is often less precise than the [*inf*, *sup*] representation. This is particularly true when an interval or its single-number representation contains zero or infinity.

For example, the external value of the single-number representation for [-15, +75] is ev(`[0E2]`) = [-100, +100]. The external value of the single-number representation for [1, ∞] is ev(`[0E+inf]`) = [-∞, +∞] .

In these cases, to produce a narrower external representation of the internal approximation, the [inf, sup] form is used to display the maximum possible number of significant digits within the output field.

CODE EXAMPLE 2-12 Single-Number [*inf, sup*]-style Output

```
math% cat ce2-12.cc

#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X(-1, 10);
    interval <double> Y(1, 6);
    single_number_output(X, cout);
    cout << endl;

    single_number_output(Y, cout);
    cout << endl;
}

math% CC  -xia -o ce2-12 ce2-12.cc
math% ce2-12
[ -1.0000     , 10.000     ]
[ 1.0000     , 6.0000     ]
```

If it is possible to represent a degenerate interval within the output field, the output string for a single number is enclosed in obligatory square brackets, "[", ... "]" to signify that the result is a point.

An example of using `ndigits` to display the maximum number of significant decimal digits in the single-number representation of the non-empty interval `X` is shown in CODE EXAMPLE 2-13 on page 63.

Note – If the argument of `ndigits` is a degenerate interval, the result is `INT_MAX`.

CODE EXAMPLE 2-13 ndigits

```
math% cat ce2-10.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

main() {
    interval<double> X[8];
    X[0] = interval<double>(1.2345678, 1.23456789);
    X[1] = interval<double>(1.234567, 1.2345678);
    X[2] = interval<double>(1.23456, 1.234567);
    X[3] = interval<double>(1.2345, 1.23456);
    X[4] = interval<double>(1.5111, 1.5112);
    X[5] = interval<double>(1.511, 1.512);
    X[6] = interval<double>(1.51, 1.52);
    X[7] = interval<double>(1.5);
    for (int i = 0; i < 8 ; i++) {
      cin << X;
      cout << X << endl;
    }
}
math% CC ce2-10.cc -xia -o ce2-10
math% ce2-10
              0.12345E+001  ndigits =8
              0.12345E+001  ndigits =7
              0.12345E+001  ndigits =6
              0.12345E+001  ndigits =5
              0.15112E+001  ndigits =5
              0.151  E+001  ndigits =4
              0.152  E+001  ndigits =3
[             0.15000E+001] ndigits =2147483647
```

Increasing interval width decreases the number of digits displayed in the single-number representation. When the interval is degenerate all remaining positions are filled with zeros and brackets are added if the degenerate interval value is represented exactly.

2.8.3 Single-Number Input/Output and Base Conversions

Single-number `interval` input, immediately followed by output, can appear to suggest that a decimal digit of accuracy has been lost, when in fact radix conversion has caused a 1 or 2 ulp increase in the width of the stored input interval. For example, an input of 1.37 followed by an immediate print will result in 1.3 being output.

As shown in CODE EXAMPLE 1-6 on page 21, programs must use character input and output to exactly echo input values and internal reads to convert input character strings into valid internal approximations.

2.9 Mathematical Functions

This section lists the type-conversion, trigonometric, and other functions that accept `interval` arguments. The symbols $\underline{x}$ and $\bar{x}$ in the interval $[\underline{x}, \bar{x}]$ are used to denote its ordered elements, the infimum, or lower bound and supremum, or upper bound, respectively. In point (non-interval) function definitions, lowercase letters x and y are used to denote floating-point or integer values.

When evaluating a function, f, of an interval argument, X, the interval result, $f(X)$, must be an enclosure of its containment set, cset(f, $\{x\}$), where:

$$\text{cset}(f, \{X\}) = \{\text{cset}(f, \{x\}) \mid x \in X\}$$

A similar result holds for functions of n-variables. Determining the containment set of values that must be included when the interval $[\underline{x}, \bar{x}]$ contains values outside the domain of f is discussed in the supplementary paper [1] cited in Section 2.11 "References" on page 74. The results therein are needed to determine the set of values that a function can produce when evaluated on the boundary of, or outside its domain of definition. This set of values, called the *containment set* is the key to defining interval systems that return valid results, no matter what the value of a function's arguments or an operator's operands. As a consequence, there are no argument restrictions on any `interval` functions in C++.

2.9.1 Inverse Tangent Function `atan2(Y,X)`

This sections provides additional information about the inverse tangent function. For further details, see the supplementary paper [9] cited in Section 2.11 "References" on page 74.

Description: Interval enclosure of the inverse tangent function over a pair of intervals.

Mathematical definition:

$$\text{atan2}(Y,\ X) \supseteq \bigcup_{\substack{x \in X \\ y \in Y}} \{\theta \mid h\sin\theta = y,\ h\cos\theta = x, h = (x^2 + y^2)^{1/2}\}$$

Special values: TABLE 2-11 and CODE EXAMPLE 2-14 display the `atan2` indeterminate forms.

TABLE 2-11 `atan2` Indeterminate Forms

y_0	x_0	cset(sinθ, {y_0, x_0})	cset(cosθ, {y_0, x_0})	cset(θ, {y_0, x_0})
0	0	[-1, 1]	[-1, 1]	$[-\pi,\ \pi]$
+∞	+∞	[0, 1]	[0, 1]	$[0,\ \frac{\pi}{2}]$
+∞	-∞	[0, 1]	[-1, 0]	$[\frac{\pi}{2},\ \pi]$
-∞	-∞	[-1, 0]	[-1, 0]	$[-\pi,\ \frac{-\pi}{2}]$
-∞	+∞	[-1, 0]	[0, 1]	$[\frac{-\pi}{2},\ 0]$

CODE EXAMPLE 2-14 `atan2` Indeterminate Forms

```
math% cat ce2-14.cc
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main() {
    interval <double> X,Y;
    cout << "Press Control/C to terminate!"<< endl;
    cout <<"Y,X=?";
    cin >>Y >>X;
    for(;;) {
        cout <<endl << "For X =" <<X << endl;
        cout << "For Y =" <<Y << endl;
        cout << atan2(Y,X) << endl << endl;
        cout << "Y,X=?";
        cin >>Y >>X;
    }
}
```

CODE EXAMPLE 2-14 `atan2` Indeterminate Forms *(Continued)*

```
math% CC  -xia -o ce2-14 ce2-14.cc
math% ce2-14
Press Control/C to terminate!
Y,X=? [0] [0]
For X =[0.0000000000000000E+000,0.0000000000000000E+000]
For Y =[0.0000000000000000E+000,0.0000000000000000E+000]
[-.3141592653589794E+001,0.3141592653589794E+001]

Y,X=? inf inf
For X =[0.1797693134862315E+309,               Infinity]
For Y =[0.1797693134862315E+309,               Infinity]
[0.0000000000000000E+000,0.1570796326794897E+001]

Y,X=? inf -inf
For X =[              -Infinity,-.1797693134862315E+309]
For Y =[0.1797693134862315E+309,               Infinity]
[0.1570796326794896E+001,0.3141592653589794E+001]

Y,X=? -inf inf
For X =[0.1797693134862315E+309,               Infinity]
For Y =[              -Infinity,-.1797693134862315E+309]
[-.1570796326794897E+001,0.0000000000000000E+000]

Y,X=? -inf -inf
For X =[              -Infinity,-.1797693134862315E+309]
For Y =[              -Infinity,-.1797693134862315E+309]
[-.3141592653589794E+001,-.1570796326794896E+001]

Y,X=? <Control-C>
```

Result value: The interval result value is an enclosure for the specified interval. An ideal enclosure is an interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if one or both arguments are empty.

In the case where $\overline{x} < 0$ and $0 \in Y$, to get a sharp interval enclosure (denoted by Θ), the following convention uniquely defines the set of all possible returned interval angles:

$$-\pi < m(\Theta) \leq \pi$$

This convention, together with

$$0 \leq w(\Theta) \leq 2\pi$$

results in a unique definition of the interval angles Θ that `atan2(Y,X)` must include.

TABLE 2-12 contains the tests and arguments of the floating-point `atan2` function that are used to compute the endpoints of Θ in the algorithm that satisfies the constraints required to produce sharp interval angles. The first two columns define the distinguishing cases. The third column contains the range of possible values of the midpoint, $m(\Theta)$, of the interval Θ. The last two columns show how the endpoints of Θ are computed using the floating-point `atan2` function. Directed rounding must be used to guarantee containment.

TABLE 2-12 Tests and Arguments of the Floating-Point `atan2` Function

Y	X	$m(\Theta)$	$\underline{\theta}$	$\overline{\theta}$
$-\underline{y} < \overline{y}$	$\overline{x} < 0$	$\frac{\pi}{2} < m(\Theta) < \pi$	`atan2`$(\overline{y}, \overline{x})$	`atan2`$(\underline{y}, \overline{x}) + 2\pi$
$-\underline{y} = \overline{y}$	$\overline{x} < 0$	$m(\Theta) = \pi$	`atan2`$(\overline{y}, \overline{x})$	$2\pi - \underline{\theta}$
$\bar{y} < -\underline{y}$	$\overline{x} < 0$	$-\pi < m(\Theta) < \frac{-\pi}{2}$	`atan2`$(\overline{y}, \overline{x}) - 2\pi$	`atan2`$(\underline{y}, \overline{x})$

2.9.2 Maximum: `maximum(X1,X2)`

Description: Range of maximum.

The containment set for $\max(X_1, ..., X_n)$ is:

$$\{z \mid z = \max(x_1, \ldots, x_n), x_i \in X_i\} = [\sup(\text{hull}(\underline{x_1}, \ldots, \underline{x_n})),\ \sup(\text{hull}(\overline{x_1}, \ldots, \overline{x_n}))] .$$

The implementation of the `max` function must satisfy:

`maximum(X1,X2,[X3,...])` $\supseteq$ $\{\max(X_1, ..., X_n)\}$.

2.9.3 Minimum: `minimum(X1,X2)`

Description: Range of minimum.

The containment set for $\min(X_1, ..., X_n)$ is:

$$\{z \mid z = \min(x_1, \ldots, x_n), x_i \in X_i\} = [\inf(\text{hull}(\underline{x_1}, \ldots, \underline{x_n})),\ \inf(\text{hull}(\overline{x_1}, \ldots, \overline{x_n}))] .$$

The implementation of the `min` function must satisfy:

`minimum(X1,X2,[X3,...])` $\supseteq$ $\{\min(X_1, ..., X_n)\}$.

2.9.4 Functions That Accept Interval Arguments

TABLE 2-14 through TABLE 2-18 list the properties of functions that accept interval arguments. TABLE 2-13 lists the tabulated properties of `interval` functions in these tables.

TABLE 2-13 Tabulated Properties of Each `interval` Function

Tabulated Property	Description
Function	what the function does
Definition	mathematical definition
No. of Args.	number of arguments the function accepts
Name	the function's name
Argument Type	valid argument types
Function Type	type returned for specific argument data type

Because indeterminate forms are possible, special values of the `pow` and atan2 function are contained in Section 2.4.2 "Power Function pow(X,n) and pow(X,Y)" on page 43 and Section 2.9.1 "Inverse Tangent Function atan2(Y,X)" on page 64, respectively. The remaining functions do not require this treatment.

TABLE 2-14 `interval` Constructor

Conversion To	No. of Args.	Name	Argument Type	Function Type

`interval`	1, 2	`interval`	`const char*`
			`const interval<float>&`
			`const interval<double>&`
			`const interval<long double>&`
			`int`
			`long long`
			`float`
			`double`
			`long double`
			`int, int`
			`long long, long long`
			`float, float`
			`double, double`
			`long double, long double`
				The function type can be `interval<float>`, `interval<double>`, or `interval<long double>` for each argument type.

TABLE 2-15 `interval`-Specific Functions

Function	Definition	No. of Args.	Name	Argument Type	Function Type

Infimum	inf([*a*, *b*]) = *a*	1	`inf`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`float`
					`long double`

Supremum	sup([*a*, *b*]) = *b*	1	`sup`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`float`
					`long double`

Width	w([*a*, *b*]) = *b* - *a*	1	`wid`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`float`
					`long double`

Midpoint	mid([*a*, *b*]) = (*a* + *b*)/2	1	`mid`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`float`
					`long double`

Magnitude[1]	max(| *a* |) ∈ *A*	1	`mag`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`float`
					`long double`

Mignitude[2]	min(| *a* |) ∈ *A*	1	`mig`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`float`
					`long double`

Test for empty interval	*true* if *A* is empty	1	`isempty`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`interval_bool`
					`interval_bool`
					`interval_bool`

Floor	floor(*A*)	1	`floor`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`double`
					`double`

Ceiling	ceiling(*A*)	1	`ceil`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`double`
					`double`
					`double`

Number of digits[3]	Maximum number of significant decimal digits in the single-number representation of a non-empty interval	1	`ndigits`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`int`
					`int`
					`int`

(1) mag([*a*, *b*]) = max(\| *a* \| , \| *b* \|)

(2) mig([*a*, *b*]) = min(\| *a* \| , \| *b* \|), if *a* > 0 or *b* < 0, otherwise 0

(3) Special cases: `ndigits([-inf, +inf])` = `ndigits([empty])` = 0

TABLE 2-16 `interval` Arithmetic Functions

Function	Point Definition	No. of Args.	Name	Argument Type	Function Type

Absolute value	$\lvert a \rvert$	1	`fabs`	`interval <double>`
				`interval <float>`
				`interval <long double>`
					`interval <double>`
					`interval <float>`
					`interval <long double>`

Remainder	a-b(int(a/b))	2	`fmod`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Choose largest value[1]	max(a,b)	2	`maximum`	`interval <double>`	`interval <double>`
Choose smallest value[1]	min(a,b)	2	`minimum`	`interval <double>`	`interval <double>`

(1) The `minimum` and `maximum` functions ignore empty interval arguments unless all arguments are empty, in which case, the empty interval is returned.

TABLE 2-17 `interval` Trigonometric Functions

Function	Point Definition	No. of Args.	Name	Argument Type	Function Type
Sine	sin(a)	1	`sin`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Cosine	cos(a)	1	`cos`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Tangent	tan(a)	1	`tan`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Arcsine	arcsin(a)	1	`asin`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Arccosine	arccos(a)	1	`acos`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Arctangent	arctan(a)	1	`atan`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Arctangent[1]	arctan(a/b)	2	`atan2`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Hyperbolic Sine	sinh(a)	1	`sinh`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Hyperbolic Cosine	cosh(a)	1	`cosh`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Hyperbolic Tangent	tanh(a)	1	`tanh`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`

(1) arctan(a/b) = θ, given $a = h \sin\theta$, $b = h \cos\theta$, and $h^2 = a^2 + b^2$.

TABLE 2-18 Other `interval` Mathematical Functions

Function	Point Definition	No. of Args.	Name	Argument Type	Function Type
Square Root[1]	$\exp\{\ln(a)/2\}$	1	`sqrt`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Exponential	$\exp(a)$	1	`exp`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Natural logarithm	$\ln(a)$	1	`log`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`
Common logarithm	$\log(a)$	1	`log10`	`interval <double>` `interval <float>`	`interval <double>` `interval <float>`

(1) sqrt(a) is multi-valued. A proper interval enclosure must contain both the positive and negative square roots. Defining the `sqrt` function to be

$$\exp\left\{\frac{\ln a}{2}\right\}$$

eliminates this difficulty.

2.10 Interval Types and the Standard Template Library

When interval types are used as template arguments for STL classes, a blank must be inserted between two consecutive > symbols, as shown on the line marked `note 1` in CODE EXAMPLE 2-15.

CODE EXAMPLE 2-15 Example of Using an Interval Type as a Template Argument for STL Classes

```
math% cat ce2-15.cc
#include <limits.h>
#include <strings.h>
#include <sunmath.h>
#include <stack>
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main()
{
   std::stack<interval<double> > st; //note 1
    return 0;
}
math% CC -xia ce2-15.cc
```

Otherwise, >> is incorrectly interpreted as the right shift operator, as shown on the line marked `note 1` in CODE EXAMPLE 2-16.

CODE EXAMPLE 2-16 >> Incorrectly Interpreted as the Right Shift Operator

```
math% cat ce2-16.cc
#include <limits.h>
#include <strings.h>
#include <sunmath.h>
#include <stack>
#include <suninterval.h>

#if __cplusplus >= 199711
using namespace SUNW_interval;
#endif

int main()
{
   std::stack<interval<double>> st; //note 1
    return 0;
}
math% CC ce2-16.cc
"t.cc", line 13: Error: "," expected instead of ">>".
"t.cc", line 13: Error: Illegal value for template parameter.
"t.cc", line 13: Error: "," expected instead of ">>".
"t.cc", line 13: Error: Illegal value for template parameter.
"t.cc", line 13: Error: "," expected instead of ";".
"t.cc", line 13: Error: Illegal value for template parameter.
6 Error(s) detected.
```

Note – Interpreting >> as a right shift operator is a general design problem in C++.

2.11 References

The following technical reports are available online. See the interval arithmetic readme for the location of these files.

1. G.W. Walster, E.R. Hansen, and J.D. Pryce, "Extended Real Intervals and the Topological Closure of Extended Real Relations," Technical Report, Sun Microsystems. February 2000.
2. G. William Walster, "Empty Intervals," Technical Report, Sun Microsystems. April 1998.
3. G. William Walster, "Closed Interval Systems," Technical Report, Sun Microsystems. August 1999.
4. G. William Walster, "Literal Interval Constants," Technical Report, Sun Microsystems. August 1999.
5. G. William Walster, "Widest-Need Interval Expression Evaluation," Technical Report, Sun Microsystems. August 1999.
6. G. William Walster, "Compiler Support of Interval Arithmetic With Inline Code Generation and Nonstop Exception Handling," Technical Report, Sun Microsystems. February 2000.
7. G. William Walster, "Finding Roots on the Edge of a Function's Domain," Technical Report, Sun Microsystems. February 2000.
8. G. William Walster, "Implementing the 'Simple' Closed Interval System," Technical Report, Sun Microsystems. February 2000.
9. G. William Walster, "Interval Angles and the Fortran ATAN2 Intrinsic Function," Technical Report, Sun Microsystems. February 2000.
10. G. William Walster, "The 'Simple' Closed Interval System," Technical Report, Sun Microsystems. February 2000.
11. G. William Walster, Margaret S. Bierman, "Interval Arithmetic in Forte Developer Fortran," Technical Report, Sun Microsystems. March 2000.

Glossary

affirmative relation An order relation other than certainly, possibly, or set not equal. *Affirmative relations* affirm something, such as $a < b$.

affirmative relational functions An *affirmative relational function* is an element of the set: {<, ≤, =, ≥, >}.

anti-affirmative relation An *anti-affirmative relation* is a statement about what cannot be true. The order relation ≠ is the only anti-affirmative relation in C++.

anti-affirmative relational function The C++ `!=` operator implements the anti-affirmative relation. The certainly, possible, and set functions for interval arguments are denoted `cne`, `pne`, and `sne`, respectively.

assignment statement An interval *assignment statement* is a C++ statement having the form: `V = expression.` The left-hand side of the assignment statement is the interval variable or array element `V`.

certainly true relational function See ***relational functions: certainly true***.

closed interval A *closed interval* includes its endpoints. A closed interval is a *closed set*. The interval [2, 3] = $\{z \mid 2 \le z \le 3\}$ is closed, because its endpoints are included. The interval (2, 3) = $\{z \mid 2 < z < 3\}$ is open, because its endpoints are not included. Interval arithmetic, as implemented in C++, only deals with closed intervals.

closed mathematical system In a *closed mathematical system*, there can be no undefined operator-operand combinations. All defined operations on elements of a closed system must produce elements of the system. The real number system is not closed, because division by zero is undefined in this system.

closed set

A *closed set* contains all limit or accumulation points in the set. That is, given the set, S, and sequences, $\{s_j\} \in S$, the closure of S is $\bar{S} = \{\lim_{j \to \infty} s_j \mid s_j \in S\}$, where $\lim_{j \to \infty}$ denotes an accumulation or limit point of the sequence $\{s_j\}$.

The set of real numbers is the open set $\{z \mid -\infty < z < +\infty\}$, because it does not include $-\infty$ and $+\infty$. The set of extended real numbers, $\Re^*$, is closed.

closure-composition equality

Given the expressions f, g, and h, with

$$f(\{x_0\}) = g(\{(y, x_0) \mid y = h(\{x_0\})\}),$$

the closure-composition equality states that

$$\bar{f}(\{x_0\}) = \bar{g}(\{(y, x_0) \mid y = \bar{h}(\{x_0\})\}).$$

The closure of $\bar{f}$ at the point x_0 is equal to the composition of its component's closures, $\bar{g}$ and $\bar{h}$.

closure of expression

The closure of the expression *f of n*-variables, evaluated over the set $\boldsymbol{X}_0$ is denoted $\bar{f}(\boldsymbol{X}_0)$, and for $x_0 \in \bar{D}_f$ is defined:

$$\bar{f}(\boldsymbol{X}_0) = \left\{ z \;\middle|\; \begin{array}{l} z = \lim\limits_{j \to \infty} y_j \\ y_j \in f(\{\boldsymbol{x}_j\}) \\ \boldsymbol{x}_j \in D_f \\ \lim\limits_{j \to \infty} \boldsymbol{x}_j \in \boldsymbol{X}_0 \end{array} \right\}$$

For $\boldsymbol{X}_0 \notin \bar{D}_f$, $\bar{f}(\boldsymbol{X}_0) = \varnothing$.

The accumulation points, $\lim_{j \to \infty} x_j$, of all sequences, $\{\boldsymbol{x}_j\}$, are elements of the set, $\boldsymbol{X}_0$. The closure of f is the set of all possible accumulation points of f given the conditions on the right-hand side of the above defining expression are satisfied.

$\bar{f}(\boldsymbol{X}_0)$ is defined for all $\boldsymbol{X}_0 \in (\Re^*)^n$.

connected set

The *connected set* of numbers between and including two values, $a \le b$, contains all the values between and including a and b.

composite expression

Forming a new expression, f, (the *composite expression*) from the given expressions, g and h by the rule $f(\{\underline{x}\}) = g(h(\{\underline{x}\}))$ for all singleton sets, $\{\underline{x}\} = \{x_1\} \otimes \ldots \otimes \{x_n\}$ in the domain of h for which h is in the domain of g. Singleton set arguments connote the fact that expressions can be either functions or relations.

containment constraint

The *containment constraint* on the interval evaluation, $f([x])$, of the expression, f, at the degenerate interval, $[x]$, is:

$f([x]) \supseteq \text{cset}(f,\{x\})$,

where $\text{cset}(f,\{x\})$ denotes the containment set of all possible values that $f([x])$ must contain. Because the containment set, $\text{cset}(x \div y, \{(1, 0)\}) = \{-\infty, +\infty\}$, $[1] / [0] = \text{hull}(\{-\infty, +\infty\}) = [-\infty, +\infty]$. See also ***containment set***.

containment failure

A *containment failure* is a failure to satisfy the containment constraint. For example, a containment failure results if [1]/[0] is defined to be [*empty*]. This can be seen by considering the interval expression

$$\frac{X}{X+Y} = \frac{1}{1+\frac{Y}{X}}$$

for X=[0] and Y, given $0 \notin Y$. The containment set of the first expression is [0]. However, if [1]/[0] is defined to be [*empty*], the second expression is also [*empty*]. This is a containment failure.

containment set

The *containment set*, $\text{cset}(h, \{x\})$, of the expression h is the smallest set that does not violate the containment constraint when h is used as a component of any composition, $f(\{x\}) = g(h(\{x\}), \{x\})$.

For $h(x, y) = x \div y$,

$\text{cset}(h, \{(+\infty, +\infty)\}) = [0, +\infty]$.

See also ***cset(expression, set)***.

containment set closure identity

Given any expression $f(\{\boldsymbol{x}\}) = f(\{x_1\} \otimes \ldots \otimes \{x_n\})$ of n-variables and the point, $\boldsymbol{x}_0$, then $\text{cset}(f, \{\boldsymbol{x}_0\}) = \bar{f}(\{\boldsymbol{x}_0\})$, the closure of f at the point, $\boldsymbol{x}_0$.

containment set equivalent

Two expressions are *containment-set equivalent* if their containment sets are everywhere identical.

containment set evaluation theorem

Let $\overline{\text{eval}}(f, \{\boldsymbol{x}\})$ denote the code-list evaluation of the expression f, using individual composition closures to compute the value of every sub-expression, whether it is the value of an operation, function, or relation. Given the expression, $f(\{\boldsymbol{x}\}) = f(\{x_1\} \otimes \ldots \otimes \{x_n\})$, whether f is a function or a relation, then for all $\boldsymbol{x}_0 \in (\Re^*)^n$, $\text{cset}(f, \{\boldsymbol{x}_0\}) \subseteq \overline{\text{eval}}(f, \{\boldsymbol{x}_0\})$.

cset(*expression, set*)

The notation, cset(*expression*, *set*), is used to symbolically represent the containment set of an expression evaluated over a set of arguments. For example, for the expression, $f(x, y) = xy$, the containment constraint that the interval expression $[0] \times [+\infty]$ must satisfy is

$[0] \times [+\infty] \supseteq \text{cset}(x \times y, \{(0,+\infty)\}) = [-\infty, +\infty]$.

degenerate interval A *degenerate interval* is a zero-width interval. A degenerate interval is a singleton set, the only element of which is a point. In most cases, a degenerate interval can be thought of as a point. For example, the interval [2, 2] is degenerate, and the interval [2, 3] is not.

directed rounding *Directed rounding* is rounding in a particular direction. In the context of interval arithmetic, rounding up is towards +∞, and rounding down is towards -∞. The direction of rounding is symbolized by the arrows, $\downarrow$ and $\uparrow$. Therefore, with 5-digit arithmetic, $\uparrow$ 2.00001 = 2.0001. Directed rounding is used to implement interval arithmetic on computers so that the containment constraint is never violated.

disjoint interval Two *disjoint intervals* have no elements in common. The intervals [2, 3] and [4, 5] are disjoint. The intersection of two disjoint intervals is the empty interval.

empty interval The *empty interval,* [*empty*], is the interval with no members. The empty interval naturally occurs as the intersection of two disjoint intervals. For example, [2, 3] ∩ [4,5] = [*empty*].

empty set The *empty set*, ∅, is the set with no members. The empty set naturally occurs as the intersection of two disjoint sets. For example, {2, 3} ∩ {4, 5} = ∅.

ev(SRIC) The notation ev(SRIC) is used to denote the external value defined by a SRIC. For example, ev(`"[0.1]"`) = 1/10, in spite of the fact that a non-degenerate interval approximation of 0.1 must be used, because the constant 0.1 is not machine representable. See also ***string representation of an interval constant (SRIC)***.

exception In the IEEE 754 floating-point standard, an *exception* occurs when an attempt is made to perform an undefined operation, such as division by zero.

exchangeable expression Two expressions are exchangeable if they are containment-set equivalent (their containment sets are everywhere identical).

extended interval The term *extended interval* refers to intervals whose endpoints can be extended real numbers, including -∞ and +∞. For completeness, the empty interval is also included in the set of extended real intervals.

external representation The *external representation* of a C++ data item is the character string used to define it during input data conversion, or the character string used to display it after output data conversion.

external value The *external value* of a SRIC is the mathematical value defined by the SRIC. The external value of a SRIC might not be the same as the SRIC's internal approximation, which, in C++, is the only defined value of the SRIC. See also ***ev(SRIC)***.

hull See ***interval hull***.

infimum (plural, infima)
The *infimum* of a set of numbers is the set's greatest lower bound. This is either the smallest number in the set or the largest number that is less than all the numbers in the set. The infimum, inf([*a*, *b*]), of the interval constant [*a*, *b*] is *a*.

interval algorithm
An *interval algorithm* is a sequence of operations used to compute an interval result.

internal approximation
In the C++ interval class, an interval constant is represented using a string. The string representation of an interval constant (or SRIC) has an internal approximation, which is the sharp internal approximation of the SRIC's external value. The external value is an interval constant. See also ***string representation of an interval constant (SRIC)***

interval arithmetic
Interval arithmetic is the system of arithmetic used to compute with intervals.

interval box
An interval box is a parallelepiped with sides parallel to the *n*-dimensional Cartesian coordinate axes. An interval box is conveniently represented using an *n*-dimensional interval vector, $\boldsymbol{X} = (X_1, \ldots, X_n)^T$.

interval constant
An *interval constant* is the closed connected set: $[a, b] = \{z \mid a \le z \le b\}$ defined by the pair of numbers, $a \le b$.

interval constant's external value
See ***external value***.

interval constant's internal approximation
See ***internal approximation***.

interval hull
The *interval hull* function, $\underline{\cup}$, on a pair of intervals $X = [\bar{x}, \underline{x}]$ and $Y = [\bar{y}, \underline{y}]$, is the smallest interval that contains both X and Y (also represented as $[\inf(X \cup Y),\ \sup(X \cup Y)]$). For example,

$[2, 3] \ \underline{\cup}\ [5, 6] = [2, 6]$.

`interval`-specific function
In the C++ interval class, an ***`interval`****-specific function* is an interval function that is not an interval version of a standard C++ function. For example, `wid`, `mid`, `inf`, and `sup`, are `interval`-specific functions.

interval width
Interval width, $w([a, b]) = b - a$.

left endpoint
The *left endpoint* of an interval is the same as its infimum or lower bound.

literal constant
No literal constant construct for user-defined objects is provided in C++ classes. Therefore, a string representation of a literal constant (or SRIC) is used instead. See also ***string representation of an interval constant (SRIC)***.

lower bound
See ***infimum (plural, infima)***.

mantissa When written in scientific notation, a number consists of a *mantissa* or significand and an exponent power of 10.

multiple-use expression (MUE) A *multiple-use expression (MUE)* is an expression in which at least one independent variable appears more than once.

narrow-width interval Let the interval $[a, b]$ be an approximation of the value $v \in [a, b]$. If $w[a, b] = b - a$, is small, $[a, b]$ is a *narrow-width interval*. The narrower the width of the interval $[a, b]$, the more accurately $[a, b]$ approximates v. See also ***sharp interval result***.

opaque data type An *opaque data type* leaves the structure of internal approximations unspecified. `interval` data items are opaque. Therefore, programmers cannot count on `interval` data items being internally represented in any particular way. The intrinsic functions `inf` and `sup` provide access to the components of an interval. The `interval` constructor can be used to manually construct any valid interval.

point A *point* (as opposed to an interval), is a number. A point in *n*-dimensional space, is represented using an *n*-dimensional vector, $\boldsymbol{x} = (x_1, \ldots, x_n)^T$. A point and a degenerate interval, or interval vector, can be thought of as the same. Strictly, any interval is a set, the elements of which are points.

possibly true relational functions See ***relational functions: possibly true***.

quality of implementation *Quality of implementation*, is a phrase used to characterize properties of compiler support for intervals. Narrow width is a new quality of implementation opportunity provided by intrinsic compiler support for `interval` data types.

radix conversion *Radix conversion* is the process of converting back and forth between external decimal numbers and internal binary numbers. Radix conversion takes place in formatted and list-directed input/output. Because the same numbers are not always representable in the binary and decimal number systems, guaranteeing containment requires directed rounding during radix conversion.

relational functions: certainly true The *certainly true relational functions* are {`clt`, `cle`, `ceq`, `cne`, `cge`, `cgt`}. Certainly true relational functions are true if the relation in question is true for all elements in the operand intervals. That is cop ($[a, b]$, $[c, d]$) = *true* if op(x, y) = *true* for all $x \in [a, b]$ and $y \in [c, d]$.

For example, `clt`($[a, b]$, $[c, d]$) evaluates to *true* if $b < c$.

relational functions: possibly true The *possibly true relational functions* are {`plt`, `ple`, `peq`, `pne`, `pge`, `pgt`}. Possibly true relational functions are true if the relation in question is true for any elements in operand intervals. For example, `plt(`$[a, b]$, $[c, d]$) if $a < d$.

relational functions: set

The *set relational functions* are {`slt`, `sle`, `seq`, `sne`, `sge`, `sgt`}. Set relational functions are true if the relation in question is true for the endpoints of the intervals. For example, `seq(`[*a*, *b*], [*c*, *d*]) evaluates to *true* if (*a* = *c*) and (*b* = *d*).

right endpoint

See ***supremum (plural, suprema)***.

set theoretic

Set theoretic is the means of or pertaining to the algebra of sets.

sharp interval result

A *sharp interval result* has a width that is as narrow as possible. A sharp interval result is equal to the hull of the expression's containment. Given the limitations imposed by a particular finite precision arithmetic, a sharp interval result is the narrowest possible finite precision interval that contains the expression's containment set.

single-number input/output

Single-number input/output, uses the single-number external representation for an interval, in which the interval $[-1, +1]_{uld}$ is implicitly added to the last displayed digit. The subscript *uld* is an acronym for unit in the last digit. For example 0.12300 represents the interval $0.12300 + [-1, +1]_{uld} = [0.12299, 0.12301]$.

single-number `interval` data conversion

Single-number `interval` *data conversion* is used to read and display external intervals using the single-number representation. See ***single-number input/output***.

single-use expression (SUE)

A *single-use expression (SUE)* is an expression in which each variable only occurs once. For example

$$\frac{1}{1+\frac{Y}{X}}$$

is a single use expression, whereas

$$\frac{X}{X+Y}$$

is not.

string representation of an interval constant (SRIC)

In C++, it is possible to define variables of a class type, but not literal constants. So that a literal interval constant can be represented, the C++ interval class uses a string to represent an interval constant. A string representation of an interval constant (SRIC), such as `"[0.1,0.2]"`, is the character string that represents a literal interval constant. See Section 2.1.1 "String Representation of an Interval Constant (SRIC)" on page 28.

SRIC's external value In the C++ interval class, a literal interval constant is represented using a string. This is referred to as the string representation of an interval constant, or SRIC. The external value of a SRIC, or ev(SRIC), is the exact mathematical value the SRIC represents. For example, the SRIC `"[0.1]"` has the external value: ev(`"[0.1]"`) = 1/10. See also ***string representation of an interval constant (SRIC)***.

SRIC's internal approximation In the C++ interval class, a literal interval constant is represented using a string. This is referred to as the string representation of an interval constant, or SRIC. The internal approximation of a SRIC, is the sharp machine representable interval that contains the SRIC's external value. For example, the internal approximation of the SRIC `"[0.1]"` is the narrowest possible machine representable interval that contains the SRIC's external value, which, in this case, is ev(`"[0.1]"`) = 1/10. See also ***string representation of an interval constant (SRIC)***.

supremum (plural, suprema) The *supremum* of a set of numbers is the set's least upper bound, which is either the largest number in the set or the smallest number that is greater than all the numbers in the set. The supremum, sup([*a*, *b*]), of the interval constant [*a*, *b*] is *b*.

unit in the last digit (uld) In single number input/output, one *unit in the last digit (uld)* is added to and subtracted from the last displayed digit to implicitly construct an interval.

unit in the last place (ulp) One *unit in the last place (ulp)* of an internal machine number is the smallest possible increment or decrement that can be made using the machine's arithmetic. Therefore, if the width of a computed interval is 1-ulp, this is the narrowest possible non-degenerate interval with a given type.

upper bound See ***supremum (plural, suprema)***.

valid interval result A *valid interval result*, [*a*, *b*] must satisfy two requirements:

- $a \leq b$
- [*a*, *b*] must not violate the containment constraint

Index

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W