N2022 Input & Output of NaN and infinity for the C++ Standard Library

Document number: JTC 1/SC22/WG21/N2022=06-0092
Date: 2006-05-15, version 3
Project: Languages C++

References:
1. C++ ISO/IEC IS 14882:1998(E)
2. William Kahan http://http.cs.berkley.edu/~wkahan/ieee754status/ieee754.ps
3. http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf
4. http://754r.ucbtest.org/drafts/754r.pdf Latest IEEE754 draft.
5. DRAFT Standard for Floating-Point Arithmetic P754/D0.17.4 2006 May 9 15:13
6. http://www2.open-std.org/JTC1/SC22/WG14/www/C99RationaleV5.10.pdf

Reply to: Paul A Bristow, pbristow@hetp.u-net.com, J16/04-0108, www.hetp.u-net.com

Contents

1 Background & motivation

Why is this important? What kinds of problems does it address, and what kinds of programmers is it intended to support? Is it based on existing practice?

Although C++ limits library provides values for quiet_NaN and infinity, and C99/ C++TR1 provides a mechanism for testing if values are finite or infinite or NaN, there is no **Standard** mechanism for streaming input and output of these as decimal digit strings. This has impeded the potential use of NaN as a way of storing 'unknown' or 'missing' values. These values may exist either because of some computation problem in their evaluation, or because they were never measured or their input left blank. Similarly, the usefulness of Infinity has been much reduced because it is not possible to output or input it.

This lack has been highlighted by use of the Boost.Serialization and Boost.Lexical_cast libraries when the facility of storing both NaN and infinite values has been much missed (compared to the invaluable use of NotADate by the Boost.DateTime library).

Some iostream implementations provide a string representation output as a result of, for example:

```
std::cout << std::numeric_limits<double>::quiet_NaN() << std::endl;
```

but

```
double v;
std::cin >> v;
```

fails to assign the hoped-for NaN to v (and worse, may assign a misleading valid numeric value to v!).

This proposal is prompted by the addition of C99 functions isnan and isfinite (and signbit) to C++: these now provide a portable and standard method of checking whether floating-point values are NaN or infinite, something that was (regrettably) not possible before.

It also tries to take account of various floating-point hardware, some of which purports to comply with some version of IEEE754. But the IEE754 family of Standards exhibits changes, contradictions, imprecisions, undefined behavior, and each platform & compile options provides different combinations of partial compliance, non-compliance, and downright un-compliance: a pragmatic solution is needed.

2 Impacts on the C++ Standards

This is a pure addition to the existing iostream class and numeric_limits class. It does not require any language change, nor any change to the existing values provided by numeric_limits. It may be sensible to add additional indications that an implementation provides both input and output of these standard strings (it is difficult to see why either one of these should be provided without the other). It will not change the behavior of existing programs that only stream finite floating-point values.

3 Design Decisions

How to detect what the hardware and software can provide?

```
std::numeric_limits<FPType>::is_iec559() == true
```

makes a claim for IEEE754 compliance, but that Standard still leaves quite a lot implementation-defined. And might one expect to have is_ieee854 and/or is_ieee754r in the near future?

Instead it seems better to provide specific indicators of the numeric_limits<FPType>::has_*() form:

```
std::numeric_limits<FPType>::has_signed_NaN()
std::numeric_limits<FPType>::has_signed_infinity()
std::numeric_limits<FPType>::has_signed_zero()
```

How to detect sign?

I note that C99 specifically names its sign detection function as signbit and not just sign, recognizing that the sign of NaN and zero and even infinity is problematic. I presume it remains true that all floating-point formats have a bit representing sign which can be both sensed and changed (the VAX floating-point, optionally used by Alpha, may not easily permit setting the sign bit – it appears NOT possible using copysign).

C Rationale says:
"7.12.3.6 The signbit macro
This is testing the sign of the value, not the sign of the exponent.
15 The model of floating-point (§5.2.4.2.2) assumes that there are three distinct fields in a floating-point representation: a sign of the value, a signed exponent, and a significand. This macro converts

the encoding of the sign of the value (even if the sign is ignored by the value, such as NaNs). The model also assumes that the same encoding fields are used for all values."

So it seems safe to rely on signbit, regardless of whether or not it has any ‘meaning’.

How to detect NaN and infinities?

It is proposed to rely entirely on the C99 functions isnan and isinf to detect Not-A-Numbers and Infinities respectively. This places the burden of deciphering the floating-point format on the C99/C++TR1 library.

Signaling NaNs?

Signaling NaNs, if detected by isnan, are assumed to be treated exactly like quiet NaNs, and after ‘round-tripping’ by output to a string or file and re-input will re-appear as

```
std::numeric_limits<>FPT>::quiet_NaN()
```

How to detect the ability to handle streaming of NaNs and infinities?

I propose adding to numeric_limits, to indicate the presence of (new) C99 Standard compliant code to handle streaming of NaNs and infinities.

Initially I suggested

```
static const bool has_NaN_io = true;
static const bool has_infinity_io = true;
```

However, existing systems provide a wide variation of input and/or output, so it seems better to indicate each of these capabilities separately for input and output.

```
static const bool has_NaN_input = true;
static const bool has_NaN_output = true;
static const bool has_infinity_input = true;
static const bool has_infinity_output = true;
```

Obviously, if the type is not specialized, or if the streaming input and output of NaNs and infinities are not implemented (yet), these values will be false: this allows existing programs to sense what degree of portability can be achieved.

How to allow programs that can handle NaN and infinity IO to adopt their previous behavior?

An optional implementation defined macro would seem the obvious way of permitting programs which rely on their previous behavior to work as with current IO libraries.

Hexadecimal format?

Although it has been tempting to consider a hexadecimal representation, so that all possible floating-point representations (including all NaNs and infinite and finite values) can be output and input, this is inherently dependent on the floating-point size, and floating-point layout, and thus much less portable.

So for simplicity, I therefore propose string representations that do not attempt to consider the ‘value’ of NaNs and infinities for all floating-point types and all sizes and all layouts, including byte order or endianness.

Existing systems

Microsoft Visual C++ provides an output for infinity as “1.#INF”,
but this string is NOT read as infinity on input, but as “1” and, unhelpfully, gives no indication from the stream state that this value may be misleading. Similarly for NaNs, there are no clues on re-input, unless a program checks the terminating characters for the NaN warning.

VisualAge C++ in contrast does handle both output *and input* of NaNs and infinities as strings “NAN” and “INF” whose case can be controlled for “nan” and “inf”, but is ignored on input.

The Dinkumware library also uses these C99-compliant strings.

GCC provides a C99 standard `[-]inf' or `[-]infinity' for infinity, and a string starting with `nan' for NaN, in the case of **f** conversion, and `[-]INF' or `[-]INFINITY' or `NAN*' in the case of **F** conversion and also strtod can handle input of these.

Standard NaN and infinity strings, or variable?
Some systems might want to print “missing” or “undefined”, something that is already possible by changes to num_put, but it is my belief that Standard strings, at least for the classic C locale, would be much more widely useful because, at least for Boost.Serialization and Boost.Lexical_cast, portability is most important, especially as very many systems have and use 64-bit doubles whose decimal 17 digit string representation is identical, even if the internal layout is not.

Same string for all types, sizes and layouts?

It is also necessary to consider if the string for all NaNs and infinities should be the same for all floating-point types, float, double and long, and also for User Defined Floating-point types, for example, the NTL classes, quad_float (128-bit with a 113-bit significand) and class RR, offering an arbitrary precision, typically 100 decimal digits.

For simplicity, it is proposed to only consider a single (signed) NaN that is defined by numeric_limits<FPType>::quiet_NaN, and a single (signed) infinity value.

```
if (finite) // Normal floating-point value – neither NaN nor infinite
{ // C99/C++ TR1 detection of (non-)finiteness.
  // Output normal decimal digit string (usual formatting).
}
else
{ // Not finite.
  if (isnan())  // C99/C++ TR1 detection of any of the many possible NaNs.
  { // is NaN.
    // Output some Standard string,
    numpunct.NaN_name(); // See below.
  // for which “NaN” would seem simple, concise and short.
 }
else
{ // Is infinity.
  // Output some Standard string,
    numpunct.infinity_name();  // See below.
  // for which “Inf” would seem simple, concise and short.
}
}
```

It is difficult to see any reason to make the strings for NaN and infinity locale dependent, for example in different languages, if their use is likely to be internal.

Some systems might want to print a locale or specific string such as “missing” or “undefined”. This is not too difficult, but they will have to deal with the more difficult problem of recognizing these strings on *input*.

Where to store the string representation of NaN and infinity?

To provide this, I suggest adding

```
string_type NaN_name const;
string_type infinity_name const
```

to std class numpunct after decimal_point… truename and falsename.

What strings should represent NaN and infinity?

C99 already specifies a range of output formats for nans and infs with %f:

ISO/IEC 9899:1999 (E) 7.19.6.1p8:

"A double argument representing an infinity is converted in one of the
styles
[-]inf or [-]infinity - which style is implementation-defined. A
double argument representing a NaN is converted in one of the styles
[-]nan or [-]nan(n-char-sequence) - which style, and the meaning of
any n-char-sequence, is implementation-defined. The F conversion specifier
produces INF, INFINITY, or NAN instead of inf, infinity, or nan,
respectively"

And input formats in 7.20.1.3 (The strtod, strtof, and strtold functions):

"- one of INF or INFINITY, ignoring case
- one of NAN or NAN(n-char-sequenceopt), ignoring case in the NAN part,
where:
n-char-sequence:
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit"

It seems only sensible to adopt this, in that TR1/C++0X presumably will/has incorporated this.

Should the strings be upper, lower or mixed case?
Some systems appear to allow both upper and lower case output, but not the more attractive mixed case “NaN” or “Infinity”.

It is hard to see any reason why the case of all characters should not be ignored on input.

What strings for NaN and inf?
I propose “NaN” and “inf” because they are:

- Simple.
- Short.
- Same length (fewer layout problems).
- Reasonably language neutral.
- Compliant with C99 specification.

Sign?
It is proposed to handle sign entirely separately in the existing standard way, implying the existence of a sign bit in the floating-point representation, and the C99 macro /C++ template function signbit to determine its value. This allows for positive and negative infinity, and for both positive and negative quiet_NaN. (This proposal does not address the question of signed zeros and their representation. It might also be worth adding an additional ‘flag’ to say of these can be output and/or input?)

Why permit negative NaNs?
Use of signbit is necessary to permit + and - infinity, and so I feel it may have some use to permit a tiny bit more information about a NaN to allow +NaN and -NaN. No pretence is made to any mathematical meaning - <u>only that applications can make use of this bit in any way they chose</u>. Nor is there an expectation that it will be portable between OSes. Nor is there an expectation of portability with other applications unless they chose the same interpretation of NaN signs).

For example, suppose that one wishes to distinguish between 'missing' values - no data input by the user, and values that have produced a NaN by some mal-computation. One could, for example, choose to signal 'missing' with -NaN and 'bad' with +NaN.

Is NaN the best way of indicating ‘missing’ values?

Could there be a better choice to indicate that a value is ‘missing’?
(Bearing in mind that NaN is usually the result of a computation rather than just ‘never existed’).

What is expected of NaNs?

The C99 Rational 2003 (reference 3) says:

“The primary utility of quiet NaNs, as stated in IEC 60559, “to handle otherwise intractable situations, such as providing a default value for 0.0/0.0,” is supported by C99.

Other applications of NaNs may prove useful. Available parts of NaNs have been used to encode auxiliary information, for example about the NaN's origin. Signaling NaNs might be candidates for filling uninitialized storage; and their available parts could distinguish uninitialized floating objects. IEC 60559 signaling NaNs and trap handlers potentially provide hooks for maintaining diagnostic information or for implementing special arithmetics.

However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways.
The IEC 60559 floating-point standard recommends that NaNs propagate; but it does not require this and not all implementations do. And the floating-point standard fails to specify the contents of NaNs through format conversion.

Making signaling NaNs predictable imposes optimization restrictions that anticipated benefits don't justify. For these reasons this standard does not define the behavior of signaling NaNs nor specify the interpretation of NaN significands.¨

Un-normalized and Denormalized Numbers

This proposal does not address the potential problems with input and output of both un-normalized numbers (all the very many equivalent ways of specifying numbers with differing exponents and decimal point positions called 'cohorts' in 754r) and denormalized numbers (those that cannot be represented with the full conventional significand precision).

Draft of Proposed Revised Text for

18.2.1.1 template class numeric_limits

TODO!