N2005 A maximum significant decimal digits value for the C++0x Standard Library Numeric limits

Document number: JTC 1/SC22/WG21/N2005=06-0075
Date: 2006-04-11, version 4
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 Previous version of this document: JTC 1/SC22/WG21/N1822=05-0082, 2005-06-14, version 3.
4 A Proposal to add maximum significant decimal digits macros to the C Standard Library.
Document number: JTC 1/SC22/WG14/N1171, Date: 2006-04-04, version 2
(a revision of WG14/N1151 until sent in the next WG14 mailing, is available at:
http://www.hetp.u-net.com/public/N1171%20max%20significant%20decimal%20digits%20MACROs.pdf
aka http://tinyurl.com/f982q
5 C ISO/IEC 9899:1999.

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?

C99[ISO:9899] provides numeric limits 18.2.1 including

```
numeric_limits<FloatingPoint Type>::digits10
```

also available (and often implemented using) via C macros FLT_DIG, DBL_DIG, LDBL_DIG.

The member stores the number of decimal digits that the type can represent without change.

In effect, it is the number of decimal digits GUARANTEED to be correct (after rounding).

While useful, this does not provide another value, often more useful, the number of potentially significant decimal digits that the type can represent. This number of decimal digits is necessary to avoid misleading display of two floating-point numbers which only differ by one or a few least significant bits.

For example, if using IEEE 754/IEC559 32-bit floating-point float values, and numeric_limits<float>::digits10 is 6,

So a number declared as

```
float f = 3.145900F;
```

might be displayed, by setting cout.precision(6) as

```
“3.14590”
```

But `nextafterf(3.145900F, 1.)`, a single bit different, and so definitely not equal, will also display as “`3.14590`” so log files may display a most misleading, and unhelpful, output like

```
“3.14590” != “3.14590”
```

Whereas if the proposed numeric_limits<float>::max_digits10 which is 9 is used, the output

```
“3.14590001” != “3.14590025”.
```

that is much less confusing, especially to the majority of readers whose understanding of floating-point accuracy limitations is incomplete.

This max_digits10 number is also the number of decimal digits required to avoid loss of accuracy when converting to a string and back to a floating-point representation, for example during serialization.

For example (W. Kahan page 4)

"If a decimal string with at most 6 decimal digits is converted to float, and then converted back to the same number a significant digits, then the final string should match the original.

If a float is correct to a decimal string with at least 9 significant decimal digits, and then converted back to float, then the final number must match the original."

(Although the C++ Standard is not clear if it is a requirement that a round trip like

```
float a = 1.F;
float aa = 0;
std::stringstream s;
s.precision(9); // max_digits10 for 32-bit IEE754 float
s << std::scientific << a;
s >> aa;
assert (a != aa);
```

or simply a good quality implementation. This could usefully be clarified).

2 Impact on the C and C++ Standards

This is a pure addition to existing numeric_limits class. It does not require any language change, nor any change to the existing values provided by numeric_limits.

3 Design Decisions

3.1 Implementation

Using the formulae provided by Kahan[2], provided

```
 numeric_limits< T >::is_specialized && std::numeric_limits< T >::radix == 2
 std::numeric_limits<float>::digits == 24  //the number of significand bits
std::numeric_limits<float>::digits10 == 6 // Guaranteed digits
```

```
floor(float_significand_digits -1) *  log10(2) == 6
ceil(1 + float_significand_digits *  log10(2) == 9
```

Note that a C++ compiler will **NOT** evaluate this at compile time, but an WILL perform an integer division, so 301/1000 is already widely used as an approximation for log10(2.).

```
float const log10Two = 0.30102999566398119521373889472449F; // log10(2.)
```

301/1000 is a good approximation to log10(2), and as noted in the C macros proposal[4], will not cause overflow using 16-bit integers and 53 bit significand (53 * 301 = 15953/10000 = 15 + 2 = 17 decimal digits needed).
But for greater significand bit floating point representations, the accuracy of this approximation may not be good enough, so provided int representation is at least 32-bit, then the ratio 30103UL/100000UL will ensure that for more than 265 significand bits.

So it is convenient to use the an integer formula which **can** be calculated at compile time:

If std::numeric_limits<int>::digits == 16 then
    2 + std::numeric_limits<Target>::digits * 301/1000;

If std::numeric_limits<int>::digits >= 32 then
    2 + std::numeric_limits<Target>::digits * 30103UL/100000UL;

This formula may also be used for integer types, including built-in types.

C Macros

C99[5] already has the macro

```
DECIMAL_DIG defined as ceil(1+precision*log10(radix)),
```

But this uses the **highest precision available**, usually long double, so using this for float and double would probably result in using more decimal digits than can be significant, so the following floating-point type-specific macros have been proposed:

```
#define FLT_MAXDIG10 (2+(FLT_MANT_DIG * 30103UL)/100000UL)
#define DBL_MAXDIG10 (2+ (DBL_MANT_DIG * 30103UL)/100000UL)
#define LDBL_MAXDIG10 (2+ (LDBL_MANT_DIG * 30103UL)/100000UL)
```

These may provide a convenient way to implement for built-in floating-point types:

```
std::numeric_limits<FPType>::max_digits10;
```

Default

An obvious default value is zero (required by 18.2.1.1 note 2), and all types, especially user-defined types will provide this unless specifically implemented for that type.

User defined Types

A high-precision user-defined type, for example NTL quad_float type using two 64 –bit numbers to provide a 128-bit 106-bit significand could sensibly provide digits10 = 31 and max_digits10 = 33 by using the above formula.

The table below shows values for some floating-point formats.

Floating point Type	**Often used for C/C++ type**	**Total bits**	**Significand bits (+ 1 if an implicit bit)**	**guaranteed decimal digits digits10**	**significant decimal digits max_digits10**
IEEE single	float	32	23 + 1 = 24	6	9
VAXF	float	32	23 + 1 = 24	6	9
IBM G/390 short	float	32	24	6	9
IEEE single extended	?	>=43	>=32	7	11
IEEE double	double	64	52 + 1 = 53	15	17
VAXG	double	64	52 + 1 = 53	15	17
IBM G/390 long	double	64	56	15	17
VAXD	long double	64	56	15	17
IEEE double extended	long double	80	>=64	19	21
Sparc doubleextended (x86)	long double	80	64	18	21
AIX quad	long double	128	106	31	33
NTL quad	‘quad’	128	106	31	33
IBM G/390 extended	long double	128	112	33	35
VAXH	long double	128	112 + 1 = 113	34	36
IEEE quadruple	long double	128	112 + 1 = 113	34	36
Sparc double extended	long double	128	112 + 1 = 113	34	36
signed fractional	?	127	128	38	40
unsigned fractional	?	128	128	38	40
unsigned fractional	?	128	128 + 1 = 129	38	40

Draft of Proposed Revised Text for

18.2.1.1 template class numeric_limits

After

```
static const int digits10 = 0;
```

insert:

```
static const int max_digits10 = 0;
```

After Note 9, insert (and shuffle down the following items):

10 The number of base 10 digits required to ensure that values which differ by only one binary unit in the last place (ulp), or by numeric_limits::epsilon(), are always differentiated.