Specific-Width Floating-Point Types

Paul A. Bristow
Christopher Kormanyos
John Maddock

Table of Contents

Abstract 2
Background 3
Introduction 4
How to specify constants with quad and half precision? 5
Specifying Precision 6
    Existing Specific precision integer types 7
    Proposed new section 7
Interaction with complex 9
Improvemed safety for microcontrollers with an FPU 10
References 11
Version Info 12

ISO/IEC JTC1 SC22 WG21/SG6 Numerics N??? - 2013-4-??

Important

This is NOT an official Boost library.

Note

Comments and suggestions to Paul.A.Bristow pbristow@hetp.u-net.com.

Abstract

It is proposed to add several optional typedefs with specified widths for floating-point types including `float32_t, float64_t _float128_t` (similar to `int64_t` for integer types).

These will be defined in the global and `std` namespaces.

And also to provide additional suffix(es) to specify extended precision constants to suit precisions lower than that of `float` higher than that of `long double`.

The objectives are to:

- Make it easier to use higher-precision.
- Reduce errors in precision.
- Improve portability, reliability and safety.

Background

C++11 supports floating-point calculations with its built-in types `float`, `double`, and `long double` as well as imlementations of numerous elementary and transcendental functions.

A variety of higher transcendental functions of pure and applied mathematics were added to the C++11 libraries via technical report TR1. It is now proposed to fix these into the next C++1Y standard.[1]

Other mathematical special functions are also now proposed, for example, A proposal to add special mathematical functions according to the ISO/IEC 80000-2:2009 standard Document number: N3494 Version: 1.0 Date: 2012-12-19

The Boost.Math library was accepted into Boost several years ago. It implements many of the functions in both documents mentioned above and has become quite widely used.

With the acceptance and release of Boost.Multiprecision that provides much higher precision than built-in `long double` with cpp_dec_float employing a variety of backends including the well-established GNU Multiple Precision Arithmetic Library and GNU MPFR library libraries as well as a full open-license backend developed from the e_float (TOMS Algorithm 910) library by Christopher Kormanyos and John Maddock.

Since Boost.Multiprecision and Boost.Math work seamlessly, allowing a `float_type typedef` to be switched from a built-in type to hundreds of decimal digits; then all the special functions and distributions can be used at any chosen precision.

Other users and domains are finding the need and utility of decimal and binary fixed-point.

Of course, moving away from hardware supported types to software using C++ templates carries a small price at compile-time, and a much bigger price at runtime.

All these development have made C++ much more attractive to the scientific and engeering community, especially those needing higher (or lower) precision for some (if not all) of the calculations, previously the domain covered by computer algebra systems for which the precision can be arbitrary.

[1] Conditionally-supported Special Math Functions for C++14, N3584, Walter E. Brown

Introduction

Since the inceptions of C and C++, the built-in types `float`, `double`, and `long double` have provided a strong basis for floating-point calculations. Optional compiler conformance with IEEE_ floating-point format has generally led to a relatively reliable and portable environment for floating-point calculations in the programming community.

It is, however, emphasized that floating-point adherence to IEEE_ floating-point format is not mandated by the current C++ language standard. Nor does the standard enforce specific widths or precisions of the built-in types `float`, `double`, and `long double`. This can lead to portability problems and reduce reliability and safety.

This situation reveals a need for more standard ways to specify precision. In addition, it is desirable to extend the precision of existing types to lower and higher precisions. The extension to lower precision is expected to simplify and improve efficiency of floating-point implementations on cost-sensitive architectures such as small microcontrollers. The extension to higher precision is useful for large-scale high-performance numerical calculations and should ease the progression to extended precision by providing precision-steps with finer granularity.

All of these improvements should improve portability, reliability, and safety of floating-point calculations in C++ by ensuring that the actual precision of a floating-point type can be exactly determined both at compile-time as well as during the run of a calculation. Strong interest in floating-point types with specified widths has, for example, been expressed on the Boost list discussion of precise floating-point types.

Recent specification of integer types with specified widths in C99, C11, C++11, and C++ draft specification has drastically improved integer algorithm portability and range.

One example of how specific-size integer types have proven to be essential is described by Robert Ramey Usefulness of fixed integer sizes in portability (for Boost serialization library).

“ Fundamental types in C++ are `unsigned char, signed char, unsigned short int, signed short int, ... unsigned long, signed long`. In addition to the above some compilers define `int32_t`, and other as fundamental types. It is a unfortunate accident of history that the nomenclature is confusing. It is an unfortunate original design choice that this size of int, char etc were not defined as a specific number of bits. However at the time there were in common usage machines with 9, 16, 18, 24, 32, 36 and 48 bit words. What else were the authors to do? It is common among programers to define types `int16_t`, ..., etc using the `typedef` facility to map integers of a specific size between machines. This does no harm and can facilitate portability. However it in no way alters the fundamental types that are available on a given platform. ”

The motivations to provide floating-point types with specified widths are analogous to those that led to the introduction of integers with specified widths such as `int8_t`, `int16_t`, `int32_t`, and `int64_t`. The specification of floating-point types with specified widths and adherence to IEEE_ floating-point format can potentially improve the C++ language significantly, especially in the scientific and engineering communities where other languages have found benefit from types that conform exactly to the IEEE_ floating-point format.

(Notes on jargon: Section 22.3 in the book "The C++ Standard Library Extensions", P. Becker, Addison Wesley 2007, ISBN 0-321-41299-0 is called "Fixed-Size Integer Types". Use of the descriptor *fixed* has lead to some confusion. So the descriptor *specific* in conjunction with width is here used to match the wording of C99 and C11 in the sections and subsections describing `stdint.h`.)

How to specify constants with quad and half precision?

Recent discussion on extended precision floating-point types in C++ has also raised the issue of how to specify constant values with a precision greater than `long double`, now signified by the suffix `L`.

One obvious way is to add `Q` or `q` suffixes to signify that a constant has at least 128-bits (about 40 decimal digits) of precision.

There may also be a need for 256-bit (about 80 decimal digits) precision, and perhaps 512-bits (about 155 decimal digits) precision.

At present, the only way to provide constant values is to use a string to extended-precision type conversion.

This `from_string` method is used for Boost.Math, Boost.Multiprecision and GCC libquadmath, for example.

It would also be useful to have a method of interrogating the size of types, similar to that provided by GCC 3.7.2 Common Predefined Macros, for example, `__SIZEOF_LONG_DOUBLE__` (but is not defined for `__float128` nor `__float80`)

We refer to floating-point types with fixed precision such as 24, 53, 113 or more binary significand digits, (and possibly even extending beyond these to potential multiprecision types).

These are defined in IEEE Standard for Floating-point Arithmetic, IEEE Std 754-2008.

There are detailed descriptions at IEEE_ floating-point format, with more detailed descrptions of each type at IEEE half-precision floating-point format, IEEE single-precision floating-point format, IEEE double-precision floating-point format, Quadruple-precision floating-point format, and IEEE 754 extended precision formats and x86 80-bit Extended Precision Format and these correspond to the proposed types below `float16_t` ....

TBD by Chris: Suffix for half-precision.

Specifying Precision

One could envision two ways to name the fixed-precision types:

- `float24_t, float53_t, float113_t, ...`
- `float32_t, float64_t, float128_t, ...`

The first set above is intuitively coined from IEE754:2008. It is also consistent with the gist of `std::uint32_t`, et al in so far as the number of binary digits of *significand* precision is contained within the name of the data type.

On the other hand, the second set using the size of the *whole type* may probably seem more intuitive to users. The exact layout and number of significand and exponent bits can be confirmed as IEEE754 by checking `std::numeric_limits<type>::is_iec559 == true`.

With the availability of Boost.Multprecision, C++ programmers can now easily switch to using floating-point types that give far more decimal digits of precision (hundreds) than the built-in types `float`, `double` and `long double`.

And portability is also reduced. For example, suppose we wish to achieve a precision higher than the most common IEEE 64-bit floating-point type supported by the X86 chipsets normally used for double. http://en.wikipedia.org/wiki/Double_precision providing a precision of between 15 to 17 decimal digits.

The options for long double are many.

At least one popular compiler treats `long double` exactly as `double` (as permitted by the C++ Standard which does not prescribe the precision for any floating-point (or integer) types, leaving them to be implementation-defined).

However the Intel X8087 chipset does do calculations using internal 80-bit registers, increasing the significand from 53 to 63 bits, and gaining about 3 decimal digits precision from 18 and 21.

Some hardware, for example Sparc, provides a 128-bit quadruple precision floating-point chip.

As of gcc 4.3, a quadruple precision is also supported on x86, but as the nonstandard type `__float128` rather than `long double`.

Darwin long double uses a double-double format developed first by Keith Briggs. This gives about 106-bits of precision (about 33 decimal digits) but has rather odd behaviour at the extremes making implementation of `std::numeric_limits<>::epsilon()` problematic.

Clang uses a similar technique

```
#ifdef __clang__
  typedef struct { long double x, y; } __float128;
#endif
```

as described in Clang float128.

If we wish to ensure that we use all 80 bits available from Intel 8087 chips to calculate Extended precision we would use a `typedef float80_t`.

If the compiler could not generate code this type directly, then it would substitute software emulation, perhaps using a Boost.Multi-precision type `cpp_dec_float_21`.

Similarly if a quadruple precision of 16-byte 128-bit Quadruple-precision floating-point format is desired, the specification of `float128_t` will either direct the compiler to generate code using the hardware, or it will do this using software emulation. This might be generated by the compiler for GCC or delegated to a `cpp_bin_float_128` type (under development for Boost.Multipre-cision).

Existing extended precision types

1. GNU C supports additional floating types, `__float80` and `__float128` to support 80-bit (XFmode) and 128-bit (TFmode) floating types.

2. Extended or Quad IEEE FP formats by Intel Intel64 mode on Linux (V12.1) provides 128 bit `long double` in C, however it appears that it only provides computation at 80-bit format giving 64-bit significand precision, and other bits are just padding.

3. Intel FORTRAN REAL*16 is an actual 128-bit IEEE quad, emulated in software. But "I don't know of any plan to implement full C support for 128-bit IEEE format, although evidently ifort has support libraries." This is equivalent to the proposed float128_t type.

4. The 360/85 and follow-on System/370 added support for a 128-bit "extended" IBM extended precision formats. These formats are still supported in the current design, where they are now called the "hexadecimal floating point" (HFP) formats.

Existing Specific precision integer types

18.4 Integer types [cstdint]

18.4.1 Header <cstdint> synopsis [cstdint.syn]

```
namespace std
{
  typedef signed integer type int8_t; // optional
  typedef signed integer type int16_t; // optional
  typedef signed integer type int32_t; // optional
  typedef signed integer type int64_t; // optional
}
```

Proposed new section

Add the following text to <cstdint>

Note

It is not obvious where these `typedefs` should reside. The obvious place is `<cstdint>` but `int` implies integer types. (or a new <cstdfloat>?)

18.4? Arithmetic types [cstdfloat] (or cstdarith] 18.4.2? Header <cstdfloat> synopsis [cstdfloat.syn]

```
namespace std {
  typedef signed floating-point type float_16_t;  // optional.
  typedef signed floating-point type float_32_t;  // optional.
  typedef signed floating-point type float_64_t;  // optional.
  typedef signed floating-point type float_80_t;  // optional.
  typedef signed floating-point type float_128_t; // optional.
  typedef signed floating-point type float_256_t; // optional.
  typedef signed floating-point type floatmax_t;  // optional.

  typedef signed floating-point type float_least16_t;  // optional.
  typedef signed floating-point type float_least32_t;  // optional.
  typedef signed floating-point type float_least64_t;  // optional.
  typedef signed floating-point type float_least80_t;  // optional.
  typedef signed floating-point type float_least128_t; // optional.
  typedef signed floating-point type float_least256_t; // optional.

  typedef signed floating-point type float_fast16_t;  // optional.
  typedef signed floating-point type float_fast32_t;  // optional.
  typedef signed floating-point type float_fast64_t;  // optional.
  typedef signed floating-point type float_fast80_t;  // optional.
  typedef signed floating-point type float_fast128_t; // optional.
  typedef signed floating-point type float_fast256_t; // optional.
} // namespace std
```

It is not proposed to make any change to `std::numeric_limits`.

It is obviously highly desirable that `std::numeric_limits` is specialized for all floating-point types. And experience with Boost.Math and Boost.Multiprecision is that the normal set of trig and others useful functions is also essential to make the type useful in real-life.

Programs can then use this to determine if a floating-point type is IEEE 754 using `std::numeric_limits<>::is_iec559`.

Interaction with complex

TBD: Describe interaction with <complex>

Improvemed safety for microcontrollers with an FPU

TBD by Chris: Describe recent confidential meetings with tier-one silicon suppliers and the relevant problems discussed therein.

TBD: Cite these as personal communications.

TBD by Chris: Explain how standards adherence and specified widths can help to solve these problems by improving reliability and safety.

TBD ba Chris: Add an example and remarks on functional safety and any relevant citations from to ISO/IEC 26262.

References

isocpp.org C++ papers and mailings

C++ Binary Fixed-Point Arithmetic, N3352, Lawrence Crowl

Proposal to Add Decimal Floating Point Support to C++, N3407 Dietmar Kuhl

The C committee is working on a Decimal TR as TR 24732. The decimal support in C uses built-in types _Decimal32, _Decimal64, and _Decimal128. 128-bit decimal floating point in IEEE 754:2008

lists binary16, 32, 64 and 128

(and also decimal 32, 64, and 128) IEEE Std 754-2008

IEEE Standard for Floating-point Arithmetic, IEEE Std 754-2008

How to Read Floating Point Numbers Accurately, William D Clinger

Conditionally-supported Special Math Functions for C++14, N3584, Walter E. Brown

Walter E.Brown, Opaque Typedefs

Specification of Extended Precision Floating-point and Integer Types, Christopher Kormanyos, John Maddock

X8087 notes

Version Info

Last edit to Quickbook file precision.qbk was at 12:20:50 PM on 2013-Mar-23.

Tip

This should appear on the pdf version (but may be redundant on a html version where the last edit date is on the first (home) page).

Warning

Home page "Last revised" is GMT, not local time. Last edit date is local time.