Specific-Width Floating-Point Types

Paul A. Bristow
Christopher Kormanyos
John Maddock

Table of Contents

Abstract 2
Background 3
Introduction 4
The proposed types and potential extensions 5
How to specify constants with quad and half precision? 6
Specifying Precision 7
    Existing Specific precision integer types 8
    Proposed new section 8
Interaction with complex 10
Improved efficiency and robustness for microcontrollers 11
References 12
Version Info 13

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 to the C++ standard several optional typedefs for floating-point types with specified widths. In particular, the optional types include `float16_t`, `float32_t`, `float64_t`, `float128_t`, and their corresponding fast and least types. The optional floating-point types are to conform with the corresponding types `binary16`, `binary32`, `binary64`, and `binary128` described in IEEE_ floating-point format.

The optional floating-point types with specified widths are to be contained in a new standard library header `<cstdfloat>`. Any of the optional floating-point types with specified widths included in the implementation must have full support for the functions in `<cmath>` and seamlessly interoperate with `<limits>` and `<complex>`. The proposed new floating-point types with specified widths will be defined in the global and `std` namespaces.

It is also proposed to provide additional suffix(es) to specify constants to suit precision lower than that of `float` and precision higher than that of `long double`.

Floating-point types with specified widths are expected to significantly improve clarity of code and portability of floating-point calculations. Analogous improvements for integer calculations were recently achieved via standardization of integer types having specified width such as `int8_t`, `int16_t`, `int32_t`, and `int64_t`.

The main objectives of this proposal are to:

- Extend the range of floating-point precision.
- Reduce errors in precision.
- Improve clarity of coding.
- Improve portability, reliability and safety.

Background

Support for mathematical facilities and specialized number types in C++ is progressing rapidly. Currently, 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.

There is also progress in C++ in the area of multiprecision, including support of multiprecision floating-point numbers. In particular, the acceptance and release of Boost.Multiprecision provides much higher precision than built-in `long double` with its cpp_dec_float data type. Boost.Multiprecision has a flexible front-end that employs a variety of backends to implement multiprecision floating-point types including the well-established GNU Multiple Precision Arithmetic Library and GNU MPFR library libraries as well as a full open-license backend that originates from the e_float (TOMS Algorithm 910) library by Christopher Kormanyos and John Maddock.

Since Boost.Multiprecision and Boost.Math work seamlessly, a `float_type typedef` can be used to switch from a built-in type to hundreds of decimal digits. This allows all the special functions and distributions in Boost.Math to 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 potentially a much bigger price at runtime. Nonetheless, the new numerical types have wide ranges of application required in numerous programming domains.

All these development have made C++ much more attractive to the scientific and engeering community, especially those needing mathematical functions and higher (or lower) precision for some of their calculations. Previously these domains were predominantly covered by computer algebra systems.

[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 specify the widths or precisions of its built-in types `float`, `double`, and `long double`. This can lead to portability problems, introduce poor efficiency on cost-sensitive microcontroller architectures, and reduce reliability and safety.

This situation reveals a need for a standard way to specify precision. It is also desirable to extend the precision of existing types to both 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 transition to multiprecision by providing built-in types with progressing precision of 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, recently been expressed on the Boost list discussion of precise floating-point types.

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

One example of how integer types having specified widths 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`.)

The proposed types and potential extensions

The core of this proposal is based on the types `float16_t`, `float32_t`, `float64_t`, `float128_t`, and their corresponding least and fast types. These floating-point types have specified widths and they are to conform with the corresponding types `binary16`, `binary32`, `binary64`, and `binary128` specified in IEEE_ floating-point format. In particular, `float16_t`, `float32_t`, `float64_t`, and `float128_t` correspond to floating-point types with 11, 24, 53, and 113 binary significand digits, respectively. These are defined in IEEE_ floating-point format, and there are 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.

There may be a need for octuple-precision float, in other words `float256_t` with about 240 binary significand digits of precision. In addition, a `float512_t` type with even more precision may be considered as a option. Beyond these, there may be potential extension to multiprecision types in the future.

The popular Intel X8087 chipset architecture supports a 10-byte floating-point format. So it may be useful to provide optional support for `float80_t`. There is no analogous type for `float80_t` in IEEE_ floating-point format, but the implementation already exists in practice.

At present, the only way to provide a literal constant value with precision exceeding the precision of `long double` is to use a string in association with extended-precision type conversion. For example, the `from_string` method is used for this purpose in Boost.Math, Boost.Multiprecision and GCC libquadmath.

Along these lines, the herein proposed floating-point types with specified widths should be copy assignable and copy constructable from string literal constants. This requires changes to the core language including the addition of new literal constant suffixes, as described below.

It would also be useful to have a method of querying the size of types, similar to that provided by GCC 3.7.2 Common Predefined Macros, for example, `__SIZEOF_LONG_DOUBLE__`. But similar macros are not defined for `__float128` nor for `__float80`.

How to specify constants with quad and half precision?

The standard specifies that the type of a floating-point literal is double unless explicitly specified by a suffix. The standard continues by specifying that the suffixes `f` and `F` specify `float`, and the suffixes `l` and `L` specify `long double`.

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` or `l`. One possible way is to add `Q` or `q` suffixes to signify that floating-point literal has quadruple precision.

Code using the `Q` suffix scheme is shown in the sample below.

```
#include <cstdfloat>

constexpr std::float128_t pi = 3.14159265358979323846264338327950288419720Q;
```

For half-precision floating-point literals, the suffix `H` or `h` could be used. One potential suffix for octuple-precision floating-point literals is `O` or `o`.

For example,

```
#include <cstdfloat>

constexpr std::float16_t euler_gamma = 0.577216H;
```

Higher precisions also require construction from floating-point literals. As the list of available suffixes dwindles, however, available suffixes might run out and the myriad of suffixes may become confusing. Floating-point literals for precisions higher than quadruple precision, then, might be better served with construction from string literals.

An alternative suffix scheme could use hybrid suffixes composed of, say, the letter `F` or `f` which stands for floating-point, to which the specified width of the type is appended, for example `F16`, `F32`, `F64`, `F128`, etc. Code using the `F128` suffix scheme is shown in the sample below.

```
#include <cstdfloat>

constexpr std::float128_t pi = 3.1415926535897932384626433832795028841972F128;

constexpr std::float16_t euler_gamma = 0.577216F16;
```

This suffix scheme is unequivocal and it can be easily extended to unlimited precision. On the other hand, it may be difficult for programmers to separate the character part of the suffix from its numerical part when analyzing source code. For example, it is particularly difficult to resolve the suffix in `euler_gamma` above, since the `16F16` obscures the `F`.

Specifying Precision

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

- `float11_t, float24_t, float53_t, float113_t, ...`
- `float16_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 integer types with specified width such as `int64_t`, 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 with the size of the *whole type* contained within the name may be 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.Multiprecision 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.Multiprecision).

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 a new header <cstdfloat> to the standard library. 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. Here, we prefer the new header <cstdfloat>.

18.4? Floating-Point Types Having Specified Width 18.4.2? Header <cstdfloat> synopsis [cstdfloat.syn]

```
namespace std {
  typedef signed floating-point type float16_t;   // optional.
  typedef signed floating-point type float32_t;   // optional.
  typedef signed floating-point type float64_t;   // optional.
  typedef signed floating-point type float80_t;   // optional.
  typedef signed floating-point type float128_t;  // optional.
  typedef signed floating-point type float256_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, nonethelsee, mandatory that `std::numeric_limits` is specialized for all floating-point types. Thus their must be explicit template specializations for all of the optional floating-point types with specified precision that are included in the implementation. This will ensure that programs can use the established `std::numeric_limits<>::is_iec559` member to determine if a floating-point type conforms with IEEE_ floating-point format.

Experience with Boost.Math and Boost.Multiprecision has shown that the normal set of elementary and transcendental functions (and possibly additional higher transcendental functions) is also essential to make the type useful in real-life computational regimes. Therefore, the implementation must provide support for the mathematical functions in the `std` namespace (TBD: Chris list functions) for each of the floating-point types with specified precision included in the implementation.

Interaction with complex

TBD: Describe interaction with <complex>

Improved efficiency and robustness for microcontrollers

TBD by Chris: Describe cost-sensitive floating-point regime. TBD by Chris - add reference to your book!

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 09:21:33 AM on 2013-Mar-24.

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.