Floating-Point Typedefs Having Specified Widths - N3626

Paul A. Bristow
Christopher Kormanyos
John Maddock

Table of Contents

Abstract 2
Introduction 3
The proposed typedefs and potential extensions 4
Handling floating-point literals 7
Place in the standard 8
Interoperation with <cmath> and special functions 10
Interoperation with <limits> 11
Interoperation with <complex> 12
Specifying 128-bit precision 13
Extending to lower precision 14
The context among existing implementations 15
References 16
Version Info 17

ISO/IEC JTC1 SC22 WG21/SG6 Numerics N3626 - 2013-04-05

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

Abstract

It is proposed to add to the C++ standard **optional floating-point `typedefs` having specified widths**. The optional `typedefs` include `float16_t`, `float32_t`, `float64_t`, `float128_t`, their corresponding least and fast types, and the corresponding maximum-width type. These are to conform with the corresponding specifications of `binary16`, `binary32`, `binary64`, and `binary128` in IEEE_ floating-point format.

The optional floating-point `typedefs` having specified widths are to be contained in a **new standard library header `<cstdfloat>`**.

They will be defined in the `std` namespace.

New C-style macros to facilitate initialization of the optional floating-point `typedefs` having specified widths from floating-point literal constants are proposed.

It is not proposed to make any mandatory changes to `<cmath>`, special functions, `<limits>`, or `<complex>`.

The main objectives of this proposal are to:

- Extend the benefits of specified-width typedefs for integer types to floating-point types.
- Improve floating-point safety and reliability by providing standardized `typedefs` that behave identically on all platforms.
- Optionally extend the range of floating-point to lower and to higher precision.
- Provide a Standard way of specifying 128-bit precision.

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.

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 implementations 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

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, precisions and layouts 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 floating-point precision in C++.

Providing optional floating-point `typedefs` having specified widths is expected to significantly improve portability, reliability, and safety of floating-point calculations in C++. Analogous improvements for integer calculations were recently achieved via standardization of integer types having specified widths such as `int8_t`, `int16_t`, `int32_t`, and `int64_t`.

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

The proposed typedefs and potential extensions

The core of this proposal is based on the optional floating-point `typedefs` `float16_t`, `float32_t`, `float64_t`, `float128_t`, their corresponding least and fast types, and the corresponding maximum-width type.

In particular,

```
// Sample partial synopsis of <cstdfloat>

namespace std
{
  typedef float       float32_t;
  typedef double      float64_t;
  typedef long double float128_t;
  typedef float128_t  floatmax_t;

  // ... and the corresponding least and fast types.
}
```

These proposed optional floating-point `typedefs` are to conform with the corresponding specifications of `binary16`, `binary32`, `binary64`, and `binary128` 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 descriptions 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.

Here, we specifically mean equivalence of the following.

```
float16_t  == binary16;
float32_t  == binary32;
float64_t  == binary64;
float128_t == binary128;
```

This equivalence results in far-reaching benefits.

It means that floating-point software written with `float16_t`, `float32_t`, `float64_t`, and `float128_t` should behave identically when used on any platform with any implementation that correctly supports the `typedefs`.

It also creates the opportunity to implement quadruple-precision (Quadruple-precision floating-point format) in a specified, and therefore portable, fashion.

One could envision two ways to name the proposed optional floating-point `typedefs` having specified widths:

- `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 having specified widths 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. Here, we prefer the latter naming scheme.

No matter what naming scheme is used, the exact layout and number of significand and exponent bits can be confirmed as IEEE754 by checking `std::numeric_limits<type>::is_iec559 == true`, and the byte order. Little-endian IEEE754 architectures now predominate.

Note

IEEE_ floating-point format prescribes a method of precision extension, that allows for conforming types other than `binary16`, `binary32`, `binary64`, and `binary128`. This makes it possible to extend floating-point precision to both lower and higher precisions in a standardized way using implementation-specific `typedef`s that are not derived from `float`, `double`, and `long double`.

Note

Paragraph 3.7 in IEEE_ floating-point format states: **Language standards should define mechanisms supporting extendable precision for each supported radix**. This proposal embodies a potential way for C++ to adhere to this requirement.

Note

IEEE_ floating-point format does not specify the byte order for floating-point storage (the so-called endianness). This is the same situation that prevails for integer storage in C++.

We will now consider various examples that show how implementations might introduce some of the optional floating-point `typedef`s having specified widths into the `std` namespace.

An implementation has `float` and `double` corresponding to IEEE754 `binary32`, `binary64`, respectively. This implementation could introduce `float32_t`, `float64_t`, and `floatmax_t` into the `std` namespace as shown below.

```
// In <cstdfloat>

namespace std
{
  typedef float     float32_t;
  typedef double    float64_t;
  typedef float64_t floatmax_t;
}
```

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

Consider an implementation for a supercomputer. This platform has `float`, `double`, and `long double` corresponding to IEEE754 `binary32`, `binary64`, and `binary128`, respectively. In addition, this implementation has floating-point types with octuple-precision and hextuple-precision. The implementation for this supercomputer could introduce its optional floating-point `typedef`s having specified widths into the `std` namespace as shown below.

```
// In <cstdfloat>

namespace std
{
  typedef float                 float32_t;
  typedef double                float64_t;
  typedef long double           float128_t;
  typedef floating-point type   float256_t;
  typedef floating-point type   float512_t;
  typedef float512_t            floatmax_t;
}
```

A cost-sensitive 8-bit microcontroller platform without an FPU does not have sufficient resources to support the eight-byte, 64-bit `binary64` type in a feasible fashion. An implementation for this platform can, however, support half-precision `float16_t` and single-precision `float32_t`. This implementation could introduce its optional floating-point `typedefs` having specified widths into the `std` namespace as shown below.

```
// In <cstdfloat>

namespace std
{
  typedef floating-point type  float16_t;
  typedef float                float32_t;
  typedef float32_t            floatmax_t;
}
```

The popular Intel X8087 chipset architecture supports a 10-byte floating-point format. It may be useful to extend the optional support to `float80_t`. Several implementations using x86 Extended Precision Format already exist in practice.

Consider an implementation that supports single-precision `float`, double-precision `double`, and 10-byte `long double`. This implementation could introduce its optional `typedefs` `float32_t`, `float64_t`, `float80_t`, and `floatmax_t` into the `std` namespace as shown below.

```
// In <cstdfloat>

namespace std
{
  typedef float       float32_t;
  typedef double      float64_t;
  typedef long double float80_t;
  typedef float80_t   floatmax_t;
}
```

Handling floating-point literals

We will now examine how to use floating-point literal constants in combination with the optional floating-point `typedefs` having specified widths. This will be done in a manner analagous to the mechanism specified for integer types having specified widths, in other words using C-style macros.

The header `<cstdfloat>` should contain all necessary C-style function macros in the form shown below.

```
FLOAT{16 32 64 80 128 256 512 MAX}_C
```

The code below, for example, initializes a constant `float128_t` value using one of these macros.

```
#include <cstdfloat>

constexpr std::float128_t euler = FLOAT128_C(0.57721566490153286060651209008240243104216);
```

The following code initializes a constant `float16_t` value using another one of these macros.

```
#include <cstdfloat>

constexpr std::float16_t euler = FLOAT16_C(0.577216);
```

In addition, the header `<cstdfloat>` should contain all necessary macros of the form:

```
FLOAT_[FAST LEAST]{16 32 64 80 128 256 512}_MIN
FLOAT_[FAST LEAST]{16 32 64 80 128 256 512}_MAX
FLOATMAX_MIN
FLOATMAX_MAX
```

These macros can be used to query the ranges of the optional floating-point `typedefs` having specified widths at compile-time. For example,

```
#include <limits>
#include <cstdfloat>

static_assert(FLOATMAX_MAX > (std::numeric_limits<float>::max)(),
              "The iec559 floating-point range is too small.");
```

Place in the standard

The proper place for defining the optional floating-point `typedef`s having specified widths should be oriented along the lines of the current standard. Consider the existing specification of integer `typedef`s having specified widths in C++11. A partial synopsis is shown below.

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
}

// ... and the corresponding least and fast types.
```

It is not immediately obvious where the optional floating-point `typedef`s having specified widths should reside. One potential place is `<cstdint>`. The `int`, however, implies integer types. Here, we prefer the proposed new header `<cstdfloat>`.

We propose to add a new header `<cstdfloat>` to the standard library. The header `<cstdfloat>` should contain all optional floating-point `typedef`s having specified widths included in the implementation and the corresponding C-style macros shown above.

Section 18.4 could be extended as shown below.

18.4? Integer and Floating-Point Types Having Specified Widths

18.4.1 Header <cstdint> synopsis [cstdint.syn]

18.4.2? Header <cstdfloat> synopsis [cstdfloat.syn]

```
namespace std
{
  typedef floating-point type float16_t;   // optional.
  typedef floating-point type float32_t;   // optional.
  typedef floating-point type float64_t;   // optional.
  typedef floating-point type float80_t;   // optional.
  typedef floating-point type float128_t;  // optional.
  typedef floating-point type float256_t;  // optional.
  typedef floating-point type float512_t;  // optional.
  typedef floating-point type floatmax_t;  // optional.

  typedef floating-point type float_least16_t;  // optional.
  typedef floating-point type float_least32_t;  // optional.
  typedef floating-point type float_least64_t;  // optional.
  typedef floating-point type float_least80_t;  // optional.
  typedef floating-point type float_least128_t; // optional.
  typedef floating-point type float_least256_t; // optional.
  typedef floating-point type float_least512_t; // optional.

  typedef floating-point type float_fast16_t;  // optional.
  typedef floating-point type float_fast32_t;  // optional.
  typedef floating-point type float_fast64_t;  // optional.
  typedef floating-point type float_fast80_t;  // optional.
  typedef floating-point type float_fast128_t; // optional.
  typedef floating-point type float_fast256_t; // optional.
  typedef floating-point type float_fast512_t; // optional.
}
```

Interoperation with <cmath> and special functions

It is not proposed to make any mandatory changes to `<cmath>` or special functions.

Any of the optional floating-point `typedef`s having specified widths that are `typedef`ed from the built-in types `float`, `double`, and `long double` should automatically be supported by the implementation's existing `<cmath>` and special functions.

Implementation-specific optional floating-point `typedef`s having specified widths that are not derived from `float`, `double`, and `long double` can optionally be supported by `<cmath>` and special functions. This is considered an implementation detail.

Note

Support of elementary functions (and possibly some special functions, even where only optional) can be very useful for real-life computational regimes.

Interoperation with <limits>

It is not proposed to make any mandatory changes to `<limits>`.

Any of the optional floating-point `typedef`s having specified widths that are `typedef`ed from the built-in types `float`, `double`, and `long double` should automatically be supported by the implementation's existing `<limits>`.

Implementation-specific optional floating-point `typedef`s having specified widths that are not derived from `float`, `double`, and `long double` can optionally be supported by `<limits>`. This is considered an implementation detail.

Note

Support for `<limits>`, even where optional, can be very useful, especially for portability. This allows programs to query the floating-point limits at compile-time and use, among other things, `std::numeric_limits<>::is_iec559` to verify conformance with IEEE_ floating-point format.

Note

Each of the optional floating-point `typedef`s having specified widths can only have `true` for the value of `std::numeric_limits<>::is_iec559` if its underlying type (be it `float`, `double`, `long double` or an implementation-dependent type) conforms with one of `binary16`, `binary32`, `binary64`, or `binary128`, or the prescribed method of precision extension in IEEE_ floating-point format.

Interoperation with <complex>

It is not proposed to make any mandatory changes to `<complex>`.

Any of the optional floating-point `typedef`s having specified widths that are `typedef`ed from the built-in types `float`, `double`, and `long double` should automatically be supported by the implementation's existing `<complex>`.

Implementation-specific optional floating-point `typedef`s having specified widths that are *not derived from `float`, `double`, and `long double`* can optionally be supported by `<complex>`. This is considered an implementation detail.

Specifying 128-bit precision

The proposed `typedef float128_t` provides a standardized way to specify quadruple-precision (Quadruple-precision floating-point format) in C++.

On powerful PCs and workstations, implementation-specific versions of `long double` as well as various floating-point extensions to 128-bit have been treated in a variety of ways. This has resulted in numerous portability problems.

The Intel X8087 chipset is capable of performing calculations with internal 80-bit registers. This increases the width of the significand from 53 to 63 bits, thereby gaining about 3 decimal digits precision and extending it from 18 and 21. If an implementation has a type that uses all 80 bits from this chipset to calculate Extended precision, it could could use an optional `typedef` of this type to `float80_t`.

Some hardware, for example Sparc, provides a full 128-bit quadruple-precision floating-point chip. An implementation for this kind of architecture might already have a built-in type corresponding to `binary128`, and this type could be optionally `typedef`ed to `float128_t`.

GCC has recently developed quadruple-precision support on a variety of platforms using GCC libquadmath. However, the implementation-specific type `__float128` is used rather than `long double`. These implementations could optionally `typedef __float128` to `float128_t` in addition to any other optional `typedef`s.

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 behavior at the extremes making implementation of `std::numeric_limits<>::epsilon()` problematic.

It may be useful if future implementations for powerful PCs and workstations strive to make implementation-specific extensions to 128-bit floating-point or the built-in type `long double` equivalent to `binary128`, and to include the corresponding `typedef` to `float128_t`.

Some architectures have hardware support for this. Those lacking direct hardware support can use software emulation.

Survey of 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. Intel C++ provides an internal 128-bit floating-point type called `_Quad`. When the `-Qoption,cpp,--extended_float_type` command line option is supplied, it supports what appears to be an undocumented data type `_Quad`. This type is equivalent to GCC's `__float128`.

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.

Extending to lower precision

Some implementations for cost-sensitive microcontroller platforms support `float`, `double`, and `long double`, and some of these are compliant with IEEE_ floating-point format. Some of these implementations treat `double` exactly as `float`, and even treat `long double` exactly as `double`. This is permitted by the standard which does not prescribe the precision for any floating-point (or integer) types, leaving them to be implementation-defined. On these platforms, the existing floating-point types could optionally be type-defined to `float32_t`. Optional support for an extension to `float16_t` could provide a very useful and efficient floating-point type with half-precision, but reduced range.

Some implementations for cost-sensitive microcontroller platforms also support a 24-bit floating-point type. Here, an extension of the optional floating-point `typedefs` with specified widths could include `float24_t`. This would be equivalent to three-quarter precision floating-point, the layout of which should adhere to the method of precision extension specified in IEEE_ floating-point format.

Some embedded graphics systems use an 8-bit floating-point representation, primarily for storage of pixel information. Here, an extension of the optional floating-point `typedefs` with specified widths could include `float8_t`. This would be equivalent to one-quarter precision floating-point, the layout of which should adhere to the method of precision extension specified in IEEE_ floating-point format.

These potential embedded extensions for cost-sensitive microcontroller platforms are shown in the code sample below

```
// Potential embedded extensions.

namespace std
{
  typedef floating-point type float8_t;    // optional.
  typedef floating-point type float16_t;   // optional.
  typedef floating-point type float24_t;   // optional.
  typedef float               float32_t;   // optional.
}
```

The context among existing implementations

Many existing implementations already support `float`, `double`, and `long double`. In addition, some of these either are or strive to be compliant with IEEE_ floating-point format. In these cases, it will be straightforward to support (at least) a subset of the proposed optional floating-point `typedef`s having specified widths by adding any desired optional type definitions and the corresponding macro definitions.

References

1. isocpp.org C++ papers and mailings
2. C++ Binary Fixed-Point Arithmetic, N3352, Lawrence Crowl
3. Proposal to Add Decimal Floating Point Support to C++, N3407 Dietmar Kuhl
4. 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
5. lists binary16, 32, 64 and 128 (and also decimal 32, 64, and 128)
6. IEEE Std 754-2008
7. IEEE Standard for Floating-point Arithmetic, IEEE Std 754-2008
8. How to Read Floating Point Numbers Accurately, William D Clinger
9. Conditionally-supported Special Math Functions for C++14, N3584, Walter E. Brown
10. Walter E.Brown, Opaque Typedefs
11. Specification of Extended Precision Floating-point and Integer Types, Christopher Kormanyos, John Maddock
12. X8087 notes
13. Intel Extended or Quad IEEE FP formats compiler '-Qoption,cpp,--extended_float_type'

Version Info

Last edit to Quickbook file precision.qbk was at 07:50:56 AM on 2013-Apr-02.

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.