C and C++ Compatibility Study Group Meeting Minutes (Oct 2021 - Special Session on Floating-Point Types)

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2835
SG Meeting Date: 2021-10-06

Wed Oct 06, 2021 at 11:00am EST

Attendees

Aaron Ballman	WG21/WG14	chair
Rajan Bhakta	WG14	scribe
Jim Thomas	(14)	
David Olsen	WG21	
Davis Herring	WG21	
Damian McGuckin		
Gabriel Dos Reis	WG21	
Ilya Burylov	WG21	
Jens Maurer	WG21	
Marius Cornea	(14)	
Matthias Kretz	WG21	
Roberto Bagnara	WG14	
Tom Honermann	WG21	
Hubert Tong	WG21	
Ian McIntosh	(14)	
Walter E. Brown	WG21	
David Hough	(14)	
Thomas Koeppe	WG21	
Michal Dominiak	WG21	

Code of Conduct: follows ISO, IEC, and WG21 CoCs (no current WG14-specific CoC)

Agenda

Discussing the following papers:

WG21 P1467R4 (https://wg21.link/p1467r4) Extended floating-point types and standard names

P1467R4 Extended floating-point types and standard names

Meeting goal:

This will be a joint discussion with the C Floating Point Study Group.

The paper proposes allowing implementations to provide new extended floating-point types in C++. C already allows such types through the integration of TS18661 into C23. The authors would like to have a discussion around the names of the types and the header where the names are defined, and a discussion of where the behavior of these types differ between languages, including implicit conversions and usual arithmetic conversions. We expect to take preference polls on naming decisions.
See P1467R5 (https://wg21.link/P1467R5)

Discussion:

David Olsen: What's going in C23 and what's being proposed in WG21: Names for floating point types. Extended floating-point types is used in C++'s paper which is not the same as what is in the C standard for the same term. The main part of this meeting to discuss is the C compatibility sections of the WG21 paper. For C++: Conversion ranks have standard types with higher ranks than extended types if they have the same format. It is a partial order (not a total order). Ex. IEEE float16 and bfloat16 are non-overlapping. Neither is a subset of the other. Conversions that are potentially lossy require explicit conversion. Like usual, higher conversion rank gives the result type. For unordered, it is ill-formed. Compile error. For C, mostly the same, but a slight difference which will be discussed later. Narrowing conversions: Lower or unordered conversion rank is considered narrowing.For C, the math functions in math.h there are unique functions and with tgmath.h uses magic to give the same name. The C++ standard does not give the real names for the extended floating point types, but if it does, it can also provide standard names for them: std::float{16,32,64,128}_t, bfloat16_t. They cannot be a typedef of a standard type (ex. float, double). This was to avoid the problems like what was there for int and int32_t for example. Suffixes for literals match C. The C and C++ proposals were developed independently with no knowledge of each other. Both ended up being very common with minor incompatibilities. In the areas of differences, the biggest one is names. C23 uses _Float{16,32,64,128} that are optional keywords. C++ likes things in the std namespace. C++ does not usually like _<Capital letter> type naming. Should C++ require the use of the C names?

Naming:

Jens: On the mailing list, std::floatX_t was unfortunate since C defines floatX_t names for different things. We don't want that. This is a point of confusion and needs to be added to the paper.
David O: Yes, I saw that, but didn't get it into this revision of the paper. float_t and double_t are in C that are not necessarily the same as float, double. Also new ones like floatX_t that are not necessarily the same as _FloatX_t.
Jens: Yes, I find that confusing.
David O: Any suggestion to resolve this?
Jens: Drop the _t from the C++ names.
Jim: The _t gives semantic operations the same type as the evaluation range and precision.
Tom: Another point was whether the C names should be in C++ or if the type alias should be directly mapped to the C++ names. I want the C names there to help with tool writers.
David O: I want _Float32 to be available for C++?
Tom: Yes. I don't care if it is a keyword or a typedef. No flexibility allowing another name.
Hubert: The implementations between the typedef and keyword have the typedef situation. You can hide the builtin typedef. This does weird stuff. The only real answer is keyword. In addition of Tom's rationale, I want the C++ std names be the C types. If we allow them to be different you have problems for people writing code.
Jens: You said if you have _F* typedefs they could be hidden?
Hubert: Yes, that happens for things like _Float128.

Jens: There are restrictions on names.
Hubert: It is UB, so yes.
Jens: Then I don't care about that.
David O: Hubert has a point, but it is not a blocker.
David O: I would expect any C++ implementation to use the C names. Do we want the C++ standard to require that?
gdr: I want the std names as they are. I have no sympathy for users messing up with _<Capital> names and they mess up. That is their problem. If the C++ standard requires an _<Capital> is not good. Option 1 is the best for the C++ standard.
Aaron: The C++ standard already has some _<Capital> like _Exit.
gdr: It is not a precedence I want to see carried forward. I understand C has it's constraints.
Hubert: Failing to make the appearance that they are the same thing makes it more questionable whether interlanguage linkage has the same thing. For future C and C++ compatibility, an allergy of _<Capital> is not the best way forward. I know C does have changes to lowercase, and in some cases due to semantic differences it is good to have both. C++ should do the same.
gdr: C++ doesn't define C and vice versa. They have semantics that we hope do it the same way. Implementations do that. It is a QoI problem. The notion that using the same names with different semantics is a weak argument.
Aaron: The thread_local difference has different semantics. C++ with _Thread_local has different semantics than thread_local. _Atomic is another one. The point of this group is to have open communication to reduce differences due to the economic harm.
Jens: Traditionally C++ doesn't define semantics for floating point operations at all, while C does. Like this for IEEE in C. C++ can do something, but likely duplicates what C does. It is unlikely to have a difference. I think strongly we should refer to C for these types due to the IEEE mappings. No one wants to do the work to write it out in C++. I am happy to be proven wrong on this though. We can drop strong hints to have these be compatible with C types. I am disturbed with the implicit conversion differences in C. Why did C choose the new types vs float/double.
David O: We'll get to conversions separately after names. Responding to what Gaby(sp?) said, if code is compiled by C or C++, it should have the same behavior. That is the goal.
Tom: From a tooling perspective, we have to consume header files. Divergence of names makes this harder. If I use a C header, I should be able to use C names without #ifdef.
Hubert: A lot of what I wanted to say was said. When users use a name, they expect a certain semantics. Not being consistent does have real problems where people mistake one situation for another.
gdr: I understand the compat story. We never did exactly what C did. We add, subtract or modify the semantics of what C did. Ex. Const correct. We have a wide range of tools to deal with differences.
Aaron: Having worked at a job with tooling in the past, I can see Tom's point. I am sympathetic to Gaby's point as well.
Tom: We have the complex issue as well. But all existing C++ implementations use _Complex. In practice this will likely happen here as well so why not have the standard reflect that?

Implicit conversions:

David O: If code compiles in both languages it will behave the same. But you can have C code that will fail in C++.
Jens: C doesn't allow BFP <-> DFP implicit conversions.
Rajan: That is correct.
Aaron: In C I can have overload resolution. You can have different resolutions.
David O: That is item 3. We'll get to that.
Matthias: If this could be solved differently it would be good. I don't want to drop this from C++. Would C entertain changing the rules here?

Rajan: This was for consistency with what was in C already before int/bool, float/double). I don't think C would change.
Hubert: We inherited a lot from C in C++. We diverged here for type safety. C may want to consider diverging from historic practice. My understanding is this is the first time that these types become a part of the standard vs a TS so there is a chance of change.
Jim: I was re-emphasizing Rajan's statement, it is not new for C. It is how C handles floating point types.
Matthias: C++ actually allowed float<->double implicitly but this is a new clean slate break.
Jim: October is the last date of changes to C, so if someone wants this, they need it soon.
Aaron: David K. May consider this a continuing proposal.
Michal Dominiak: We can also have a NB comment.
Aaron: I would prefer something before that. I would be glad to help with anyone wanting to write a paper for this.
Jim: This is a little out of scope for CFP.
gdr: The implicit conversions are a far more serious issue as it is subtle since it can be something that a programmer does not write.
David O: As I said earlier, if it compiles in both, there is no difference in behavior.
gdr: If you have a double + float64?
Michal: That is a usual arithmetic conversion, not an implicit conversion.
gdr: Promotion rules too.
Aaron: Not on the list but I want to add it.
gdr: Someone writing a program that compiles in both and has different results is my issue in whatever form (not necessarily implicit conversions).

Usual arithmetic conversions:

David O: double + _Float64 has _Float64 in C and double in C++. There should be no difference in a way users will notice in practice. For overload resolution, it is C++, not C.
gdr: If you include math.h, you have the C declarations plus more. It can behave differently.
David O: There are different overloads, but they behave the same.
gdr: Even if C doesn't have overload resolution, C++ does. This could be different.
David O: There is a difference, like printf. Varargs has an issue. _Float32 doesn't get promoted and there are no format specifiers for it. I do want to get rid of this, but it is not a show stopper. It is hard to come up with a way where it is meaningfully different.
Aaron: Re printf, N2601 (integrating part 3), it says non-prototype promotes to double. It doesn't cover the varargs case.
David O: That is out of date.
Jim: That section is being removed.
David O: The new types do not promote.
Aaron: So you can't pass them through varargs?
David O: No, you can, just not printf since no format specifier.
Jim: People asked for the rationale: C chose this for an implementation that has the IEEE types but not semantics. You'd get the semantics of the IEEE format and type in this case.
Matthias: This makes a lot of sense to me. I don't see anything in both documents that says anything about the semantics if the types are not the same. C's way makes more sense. Doing float x = x + 1.0, it is the same issue for adding a 1 to a float giving a double. This nails down the the semantics better. I want the same behavior and the C behavior is better and safer.
Jim: I agree with that, but want to point out the C spec has it in an annex. It requires float to be 32-bits and have IEEE semantics. But it is quite common to have float not follow that.
Jens: We've heard why C does it the way it does. You can have differences in calling a varargs function between the two languages. I want to know why C++ is different here.

Aaron: How do these conversions deal with extended integer rules?
gdr: They are the opposite.
Jens: The rationale for integer types doesn't apply since it is part of the precision.
Hubert: C++ rationale: Existing C++ code with a limited set of overloads can get an exact match. Otherwise you get unordered. If we switch to C, we need to add overload resolution rules based on representation.
Matthias: It should chose a double overload.
Hubert: In your case, due to implicit conversions it makes sense. But for long double, it may not.
David O: Overload resolution is not completely resolved. I would push to have it resolve to double for _Float64 + double.
Matthais: I don't see that you have IEEE semantics. I only see the representation and alignment have to be the same.
Jim: The requirement comes through having to support annex F.

Mattias: I would like to take the naming poll last because my vote depends on the others.
Jim: Is abstaining valid for CFP?
Aaron: I would vote as WG14 members.

POLL: Should the usual arithmetic conversions in C++ follow the rules in C?

Committee	SF	F	N	A	SA	Notes
WG14	0	6	0	0	0	Consensus
WG21	2	3	2	2	0	Weak consensus, Authors were 1:F, 2:N

Consensus between the committees. Weak on the WG21 side.

POLL: Should the C implicit conversion rules change to follow the rules as proposed for C++?

Committee	SF	F	N	A	SA	Notes
WG14	0	0	0	2	2	No consensus
WG21	2	5	0	2	0	Consensus, Authors were 2:F, 1:A

No consensus between the committees.

POLL: Should the _Float* C names be available (through some means) in C++ and be used as the types behind the std::float* aliases?

Committee	SF	F	N	A	SA	Notes
WG14	5	1	0	0	0	Consensus
WG21	2	2	3	0	2	Weak consensus, Authors were 1:F, 2:N
SA vote rationale: My reasons against were that similar but slightly incompatible types is not good for the same names. My recommendation to Microsoft would be against. For the French national body, I can bring the issue to them.						

Very weak consensus.

Michal: Does this imply the keywords are available in C++?
David O: No.
Michal: I think what we are trying to ask is if the C types are available.
David O: Under those names.
Michal: Then lets separate the poll into 2. Source compat vs binary compat.

Hubert: Binary compat is beyond anything we've done for interoperability between C and C++. The question as given is good for semantics. I wouldn't want the C++ committee to specify the binary compat any more than before.
Michal: Maybe strongly worded recommendation?
Hubert: We can't have a normative recommendation without talking about phases of translation and talking about C and C++ linkage.
Michal: Consider my request withdrawn.
Walter: Available how?
David O: Not specified here. Global namespace.
gdr: The the _<Capital> are already reserved for implementations. This poll is saying a program using these names are not valid.

POLL: Use the std::floatN as opposed to std::floatN_t for aliases?

Committee	SF	F	N	A	SA	Notes
WG14	2	3	0	0	0	Consensus
WG21	2	2	1	3	1	No consensus, Authors were 3:A
SA vote rationale: Similar reasoning for the previous poll. Should there be an alias at all?						

No consensus.

gdr: Similar reasoning for the previous poll. Should there be an alias at all?
Hubert: Whether it is an alias at all was not being polled.
Hubert: Can I get more rationale as to why _t is needed?
David O: Sure. I will do that.

Hubert: Jens raised the point, why is the non-_t variant superior? _t is consistent to what we did before in other places in C++ like nullptr_t.
David O: _t is consistent with other places in C++. The problems are that C defines _Float32_t is not the same type as _Float32.
Matthias: int32_t is an alias of int, while _Float32_t will NEVER be an alias of _Float32.

End at 1:03pm EST