C and C++ Compatibility Study Group Meeting Minutes (Sep 2021)

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2814
SG Meeting Date: 2021-09-10

Fri Sep 10, 2021 at 1:00pm EST

Attendees

Thomas Koeppe	WG21	chair
Mark Hoemmen	WG21	scribe
Matthias Kretz	WG21	
David Olsen	WG21	
Davis Herring	WG21	
Ville Voutilainen	WG21 WG14	
Will Wray	(14) (21)	
Rajan Bhakta	WG14	
Hans Boehm	WG21	
Jens Maurer	WG21	
Jens Gustedt	WG14	

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

Agenda

Discussing the following papers:

WG21 P2423R0 (https://wg21.link/p2423r0) C Floating Point Study Group Liaison Report
WG21 P2314R2 (https://wg21.link/p2314r2) Character sets and encodings
WG21 P2378R0 (https://wg21.link/p2378r0) Properly define blocks as part of the grammar

P2423R0 C Floating Point Study Group Liaison Report

RB: background: during previous [meeting}], not last week but before, liaison group asked me to come to this group and give overview of what C is doing -- binding to the IEEE 754 standard. How this will affect C++. Asked for 1-page summary, for what C++ and SG6 (numerics group) thinks. This [the paper being presented, P2423] is the one-page (double-sided) summary.
RB: WG14, in about 2008-9 wanted to get started on updating our reference to IEEE binding in C standard. New 2008 standard was published, so they created C floating-point study group to create binding to C. As we went through process, we developed one main TS in five parts, to show binding of IEC 60559 to ISO C. Both targets are moving targets. First we focused on C11, then C17, C18, ... and now C23. IEEE is also moving; new version came out 2020/1. [Scribe note: IEEE-754-2019 was published in 2019. ISO/IEC 60559:2020 was the published international standard.] New updates to this paper have not been accepted yet by WG14. Breaking into 2 parts: binary floating point, then decimal floating point. Part 3: interchange types: optional extensions. Part 3 was brought into WG14 as conditionally normative annex. Parts 1 and 2 are mandatory for IEEE; part 3 is optional. Ditto with Part

4a, voted into main part of (C) standard, but conditional form (macro). I won't talk about Part 4b or Part 5 today, because they were not voted into WG14 C Standard.
RB: Part 1: binary floating point, required operations. We added macros to C standard for integer type widths. Integer, because we needed the width of integer types, for fromfp operations: getting the parts of a floating-point thing, like mantissa. Query and set environment flags. New macros and functions required by IEEE 754. tgmath versions of the same thing. tgmath is "poor man's implementation of function overloading" -- C way of resolving based on types to different functions. Constant rounding modes -- new pragma which sets the static rounding mode direction. Can do at static or block scope. All operations in that block must follow that rounding mode. Back-and-forth on what functions to which this should apply: certain standard functions will be affected by that constant rounding mode. You can implement this if you only have dynamic rounding modes, which are required before anyway. Macros for signaling NaNs, and get info about floating-point values.
RB: Part 2: Decimal floating point. New types that are similar to double, long double, etc. but base 10, not base 2. Types are conditionally added (if macro defined by implementation, then types are available). New macros and functions unique to decimal: e.g., quantum. Conversion between different decimal floating-point encodings (e.g., software encoding on x86 vs. hardware on IBM).
RB: Part 3: Tower of types: unbounded spec, to allow adding extended types to the floating-point model, both for binary and decimal. e.g., binary type, 256 bits, or 512 bits -- spec lets you do that in a regularized and IEC standards-conforming way. e.g., min bits required for mantissa and exponent, operations that can be done on them. If an implementation claims it supports float256, this is a spec for that, etc. Also, spec for extended types (with "x" in them) -- not the main multiple of 32 bits, but something extra -- e.g., extra exponent bits. Min precision and min exponent range, but you can go beyond that with the "x" types. Interchange types can be arithmetic (can do e.g., a+b on it) or nonarithmetic (just for encoding). Also functions: cos, sin, etc. Quantum exponent in decimal for these types, too. Generalized in binary form for complex and imaginary types.
Ville Voutilainen (VV): Thomas asks on chat if you could describe which header gets modified by this...
RB: Headers with part 1: float.h, math.h, complex.h, limits.h (integer width macros). Same for all the other parts. Most changes are to math.h. Other changes -- encoding and decoding functions should all be in math.h or float.h -- we did not try to add any new headers for that. One semi-major change related to this -- still being discussed -- but in standard -- for freestanding implementations, we added new requirements that were not there before. Some changes ... format specifiers, conversion between strings and decimal floating-point types, added to stdlib.h header. printf headers may need to change, but that depends on the implementation.
Thomas Koeppe (TK): Would you like to take hands now or finish presentation?
RB: Take the hands now; Part 4 is small.
TK: Please finish Part 4 first, then questions.
RB: Part 4: additional recommended functions by IEEE; not required to have them. e.g., radian-based cosine functions. Additional utility functions that IEEE believes should be standardized. WG14 voted these in as part of the main library part of the standard, macro-guarded (but not in an annex).
Jens Maurer (JM): The new floating-point types, there is _Float64, which is interchange type, which means it might not necessarily have arithmetic defined, yet you say binary complex floating-point types generalized to ... was added. How do I reconcile these two statements? [Scribe note: My minutes were a bit garbled, but RB's answer clarifies.]
RB: Certain interchange types are required to be arithmetic: _Float32 and _Float64 are required to be arithmetic. Equivalent to float and double, though not compatible types. Decimal32,64,128 are required arithmetic interchange types, though the whole decimal support is optional (you need those types if you support decimal). If you say you support binary floating-point, then you must have _Float32,64, but you don't have to support _Float128, _Float16, etc. You must have _Float16 at least as a non-arithmetic type.
JM: Why the X stuff then? Float64x?

RB: That is designed to help with Intel 80-bit floating-point type. X says, minimum 64-bit floating-point type, but you could get something beyond that. e.g., _Float32x in practice, must be there because you have _Float64; those two types can be the same. X types are not required.
JM: OK. I trust there are macros that tell me which properties I get.
RB: Yes.
JM: OK.
Davis Herring (DH): A subset nearly of what JM said: When you talk about added math functions, you had to support ... interchange types ... did you mean only the interchange types that happen to be arithmetic?
RB: Restating your question: If you say you're supporting certain interchange types (vfp ...?), then you have to define them as arithmetic types ... the support for those types don't have to be arithmetic; can be data-only types.
JM: The additional functions I mentioned here simply won't be there, and there's a separate macro that tells me whether e.g., _Float128 is arithmetic or just interchange?
RB: We don't have a macro that tells if it's arithmetic or just interchange. That's implementation-defined. The implementation has to say it supports _Float256, and whether it supports that as an arithmetic type. If it does the latter, then it must support e.g, cos256.
DH: The question was, do you get cosine and friends for all the arithmetic types? However, I'm also curious whether you can use the preprocessor in some way to decide whether a type is arithmetic.
RB: I don't have the answer offhand, but my belief is yes, it can be checked at preprocessor time without looking at the documentation. I can check that and get back to the group.
DH: In C++ you can actually detect this anyway.
RB: It came to me: there is a way you can check. The macro floateval method is only available for arithmetic types.
DH: That's sufficient.
Matthias Kretz (MK): In your first answer to Jens, I heard you say that _Float32 and ... are distinct types; is that right?
RB: Yes, that's right. If your regular float type is IEEE, then it must have the same representation and alignment as _Float32, but they are not compatible types -- distinct in the type system.
MK: Yes, but why?
RB: That was a big discussion. E-mail sent out with rationale; I don't have it on hand. WG14 preferred having them be distinct types. Floating-point group had no strong preference either way.
MK: My gut feeling is that it's bloat if they are distinct types. I would like to see a good reason for them being distinct types.
RB: I would support a paper coming through that (does not do that).
MK: It becomes a little tricky with long double with 80-bit types.
RB: ... In that case, _Float128 may or may not match long double. The specification for that would be pretty hard.
MK: Macros and functions for floating-point environment flags and modes: What functions do or don't do is a problem in terms of reordering. Compiler doesn't see them as ... . is that a bug in the compiler or the spec?
RB: It depends on your implementation. If you have stdfenv access on, it would be a bug if you do these out of order. If off, though, it's ....
MK: Yes, I agree. I've seen issues with that, though maybe just implementation issues. This comes to my second point: fnaccess: I've seen compilers not supporting it, esp. if mixing -- single translation unit, or calling inline functions with different mode. Nontrivial to implement. gcc and clang both do not implement. Is it a good idea to add another one (fnround) that won't get implemented?
RB: More of a question for IEEE than us. Our mandate was to bind the standard to C.
MK: OK.
RB: This was brought up in debate.... barring special cases, just general operations like plus and minus, that implementation is straightforward.

MK: Yes.
RB: I don't see that being a problem.
MK: My specific problem is you [first] change fnround in a function, then you call a function outside the block. Which fnround applies to that block?
RB: Conclusion of discussion was that it should not apply to the function ... if you implement with dynamic rounding modes, then anything inside that function [would] follow that rounding mode. Or the opposite, if you have true static rounding mode. We carved out those special functions, with macros, you can have definitions for each possible rounding mode ... but nested calls don't work in that aspect. That is one thing we know will be a problem.
MK: I understand, but still unhappy to standardize until I know that implementations can and will do this.
TK: We don't have the power here to affect things that happen. This is already approved for C23, right? Our role here is to find out how this affects liaison matters, C++. For the details of what C is doing, we can refer that to the existing approved proposals.
MK: Yes, if I want to give feedback I'll have to do that directly.
Hans Boehm (HB): I have a basic question: does the standard say anything about accuracy? If IEEE standard, it requires correct rounding for many functions, e.g., basic trigonometric functions, which I suspect is not widely implemented.
RB: Right, we did defer [scribe note: diverge] -- IEEE requires correct rounding, but C's existing history for many functions like cos and sin not requiring correct rounding, so they are not required to be correctly rounded. However, we do reserve "cr" prefix to have correctly rounded versions of those functions. C thus offers a mechanism to have correctly rounded versions of functions.
MK: To comment on that part: IEC 559 the part on the math functions is optional -- the trig functions are in an optional section.
RB: Had thought the pi versions ... non pi versions
HB: It used to say that, but now required.
MK: "recommended functions"
HB: They are required to be correctly rounded if provided.
MK: You should say they are not provided, unless cr prefix.
RB: We supply cosine as IEC 559 1984 says. ....
TK: Does that answer your question, Matthias?
MK: Yes.
TK: Back to liaison matters: RB, there's nothing that could change C, right?
RB: Right, everything I talk about today has been voted into C standard.
TK: Any feedback you would like for C committee?
RB: I didn't have anything, but if you have concerns or issues, please feel free to bring papers to WG14, and contact me.
David Olsen (DO): Proposal for C++ started independently of this, with purpose of getting 16-bit floating point into standard. In C++, we're proposing that implementations can define additional floating-point types; if they do, the new types must follow the rules. We're also proposing some names for the IEEE types. Standard names. Like in C, would be optional. C++ likes names in the std namespace. We're debating that on one of the C++ lists now, what those names should be, but there's nothing that prevents C++ implementations from using the same names underneath.
RB: My hope would be that C++ uses the same names, even if they are in the std namespace. In that vein, it would be nice if they followed the same type of names.
DO: With very few cases are there _Capital letter names in C++. But yes, names are being debated right now. I've been tracking C changes and trying to keep C and C++ compatible, so it's possible to have code that will work the same in both languages. Possibly diff names is the biggest possible difference, but otherwise, if your code as the name that works in both languages, you can write code that will work the same in both C and C++.
TK: Assuming we will rebase C++ on C23, what should we keep in mind?

VV: Naming: I do expect it will be supported in C++ implementation[s] and maybe in spec to use the same type names as C, then probably will be teeth-grinding that will also result in having namespace names that are C++-only, in addition to Capitals, and implementations will probably have to provide both. .... It doesn't seem entirely plausible that we would _just provide the underscore capital things, but I expect we would support those as well. ... that's just my guestimate, I fully expect it may not be a walk in the park to discuss this, but that is what will eventually succeed, after the possible teeth-grinding.
DO: To be clear: the current C++ proposal does not try to bring in everything that C has done. I'm working on trying to solve a problem in the C++ community, in a compatible way.
VV: There are actual new types that have these underscore naming conventions, so we may end up supporting those directly, then have a different spelling that ... under some namespace or not.
DO: My comment goes beyond just the naming issue.
VV: OK.
DH: We have some prior art here, in the form of things like bool and noreturn, where C introduces underscore cap names, and also provides macros so that C++ things get mapped optionally, at the discretion of the implementation, onto the underscore cap names that are actually part of the language. Is there any notion of doing that for the floating-point types? Context: we talked about rebasing on to C23: surely that will happen someday, but problem is that C++ incorporates the C standard library, not actually the C core language. C++ does not have _Bool in the language at all. It might work in implementations, but it's not part of the language. It would be a bit strange to have this first use of underscore cap be either as macros going the other way (expand to C++) or expand to macros going to C. It would be nice if we followed existing precedent of _Bool and noreturn.
RB: On the C side, WG14 has been positively receptive to changing names in the future so it's not following the underbar capital format. When this first TS was created, that was the only way we did it. Recent papers have begun to change it. I don't know how C++ does it. That naming thing does not have to be a long-term issue. There is a good chance WG14 could change it.
TK: We would need to change it before C23 ships. The interop code should be using macros, not these bare names.
RB: Right. The next C meeting is the last one where you can have any new proposals.
DH: Just to confirm: there are currently NO macros for any of these float names.
RB: Correct, in C there are none.
TK: This is taking fairly long, so we only want to do one more paper. Just to go back to polls: WG14 does not want any polls. WG21, does anyone want any polls? When we rebase, we will be pulling in the headers from C as C has defined them, unless we do something.
JM: There is nothing we can do this moment about that, and I understand the agenda for the next meeting has been already filled, so unless we raise a stink, there is nothing we can do until C23 is feature complete. When C++ rebases onto C23, that will be an explicit paper that decides whether or whether not to include specific features. That's thus something we cannot do right now.
TK: I recommend writing an e-mail to the C++ reflectors to people who might be interested.
VV: You might want to run this by the [C++] Numerics Study Group (SG6) as well.
RB: Aaron mentioned that we did invite them to this meeting for this paper.
DO: Numerics group saw P1467 C++ proposal a few times. When C++ rebases on C23, the Numerics group definitely must be involved.
TK: A lot of interesting changes that will affect C++, but probably not much we can add at this point.
DH: On subject of Aaron having invited SG6, for the record, Matthias and I are regular SG6 people, so you do have representation.
TK: Are you going to take any action in C++ about this?
MK: Not sure yet. I learned something today; I have to find out what to do about it.
TK: Thank you RB.

P2314R2 Character sets and encodings

TK: Next, Jens Maurer's paper, P2314R2, Character sets and encodings.
JM: In C++, in our lexing section, we have the fundamental approqach that what C calls "extended characters" that come in from a source file, are mapped to universal-character-names. From then on, the entire standard deals in those. That does not represent reality because no compiler actulaly does that. There is actually one corner case where nobody does what the C++ Standard says. This paper switches C++ lexing to what is approximately the C99 approach: on reading the source file, we will represent the source file as unicode characters. From then on, we'll be dealing wiht unicode characters. [Here I show an] example of how this change is an observable change from C++20, even though every implementation I've tried already does what this paper proposes. Mechanics in achieving that, but that's all on mechanics side. Side track in this paper does not cause material wording changes -- places/contexts where string literals can appear.... In order to facilitate talking about the encoding used for string literals in the actual binary code, the current C++ text talks about the execution character set, which literally means a set of characters; it does not discuss encoding. All we're interested in from language specification is the sequence of numbers that comes out ... when initializing an array of characters. That gets cleaned up by introducing the term "literal encoding." There are character sets in the run time that might depend on the locale, but we're not touching this area of the spec, which is inherited from C anyway. I think that's the gist of the paper.
TK: Thank you. What questions do you have for WG14?
JM: Mostly none. Aaron told me to show up and present the paper.
TK: WG14 members, do you have any reactions to this, is there anything that may cause incompatibility, or will it fit in unobservably with C?
JM: The intent is that the three models presented in the C99 rationale, and any difference ... would be undefined behavior or unspecified. ... should not affect C compatibility in any way.
Jens Gustedt (JG): I don't see incompatibilities I think, but I'm not too much a full expert of this. What I see is things I'd like to have in C, in particular the table of characters.
JM: Yes, everything is expressed in unicode characters these days.
JG: I'd really like to have this in C -- at the moment, it's fuzzy what you mean by character -- this would give more precision. I don't see, on the other hand, how we could add that to C23. It's coming too late for that. Otherwise, I like this.
TK: I think an agreement in direction is good, even if the wording isn't there.
JG: I think the intent in C is to have these things aligned, and I don't see any point here that would particularly worry me.
TK: Open to any further requests for discussion. I get the impression there are no polls we want to take here. What's the state in C++? Is this adopted?
JM: This is, I think, approved by EWG electronic polling and is in Core's inbox. The next plenary is in October; I'm not certain Core will have enough bandwidth to process this [by then]. Very likely to make it into C++23.
TK: If any major objections or problems, now would be a great time to raise them.
JM: It seems a bit of a big ask to present this in five minutes of overview.
TK: Right, we're mainly raising awareness [for later response if needed].

P2378R0 Properly define blocks as part of the grammar

TK: Jens Gustedt, you have a paper.
JG: We discussed this paper last week at WG14 plenary. It has been basically approved, but with some changes.
JG: P2378r1. In the spec of the C grammar, there is no occurrence of the term "block." "Block" is introduced with a collection of phrases somewhere. This paper only changes the specification. However, if you look at it closely, you'll see that there are possible problems in the syntax already. We distinguish

two types of blocks in the syntax: primary-block and secondary-block. The latter rule only occurs in constructions like the depending statement in an if, if-else, switch, while, do, or for. It's really only a rewriting of what was there before, so that it falls out directly from the syntax. The only place where it poses problems, is the block that corresponds to function definition. If we also have lambdas, [it] would also be the same for lambdas, where not clear if parameter list definition of the function is part of the block which is defined by the function, or not. The change that we have decided here now that will probably go into C23, is to say that the parameter type list, the attribute that can live between the parameter type list and the function body, all together form one single block. Advantage is that it's clear where the visibility of a parameter identifier starts and ends .... This is basically what implementations do, but was not really clear from the text as it had been given.
JG: This causes one extra "interesting" problem: compound literals, which in C are these temporary objects which have a bit different definition in C than in C++. In short, I have split that off to a second paper: a compound literal which occurs in an expression is a temporary object, but its lifetime is the surrounding scope. Function literals that appear in expressions ... this was not clear. Switching to second paper: PyyyyR0 [scribe note: JG confirmed that the second paper did not have a paper number yet]. "Disambiguate the storage class of some compound literals." vla parameter "para," depends on a function call, inside of which there's a compound literal. Where does this compound literal live? Is it automatic or static? Compilers disagree (clang and gcc). WE proposed and WG14 accepted that these objects should live inside the function and not be static objects. The second part should probably not impact C++, for two reasons. First, the compound literals that C++ added or is going to add, they still live only as temporaries inside the expression where they occur. Second, it only occurs in C because vla, which C++ does not have. Thus, probably this change is not so important for C++. There remains the change of the other paper. (Second paper does not have a paper number yet.) Discussed in WG14 just a week ago, in principle we have agreement that it should go into C just as presented here. I have no clue how this syntax is defined in C++, whether this matters or not. The intent here is to do no change, which means that we still defer on some things ... from C++. If you perhaps remember this last case here, of a for statement where we have a declaration, we have different scope rules for the iteration variables. This still will be the case as before. It was proposed to WG14 to change those scope rules; WG14 decided not to change that, so the rules are different in C and C++.
TK: What would you like from C++?
JG: I would just like to see if anybody sees difficulties, in particular the thing which is more precise than it was before, is this defining the scope of a parameter to extend to the gap between the parameter list and the function body. This was not clear in the syntax before in C. This is a new specification in a sense, which was not there before. This becomes important if somebody adds attributes which use parameter names.
JM: C++ always had stuff after the closing parenthesis of the function parameter list, e.g., noexcept spec that could refer to parameters. We recently approved a paper from Davis Herring that refined the ambiguity in C++ lookup rules in that area. In C++, we have covered the name scope oddities that sometimes exist, mostly with special prose rules if necessary, and not with refactoring the grammar. I don't see an urgent need for C++ to do something like this.
JG: The question would be if this would introduce incompatibility.
JM: Name lookup works hierarchically; the innermost one first [so global names won't collide]. That matches what C++ does, as far as I know.
DH: Overlapping with what JM said: In C++11, where the trailing return type was introduced, ... there's no incompatibility there, which is nice. ... [missed some context] it would be difficult to construct an example where the code has different meanings in C and C++. I mention that because it's a potential incompatibility, but I don't expect you would find it unless you went looking. .... While my paper did clean up the terminology in a way that makes it look like your use of secondary block (albeit in prose, not in grammar), we ... had rules ... but it was done by ... the substatements of an if simply get a scope as if they had braces -- little subtleties but I expect they won't be interesting. I mention this because we are

using a different specification technology, but we do almost the same thing, and in places where they are not the same, I expect that there will not be any compatibility.
TK: I get the feeling that there's nothing here for us to settle. We could perhaps inform C++ Core of this paper to have another look, that code that is both valid C and C++ would have the same behavior, but it seems we're not overly concerned here. ... Would anyone like a poll or raise any more concerns or ask more questions?
JM: We're out of time already.
TK: Is this accepted for C23 already?
JG: This is accepted in principle for C23, with some wording things.
TK: So we will inform WG21 of these, but it sounds like you have the go-ahead from this group.
DH: On function params vs. block scope: one reason we have to contend with that in C++, is because we have function try blocks. Not as simple as end closing brace -- which closing brace? There is a reason we're doing [specifying] it differently.

End at 3:00pm EST