C and C++ Compatibility Study Group Meeting Minutes (Feb 2022)

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2933
SG Meeting Date: 2022-02-11

Fri Feb 11, 2022 at 1:00pm EST

Attendees

Aaron Ballman	WG21/WG14	chair
Philipp K. Krause	WG14	
Robert Seacord	WG14	
Tom Honermann	WG21	
Hans Boehm	WG21	
JeanHeyd Meneide	WG21/WG14	scribe
Corentin Jabot	WG21/WG14	
Joshua Cranmer	(21)	
Rajan Bhakta	WG14	
Gaby Dos Reis	WG21	
Zhihao Yuan	WG21	
Timur Doumler	WG21	
Jens Mauer	WG21	
Alex Gilding	WG14	
Jens Gustedt	WG14	
Hubert Tong	WG21	

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

Agenda

Discussing the following papers:

WG21 P2478R0 (https://wg21.link/P2478R0) _Thread_local for better C++ interoperability with C
WG21 P1774R5 (https://wg21.link/P1774R5) Portable assumptions
WG14 N2919 (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2919.pdf) Relax requirements for variadic parameter lists

P2478R0 _Thread_local for better C++ interoperability with C

Rajan: As a recap, there was feedback from WG14 that a different paper to make `thread_local` be an actual keyword that is being voted on next week. This impacts this proposal since that means C and C++ will have the same keyword but with potentially different semantics.

Hubert: Thank you Rajan for noting that the vote is still coming up. The hope for this meeting is to get an idea from the C++ Committee side if they appreciate having a different syntax for `_Thread_local` versus `thread_local` and the small but important differences between the two. C++ should have the exact `_Thread_local` keyword that retains C semantics of not requiring dynamic initialization or destructor tracking. Do I need to give a summary of the paper?

Aaron: I think a summary would be good here.

Hubert: Okay, so the general summary is that C++ has dynamic initialization and C does not. In practice, only the dynamic initalization for _Thread_local case actually makes any serious differences between implementations, where deferred initialization needs to be used for C++, while C has no such concept. The issue is that C references a `thread_local` variable, it will read uninitialized data. At least in Clang, if you use exactly the `_Thread_local` spelling, it will error if the type and value requires dynamic deferred initialization. This proposal is meant to standardize existing practice for the difference between the 2, rather than attempting to paper it over with just the `thread_local` macro defined in <thread> for C++.

Aaron: Questions? Yes, Jens Gustedt.

Jens: Not really a question, but... so I have a paper that will be writing `_Thread_local` as `thread_local`. That means that the spelling difference may go away between the two, so this might continue. I don't think this problem is worth solving since C should not be burdened with C++'s initialization issues, plus we already have spelling for lowercase `thread_local`.

Hubert: To answer why we should care and this is a problem, we have problems in shared C and C++ environments. C code will be unable to tell the difference between thread_local and _Thread_local, and so all C would need to pay for this difference since it cannot tell properly what the source of the variable is.

Aaron: Yes. Next person- Robert?

Robert Seacord: `constinit` is a keyword that's in C++20 that solves this problem. Why don't we pursue that route to solve this problem in C?

Hubert: We assumed that the C Committee would not be interested in the `constinit`. Furthermore, we have a problem with incomplete types and other types, where the initialization or destruction characteristics are not known until later. In C++ this is okay because it will just dynamically initialize, but in C this means the value might not have or need a deferred initialization. we help guarantee this with [[clang::nodestroy]], to ensure the behavior is consistent. Need more than constinit, and proposing to WG14 is a lot to do. (EDITOR'S NOTE: here, Hubert describes referencing an object thaat is made to be thread local that has special destruction semantics. The initialization may trigger a constructor call, and in C++ that means a destructor call must also be emitted. If this is behind an opaque pointer, C can get this wrong.)

Robert: Okay, that makes sense to me. I'll take my hand down.

Aaron: Ah, that makes sense, okay. Jens?

Jens: I don't see this as a problem for us in C. We would need to restrict the design space of C for a problem that C++ has had for a very long time and has not solved (static initialization fiasco) and we are lost in

Aaron: A clarification, this paper is about presenting to the C++ Committee adding to `_Thread_local` as a special keyword to C++, so it takes on C's semantics. The semantics would change using `thread_local`.

Hubert: Yes. And we also need to coordinate with WG14 because they need to change the behavior to

Aaron: Okay. Corentin?

Corentin: As a C++ developer, I expect my C code and C headers to work normally.

Hubert: This is a safety issue, not just a performance issue. And maybe I failed to properly describe that, but I did mention that it has uninitialized memory--

Corentin: Okay, well, can you explain it to me a bit more?

Hubert: When you create a `thread_local` in C++, and it has deferred (dynamic) initialization because it has a destructor or a constructor, and so if you access that from C it does not know this. This means it WILL have you access uninitialized memory.

Corentin: Okay, I understand it bettter. I'm not sure I agree with the current direction but I understand, thank you.

Robert: So to ask a clarifying question: next week, we're going to be voting on making `thread_local` a real keyword. Are we going to deprecate or remove `_Thread_local`? A question more for Jens since it's his paper at the WG14 next week.

Jens: The problem we have here is that there is already deployed `thread_local` as a macro and there is C macro is writing it already. This is a problem because that means the code changes meaning between the two, since we're still using `thread_local`, since C11. We should try to make a

Rajan: Hubert, is it okay if I answer it?

Hubert: Go ahead.

Rajan: When you preprocess the source, you get the `_Thread_local` proper spelling because it's just a macro. It doesn't necessarily break C code because it's always going to be preprocessed into `_Thread_local`. The problem is if we go through with the proposal next week to change the syntax, and then there's an actual problem. Right now preprocessed source gets the right semantics with `_Thread_local` in C, and only becomes a problem for C++ code compiled with C++ headers referenced from C.

Jens: The code is still broken between C and C++, because C++ will still choose the C++ semantics, even if C chooses right semantics for its mode.

Tom Honermann (in chat): Would it to be terrible to have the C semantics indicated via extern "C"? extern "C" thread_local tlv = …;

Alex Gilding (in chat): we found that for at least some keywords, some existing impls provide the lowercase spellings as first class keywords, not macros

Robert: So we're in an uncertainty case because both Committees are using both of the `thread_local` keyword versus macro do different things based on how the header is used / built for shared code.

Rajan (in chat, to Alex Gilding): If that is the case for C, that is already non-conforming since it takes away user namespace.

Corentin: C++ users will never know when to use the `_Thread_local` spelling versus the `thread_local` spelling, and so it will blow up in their C user's face. Plus, if we are confused, our users will be confused. I like Tom's suggestion in the chat to use `extern "C"` of some kind to tell the compiler to do something.

Aaron: We've got 5 minutes left.

Hubert: To Corentin's point, we're really worried about the headers we have. Users in C++ don't have to use thread_local in shared C/C++ code, and because stdthread.h doesn't exist in C++, so it becomes harder for them to use `thread_local`.

Rajan: The goal is for fail fast with compiler errors.

Hubert: Yes, that's good. We came in expecting 2 polls. The first is whether we believe C++ should evaluate a C compatible form that matches the `_Thread_local` spelling. The second poll

POLL: Does SG22 encourage WG21 to evaluate a C-compatible form of _Thread_local (specifically) with the appropriate C semantics along the lines of P2478R0?

Committee	For	Against	Abstain	Notes
WG14	4	1	3	Weak consensus (abstentions)
WG21	2	4	4	Consensus against

Overall: The SG22 Compatibility Group does not have OVERALL consensus.

No vote rationale: Against, because I wanted any kind of spelling that was not either `_Thread_local`/`thread_local`. We see the problem, but I want a different spelling.

No vote rationale: Similar to above, some other kind of spelling.

POLL: Does SG22 recommend WG14 retain the requirement for <threads.h> to be included to use the "thread_local" (specifically)?

Committee	For	Against	Abstain	Notes
WG14	2	3	3	No consensus
WG21	4	1	5	Weak consensus (abstentions)

Overall: The SG22 Compatibility Group does not have OVERALL consensus.

P1774R5 Portable assumptions

Timur: Okay, I have a few slides...

Timur Presents. Effectively, this is about __assume(expr), __builtin_assume(expr), and similar and the existing practice. Timur then shows off the solution in C++. He has considered several different kinds of

syntax: keyword, macro, magic library function, novel syntax, and attribute. Keyword is existing practice, breaks identifiers. Macro is strictly worse than keyword thanks to how arguments are parsed for macro functions. std::assume is good, but the expression passed to std::assume MUST NOT be evaluated. Nothing in the language exists like this. Novel syntax, such as [[assume: expr]], has downsides with being non-backwards-compatible, non-C-compatible, and adding new syntax to the language for not too much big benefit. Attribute syntax such as [[assume(expr)]] is the best since it already has all the proper semantics. It's C compatible because attribute syntax already workds, and matches [[likely]]/[[unlikely]]/[[fallthrough]] and similar. Attributes are also "ignorable": the program is still valid after the assumption is removed, so it's natural to use attribute spelling.

Timur opens for questions.

Jens G. (Question/Comment): Attributes might not work because they may not be used in all the places expressions are used. So why don't we use a similar syntax for `unreachable()`, or vice-versa.

Timur: (Explanation of how `unreachable()` can be used with expressions, specifically to mark a branch that should never be taken and have that information flow backwards, while assume provides information that can be used in a forwards-compatible manner.)

Aaron: You want `unreachable()` to have expression semantics. There is not an expression-level need for assume, since it's just informing the compiler. You can place it ahead of a temporary, and its' fine.

JeanHeyd: To get the assumption to apply for a block, can you put the attribute at the top of a block in anotherwise empty statement?

Timur: Yes, there's some slides for that.

Hubert: It holds at the point it appears, and onwards (e.g., in the whole block). If there was a modification to something in the in the block, then the assumption doesn't hold anymore.

Philipp: This is good but we try not to have multiple different ways to do something. We might not want unreachable if it comes.

Aaron: Also, WG14 doesn't really have magic library functions, so that would be somewhat novel (for unreachable).

Timur continues to present. He talks about the semantics of `[[assume(expr)]]`, particularly, and how. He then displays the the wording for the paper. Most of it fits on one slide, there's a few extra added sentences elsewhere to deal with [[attribute]] syntax and duplication rules.

Robert Seacord: So can this be used to create a "restrict" with this?

Timur: No, there's no compiler that can solve that problem.

Jens G.: I am not at all against this paper here, but I just wanted to note that the assume attribute cannot be used in the same e.g. for a ternary expression. Secondly, you cannot ignore the unreachable() syntactically.

Timur: Okay, that's fair. But you can get very close semantics with, for example [[assume(false)]] and unreachable() have the same stuff.

Hubert: One note, you can create unreachable by using pure/const and inline it and use Clang to write,

```
void unreachable(void) { [[assume(false)]]; }
```

and it will have similar expressions.

Hans: So if I used a variable that is accessed from multiple threads in `[[assume(...)]]`, will that provoke a data race?

Timur: No, so the assume means that it cannot provoke undefined behavior from evaluating the expression. So it won't throw an exception, it won't create a data race if it accesses a variable, and it won't do other undefined behavior. Things cannot rely on the evaluation of `assume(...)` since it is potentially ignorable and unevaluated.

Hans: Okay, that's what I was hoping.

Hubert: You can cast a pointer to `uintptr_t` and then use assume on that, and it might help you achieve alignment checks and get the compiler to do some amount of pointer alignment, at least on Clang.

Timur: Yes, but C++ has a fix for that already: std::assume_aligned(). This prevents problems with implementation-defined-ness issues for `uintptr_t` conversions (or the type simply not existing because it is conditionally defined).

Aaron: Okay.

Timur: Do I need to write a C paper?

Aaron: We don't need force authors from either Committee to give their paper to the other.

Timur: Well, I volunteer to do that, I have every intent to C.

Aaron: Okay! And I will go on the record to help with that!

Timur: Great!

(Some discussion of process for WG14/WG21.)

Jens G.: How come unreachable() fell through when [[assume(expr)]] was on the way? Especially if unreachable() can be expressed in terms of [[assume(false)]] in an empty function? Is that a failure in WG21 process?

Timur: WG21 made it clear that they wanted both, when reviewing unreachable() and when reviewing [[assume(expr)]]. It seems like C has a different preference but for WG21 they wanted both, after having the discussion for both.

POLL: Does SG22 encourage proposing the functionality in P1774R5 to WG14?

Committee	For	Against	Abstain	Notes
WG14	6	1	1	Consensus
WG21	8	1	2	Consensus

Overall: Consensus to move forward

WG21 No vote rationale: There is a usability problem because assume leads to more brittle code in usage experience (with Microsoft Visual C++). The MSDN doc has a warning that says DO NOT use `__assume(...)`, and should only be spelled assume(0) which gets turned to `builtin_unreachable` / `__unreachable` or equivalent. (Timur offers to each out to this person to learn more, albeit it is clear from previous discussion in Evolution Working Group that they acknowledged the existence of this and decided to go ahead anyways to the next stages for WG21 for this proposal. SCRIBE'S NOTE: Compilers can still just straight up ignore the attribute and the code shall still be conforming, so implementers can still get what they want if they don't like `[[assume(expr)]]`.)

WG14 N2919 Relax requirements for variadic parameter lists

JeanHeyd: there's a new draft of the paper for the C++ mailing (https://isocpp.org/files/papers/D2537R0.html)

Alex: seen N2919 in WG14, but not officially (we will see that on Monday)

JeanHeyd: C23 is removing K&R function declarations, functions without prototypes. e.g., void foo(); accepts any number of arguments in C

JeanHeyd: but there's still an issue with inter-language function calls, where you create a K&R function without a prototype in C and then call it through assembly/other languages, so long as the programmer was careful with ABI.

JeanHeyd: the proposal is to not require va_start specify the last parameter before the ... (and allow function declaration of void f(...) with no explicit parameters. This part is being discussed on Monday in WG14.

JeanHeyd: We tested implementations and found that no implementation seems to need the first argument. You can't always get to the arguments via the first parameter anyway; some ABIs still pass arguments in registers, including the variadic ones.

JeanHeyd: proposal is to allow an elipsis as the only parameter in a function declaration, and to drop the required second argument to va_start.

JeanHeyd: a pure library implementation is possible for va_start, the paper has a link to the implementation

JeanHeyd: wording is funny for va_start because C doesn't have __VA_OPT__ yet (the C++ wording is a bit easier).

Hubert: C++ has ABI aspects as well, the functions without prototypes can have a different calling conventions than a function with .... There are implementations that use stack-only for the ellipsis arguments. I don't think you can get rid of the first parameter.

Alex: __builtin_va_start seems like something that is trivial for implementations to implement, and the compiler then has the information it needs to handle this.

Hubert: my concern is more with the C and C++ ABI compatibility aspects of it, where C++ already allows ellipsis-only function declarations.

JeanHeyd: so the issue is void f(...); in C++ and void f() in C? I am unaware of ABI difference there if it has both C and C++ implementations.

Hubert: there are platforms that change behavior between ... and K&R C.

JeanHeyd: so this won't be a drop-in replacement for K&R, you'd need an additional syntax (such as an attribute).

Hubert: correct, an attribute would have been a reasonable approach

JeanHeyd: I'm not sure how easy it would be to declare an ABI attribute; it would be difficult to standardize (outside the scope of the standard), but maybe we could use hand-wavy wording, but it'd be hard to get past WG14. Worth bringing up in WG14 on Monday.

Jens G: declare functions with the new syntax as well as define functions with the new syntax?

JeanHeyd: yes, both declarations and definitions

Jens G: A little surprised that we need to have definitions for these kind of things

Alex: it's more of a symmetry thing; users aren't encouraged to use this, but it's consistent to allow definitions.

Jens G: that does add complications because of the definitions. Declarations are easy sell, definitions less so.

Jens M: several layers of complexity. I agree with what Hubert said; this won't help for K&R compatibility. C++ definitely has taken the design space for what void f(...) means, so the concerns about ABI compatibility with K&R are likely valid. A separate question is whether the ... function definition should be able to access those variadic parameters. Maybe the feature is important enough to be worth the effort for compilers to change how they handle stdarg.h for adding new builtins.

Hubert: the ability to define in C without the first parameter may be useful for "struct hack" common initial sequence

Jens M: I think the paper may need to add more motivation along those lines?

Zhihao: there's no motivating example of why va_start needs to change, so is making that change required?

JeanHeyd: there is no parameter to pass in, so in order for va_start to be usable.

Hubert: what is the motivating case for K&R? Is there any case where you're calling in to C and it's K&R defined with the promoted versions of the types?

Alex: it is separable between declaration and definition

JeanHeyd: I ran into pain with the requirement that there be a first parameter in real world code

POLL: Does SG22 think N2919/D2537R0 is worth pursuing as-is in WG21?

Committee	For	Against	Abstain	Notes
WG14	3	0	5	Weak consensus (abstentions)

WG21	4	0	3	Consensus

Overall: consensus

Abstentions were around the ABI concerns and the split between declaration and definition

Wrapup

Aaron: sorry for running a bit long, thank you to everyone who was able to stay a bit late to wrap up.

Aaron: I'll have the next meeting schedule out after the WG14 meetings end, but should be back to our usual time in March.

End at 3:13pm EST