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

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2815
SG Meeting Date: 2021-10-01

Fri Oct 01, 2021 at 1:00pm EST

Attendees

Aaron Ballman	WG21/WG14	chair
Thomas Köppe	WG21	scribe
Ville Voutilainen	WG21/WG14	
Miguel Ojeda	WG21/WG14	
Will Wray	(14)/(21)	
Peter Brett	WG21	
Jens Gustedt	WG14/(21)	
Jens Maurer	WG21	
Corentin Jabot	WG21	
Gabriel Dos Reis	WG21	

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

Agenda

Discussing the following papers:

WG21 P2362R2 (https://wg21.link/p2362r2) Remove non-encodable wide character literals and multicharacter wide character literals
WG21 P0627R5 (https://wg21.link/p0627r5) Unreachable control flow **and** WG21 P2390R1 (https://wg21.link/p2390r1) Add annotations for unreachable control flow
WG21 P2361R2 (https://wg21.link/p2361r2) Unevaluated string literals

P2362R2 Remove non-encodable wide character literals and multicharacter wide character literals

PB presents P2362R2 ("Remove non-encodable wide character literals and multicharacter literals"). These literals exist in C++ and in C, so this concerns the liaison group.

PB: If wchar_t is 32 bits, there aren't any non-encodable character literals, and there is no change. Problems exist when wchar_t has 16 bits. Some implementations discard data, some have diagnostics (which may be on or off by default), some reject.

TK: Is this only about character literals, not string literals?

PB: Correct.

PB: Regarding multi-character literals: for narrow characters, the type is `int`, so one can meaningfully pack multiple characters. But a wide multicharacter literal still has type `wchar_t`, so there is no way anything could be packed. This isn't meaningful, and there's implementation divergence. I propose to just make wide multi-character literals ill-formed.

VV: For C++, EWG has approved this change already.

PB: If SG22 approves, I think.

VV: Yes. It's making progress.

JG: I've always been mystified by these multi-character literals. I think some people in WG14 have very strong opinions on this, which is why everything is allowed and "implementation-defined". We might really want to poll WG14 on this.

PB: The only parts that this proposal addresses are the ones that are currently completely unusable [ed. e.g. narrow literals are not touched by this proposal].

AB: Are Unicode character literals affected at all?

PB/CJ: no (but will check), since those are guaranteed to be representable.

AB: Speaking as chair, I'm not too concerned here since this doesn't affect interfaces. I don't see a reason personally not to progress, but if JG wants to step carefully and wants to discuss this in WG14, I understand. [Discussion on test suites that may be affected.] It's possible someone in WG14 raises a surprising effect, but I can't imagine how this would affect C++.

JM: From an implementation viewpoint, if you consider C++ "an implementation", then it would be a possible, conforming implementation of C++ to diagnose, even if C considers it implementation-defined. C++ has in the past reduced the latitude C provides. Portability under the C rules already ensures that portable programs don't depend on this implementation-defined latitude.

CJ: Agreed with JM. We already implemented this change in Clang. We'd also suggest that WG14 consider a similar change.

JG: There are still two weeks, a small paper could still make it. Only a very small wording change would be needed, and the rationale would be the same.

AB (and others) would be nice, though there's no requirement for C to also do this right now so that C++ can progress.

AB: CJ has a general cleanup paper coming up via SG16 ("Unicode"). Is there a general appetite for SG16 to help get C++ "in sync" with C?

PB: Our current focus is C++, but there is definitely a possibility (not now, fiddly, multi-year) to bring lexing, literals, string processing in a way where C and C++ are specified almost identically.

AB: I was mainly interested in making sure that nobody in WG14 needs to rediscover what we've learned (WG14 has fewer resources than WG21), and we should communicate everything well.

CJ: If people want a high level summary of c++ changes - https://wg21.link/p2178r1.

POLL: Does SG22 agree that P2362 does not introduce any incompatibility between C and C++?

Committee	SF	F	N	A	SA	Notes
WG14	5	1	0	0	0	Consensus
WG21	6	2	0	0	0	Consensus, author position was SF

Consensus, congratulations.

P0627R5 Unreachable control flow
P2390R1 Add annotations for unreachable control flow

JM and JG present P0627R5. Compilers have built-ins to annotate unreachable control flow, which can help generate better code. We propose to expose this as a standard library function. For C++, we propose `std::unreachable`. JG has a proposal for C which makes this a core language feature that happens to look like a function call. In C it's explicitly specified so that you can't take its address, whereas in C++ we rely on blanket wording that disallows taking the address of standard library function unless explicitly allowed. JG presents the C paper. For C, `unreachable` would be macro in `<assert.h>`, which expands to some new keyword. This would allow C++ to specify this facility in `<cassert>`. There is also an alternative proposal as an actual, normal function, plus wording to achive the desired behaviour. (Since C does not have the same machinery as C++ does for requiring preconditions.) I'm not sure which version WG14 will prefer.

AB: There's some precedent for making functions unaddressable in C [ed. by default, functions _are_ addressable in C], namely in `setjmp`, for which it's unspecified whether it's an identifier or a macro.

JG: This is a bit different from `setjmp`.

AB: A personal note: I'd be worried if C got this as an expression and C++ got this as a function. It's a gut feeling, though, I haven't got a concrete example of code that would be impacted by the distinction.

GdR: For half a century `<assert.h>` has only defined macros. In C++ this affects the new module system. In C++, the `std::unreachable` function is in `<utility>`. What can we do about this friction? JG: I see the point. If it's a new keyword that's an expression, then we'd put the `unreachable` macro in `<assert.h>`. If it's a function, then we could still have `<assert.h>` define a macro.

GdR: I see it as problematic if `unreachable` is a macro (in `<assert.h>`). It it were a function, we could put it in `<stddef.h>`/`<cstddef>`. A macro would prevent me having my own function of that name.

AB: This may be a misunderstanding; the C allows every function to be a macro.

GdR: But C++ disallows that for the "C headers". If we want to reduce friction, we need to find a common header and not use macros.

JG: The C++ assert.h header could use, say, a using-declaration, whereas C's header could define a macro.

AB: C++ already has to deal with this for every functional interface from C.

GdR: But this is about the user not the C++ standard library implementor. Does including `<assert.h>` in C++ define the macro or not?

AB: That's a good question.

MO: What about noreturn?

JM: I'm not a fan of it.

MO: Me neither.

JG: ???

MO: We should probably agree between C and C++.

JM: It doesn't make a difference, it's UB to reach that question.

AB: Good point. It'd be hard to explain to users why one language has it and one does not, but it does indeed not matter.

TK: Regarding interop: headers aren't so important, since interop code can use `#ifdef __cplusplus` to select the necessary headers. What matters is that the actual code is spelled the same (namely `unreachable`). Regarding whether headers define macros, we can specify C++ any way we like, and we can make it _not_ define macros. We already do this for other headers, e.g. `stdbool.h` defines a macro `bool` in C, but not in C++.

JM: If C defines this as a function, there will indeed be a compatibility issue when we rebase C++ on C. Also, note that C++ currently has no way to get `unreachable` in the global namespace.

GdR: It'd be easier if `unreachable` was a function, in a header like `stddef.h`, so that C++ gets it that way (including the `std::`-qualified version).

AB: assert.h is not freestanding in C.

JM: Because it aborts. stdlib.h is a possibility.

JM: I can update the WG21 paper to move this into one of our C headers. We can for the time being add this to only the C++ header (`<cfoo>`), which we've already done for `<cmath>`, and when we rebase on C23 eventually, we can just drop that. But timing will be tough, since both WG21 and WG14 need to be feature complete very soon. And to persuade WG21 to modify a C header, we need to be somewhat sure that WG14 will use the same header, or we'd have the worst of both worlds.

JM: repeat technical objection that assert.h is not freestanding, but this feature clearly should be freestanding. Maybe stddef.h.

AB: If JG goes with a core-language design, not a function design, then what? JM: Would still like it to be a function-like macro.

TK: If C went with the core-language approach, then C++ could make assert.h/cassert not define the macro, but wouldn't be able to provide the global namespace name from `<assert.h>` without including `<utility>`.

JM: Regardless of whether C will use the functional or core-language design here, this will not go in `<utility>` anymore, but in one of the C headers.

VV: C++'s `<assert.h>` wouldn't need to include `<utility>`, but could just "also declare the function" (internally using a common, internal header). There's no problem using `<utility>`.

[More discussion] assert.h is not a good idea, and some C header would work nicely for C/C++ interop.

JG: Points taken from the discussion: use stddef.h, make it freestanding and mention this explicitly, make the macro a function-like macro.

AB: Question for JM and JG, do you want any straw polls?

JG: I have the direction I need.

JM: any objections to the above actions?

None, general agreement.

P2361R2 Unevaluated string literals

CJ: Concerns string literals that are never part of the program, e.g. static assertions. Should such strings allow encoding prefixes (wide, Unicode)? We propose that encoding prefixes should not be allowed, nor should numeric escape sequences.

AB: I have some feedback: Banning numeric escape sequences may be overreaching, users may want to put explicit bytes into, say, their `asm` statements.

JM: The asm text is asm source code.

TK: I agree that it might be overreaching, since users may well have a particular setup that works for them in which particular numeric values make sense. It's not portable, sure, but it might be working. Should we take that away?

CJ: Numeric escape sequences usually don't make sense in those places. You could use universal character names if you like.

TK: Yes, I agree that one could do things better, but I'm just wondering if we're taking (non-portable) code away from people for whom it might be working well.

AB: CJ, any questions/polls? We haven't heard from the C people yet.

JG: Haven't given it much thought yet.

AB: It doesn't look like it would introduce incompatibilities. One could contrive cases such as vendor-defined attributes, but it's a stretch.

AB: I suppose the only real question is whether this should proceed to EWG for C++.

[No general objections. VV concerned about numeric escape sequences, but to be worked out by individuals.] CJ, this is fine to go to EWG.

End at 2:48pm EST