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

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2752
SG Meeting Date: 2021-06-04

Fri Jun 4, 2021 at 1:00pm EST

Attendees

Aaron Ballman	WG14 WG21	chair
Jens Maurer	WG21	
Clive Pygott	WG14 (21)	
Jens Gustedt	WG14 (21)	
Miguel Ojeda	WG14 WG21	
Will Wray	(14) (21)	
Martin Uecker	WG14	
Inbal Levi	WG21	
Robert Seacord	WG14	scribe
Ben Craig	WG21	
Steve Downey	WG21	
Ville Voutilainen	WG21	
Tom Honnermann	WG21	

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

Agenda

Discussing the following papers:

WG21 P2312R1 (https://wg21.link/p2312r1) Introduce the nullptr constant
WG21 P2338R0 (https://wg21.link/p2338r0) Freestanding Library: Character primitives and the C library
WG21 P1949R7 (https://wg21.link/p1949r7) C++ Identifier Syntax using Unicode Standard Annex 31

P2312R1 Introduce the nullptr constant

Jens Gustedt: Talk about nullptr constant old fellow known for C++. Don't have something corresponding in C++. Hopefully this will make things more smooth between the two languages but should also improve C. There are also reasons this will improve the C language.

There are three reasons. Some of the reasons that make NULL a bad choice between it can actually be different kinds of integer constant expressions that can have type int, long, or void *. This means that the type of this beast is not fixed and you can have really bad surprises in C. Give a second condition where you have a conditional expression and the type of these conditional expressions can be different depending on the type of the NULL constant.

The third one is especially important because NULL is a bad choice if you want to pass into a varargs function because you don't know what NULL is so it could be 4 bytes on the stack or 8 bytes on the stack but you don't know which so you have undefined behavior. For this reason it is a really bad choice.

There is already a history in C. Thought he had to convince WG14 committee members to have a macro that converted to void * but WG14 is very fond of the feature itself and didn't want it as a macro but to have all the features that it has in C++. In my implementation here there might be some differences but this is from integrating into a different language with different rules.

nullptr adds a keyword as an integer that has almost no type. Has a point nullptr_t which is the same type as C++ but should only be used in context where it immediately converts to a real pointer type either object pointer or function pointer. Also some other contexts listed in tables such as parenthesis around it can be omitted or the target value of a generic choice or in a controlling expression which has the obvious value that it evaluates to zero as we still have ints for a controlling expression. Negation gives one, nothing fancy going on. Emphasize that grammatical lexical replacement by the calendar and nothing more should happen.

nullptr is an incomplete type

Will Wray: nullptr -> false, !nullptr -> 1 (why not !nullptr -> true?)

Aaron ballman: I think that might just be an oversight? I'd imagine it results in true, not literally 1

Jens: This is basically the nullptr keyword used in any context that it converts to nullptr. The nullptr_t type is only meant to play a minor role if you really need that to do generic programming. Because this type should basically not be used it is perfectly fine that it is not a primary type but just a typedef that resides in stddef.h.

As already mentioned after we define this type we add it to type generic interfaces and function calls and function something else.

Propose that this old fashioned macro should be deprecated although this probably won't pass at WG14.

Lost Jens.

Found Jens.

Specification on that constant it's relatively short and relatively easy.

Restarting conversation…..

The changes are around the text of the C Standard need to add to text of key words and a bunch of C internal stuff. We have the real text on one page which is the nullptr constant definition which has in first paragraph for syntactical replacement if it occurs in some special context how it is adjusted and when it appears in a context that is not a constant expression.

I thought I had an example of Generics somewhere but not much importance.

It's intended to be the same as in C++ at least in the effects of the keyword but it's not so critical as it doesn't appear in interfaces too much it's mostly a question of people getting used to using it.

Mostly looking for opportunities where this creates incompatibilities with C++.

Jens Mauer: Under #2 third bullet is quoting the C++ rules that no object can create objects of this type. Objects can be created this is not fascinating but can exist. So parameters are not the odd one out.

Jens G: Wanted to avoid creating yet another object type which cannot have trap representations. Wanted to avoid any discussion of what happens if we have a nullptr_t object and what happens. This allows parameters of type nullptr_t but you can't access them because they are not interesting as we know the value anyway so it makes no sense to use the variable in an if for or whatever.

The only thing it leaves open for the ABI and ABIs that are shared with C++ will take those forms for others may be more lose. A function that has nullptr declared a parameter may skip that parameter and an ABI can be created that does it like that just don't want it to go into any discussion or need for specifications for this sort of thing.

Maurer: Paragraph "No object can be created with this type" is not the same as C++. They use a special type with the following properties. In general in C++ we have opted to make it a regular type that is isomorphic with what * so it just has the type of what the right hand type.

Jens G: so there are conversion rules for tertiary operations

Maurer: If one part is a nullptr type thing and the other thing is a pointer type then the result type is an int *. If you take the nullptr branch at runtime it is converted to int *.

Jens G: so basically the rule we have for void *

Mauer: but C++ we don't convert to void * but in C we do. so that would be a difference. btw, you are worried about trap representation C++ says that they never look at the bits because you already know what it is because you never look at the bits.

Jens G: Would it hurt much having it like it is here? The idea here is to have it under syntactically where it is replaced before you can do anything with the type for example if you use int 0 where everything is done at compile time and you never have types of such things. The type immediately vanishes after compilation.

Mauer: Not having objects of that type is a difference but shouldn't matter that much as people shouldn't create types of nullptr_t as it is uninteresting. Not sure if the change in specification approach would create some differences. The difference is substantial enough that it might cause some corner cases.

Chat:

Jens: you seem to have locked up

https://eel.is/c++draft/expr.call#12.sentence-5

From Martin Uecker to Everyone: 01:21 PM

all representations could mean the same value

From Ben Craig to Everyone: 01:22 PM

with overloading, there is some use in having nullptr_t parameters. Without overloading, I'm less sure of the utility. I don't know enough about _Generic to know if nullptr_t becomes interesting there or not, or what the tricky parts are.

From Jens Maurer to Everyone: 01:22 PM

Aaron: This is just for varargs functions.

From Aaron Ballman to Everyone: 01:23 PM

that's why I needed it :-D

Aaron Ballman: varargs…

Jens: you can do varargs trip with nullptr trip. You could have it as a stdargs argument in C++ but otherwise you would varargs with another type other than what you fitted into function call and you can only use nullptr_t and you will need to know that the thing that arrives is a nullptr_t and you can't use it as an end marker.

AB: In C++ if you use nullptr it is converted from nullptr_t to a void* so it can be used on the other side as a null pointer so need something like that in the paper.

Jens G: You can look at void * as an int *. Varargs you can't pass a signed value and can't switch unsigned type as long as it fits, so you can't put in a negative and use as an unsigned type then you are screwed. int * is not guaranteed to have the same representation as void *. Varargs has a special rule that allows void * and char * to be interchangeable but needs to be explicit. Any other pointer type there is no chance.

AB: part of the reason this comes through is that you might need a nullptr_t as a function call and you might actually want it to be a nullptr_t. Common implementation is overloading in C and it could come up in practical terms. Otherwise we making implementors guess at what they should do.

Jens G: Finds that varargs is one of the worst interfaces in C and is only there because scanf and printf.

Martin Uecker: Same comment as Aaron not a fan of adding to nullptr in C because it converts and it is difficult to convert so prefer not to add types. If you add them you can use them for generic programming but then it is better if you can create objects like regular types because you might want to use it in a macro.

Jens G: understand that WG14 is something that people wanted. Best idea to have the type with the least possible specification to not constrain ABI too much so this is what came out of it. I'm not really convinced that we want to create variables of type nullptr_t.

AB: Haven't seen code where you create a variable of type nullptr_t but it can be passed through but that may be in Template code so less of a C problem and maybe on the fence.

Jens G: Example of generic with example even in C you can do generic programming with this type for example you can declare a function or specialization of our functionality that receives a nullptr_t type and then a generic type which just receives a pointer of your target type and they connect it with a _Generic expression with a choice according to the type and you get the right function pointer type and in C now you can do fancy things like call the function with zero which would take the function branch but if you use it with nullptr it will use the other function and avoid the patch code. You can also make it a constraint violation to pass in a zero and only allow a nullptr.

Ben: C++ has at least one sort of type thing which is a special case of just void which causes all sorts of generic programming problems so there has been proposals to allow people to create types.

Jens G: void is also an incomplete type that cannot be completed so you can just add some additional properties that you can use it in generic programming.

From Martin Uecker to Everyone: 01:40 PM

most of us avoid _Generic ;-)

void should be split up into a unique and the empty type

Jens G: Might be some incompatibility on C++ without specific uses cases. Basically fine with feedback.

AB: What are the next steps?

Jens G: Next step is clear this is scheduled in 2 weeks. And the paper won't change before then so the same paper will be presented and mention the concerns that came up here and based on the feedback and either change in direction that is more conforming to C++ depending on what people tell me. Anything to push through and have a version that works and is not too nasty.

Aaron Ballman: Will plan on hearing paper again unless WG14 loves it.

P2338R0 Freestanding Library: Character primitives and the C library

Ben Craig: P2338 been working on C++ side trying to improve the state of free standing on library and language side. Gotten to the point in the library where C and C++ overlap. Want freestanding to support the maximal subset of the library that doesn't require OS interaction or space overhead. Different from status quo in which we have nearly nothing in free standing. Trying to improve experience for microcontroller instead of embedded which means too many things.

Needs to avoid exceptions, RTTI, thread-local storage, heap free storage, floating point, thread and OS sync including non-lock-free atomics, files, console IO, networking, etc.

Floating point should be avoided because of space overhead and because it is wrong in Kernel because you can corrupt user mode state.

Lots of other stuff dealing with threads because you have to use a lock which is an OS resource.

If it doesn't hit these features Ben wants to include it.

On the library side there are shared headers and optional free standing added in C++23 which add fenv.h and math. h and may have intentionally pulled in locales.

The paper being presented doesn't address these issues but we may have pulled more into free standing than we intended.

Robert Seacord: uncomfortable with problem not being addressed.

Aaron Ballman: One possibility is that we can discuss on reflector and cc C floating point group that they brought this in. Might be able to nerd snipe someone into.

Action Item: Ben will ping WG14 reflector to make sure that happens.

Locked atomics are in, some of cstdlib is in. _Exit is implemented as part of quick_exit so it's getting dragged along.

operator new and delete is in the core working group so that they can have implementation defined behavior on free standing so they don't need to do anything.

Stuff to add in freestanding. Additions cstdlib, cintstypes, cerrno.

size_t is already there

actual additions are going to be bsearch, qsort and abs(int) but not other overloads.

Robert Seacord: C is moving in the direction of removing interfaces that use intmax_t

Ben: OK with not including.

Ben: add functions that don't use errno according to C standard

Robert Seacord: POSIX references errno a lot

Ben Craig: According to C there are a bunch of functions that don't use errno.

errno has problems because it uses thread local storage.

memccpy is not in c only C++

<cwchar>

Notable omissions include errno, strtol, isspace and stuff uses local, not expanding exit functions, setjmp/longjmp.

adds wcstok. a good implementation

atoi uses locales and thread local storage.

Can add lock free atomics.

Atomics are optional in C anyway, so people can add if they want to or leave it out.

Must do all these things to be compliant.

Jens G: Can't add atomic because it is also optional for hosted implementations

Ben Craig: existing experience. musl, newlib, and uclibc-ng include all of the C parts commonly used in embedded.

memcpy is the big one really want to have that in free standing. all the string parsing stuff is there in the windows kernel.

Breakages there is already wording in C Standard that you can't add your own <string.h> but can define their own memcpy or name it slightly differently.

Aaron Ballman: Meeting coming up in June and another meeting in October. Need to be feature complete by the end of October or maybe the end of December. Won't be new features that will come in and after that. Also our schedule is also full for June and October.

Jens G: Basically the agenda is already full and it is up to us to fight for more slots. Not too worry about this hitting the C23 deadline although we like this. Time for papers in the 2022 2023 time frame. Would like to get feedback if this is appropriate because free standing won't want to add. Maybe these implementers.

Ben: Want to get feedback.

Aaron: One of the questions might be why should we mandate as implementations. Might want to focus on things that are ridiculous that everyone requires.

Ben Craig: Could take the approach of taking the high value line. Might have a decent philosophical break. Not a huge need for wchar functions. The string.h stuff is going to be used much more.

One of the goals of this proposal is to make it possible to write portable free standing libraries.

Jens: at least two different communities of standing communities including microcontrollers. Get some good feedback from a really small compiler for small devices. Guys like IBM that run mainframe as free standing. But it would be good to speak to people with these compilers. Hubert Tong has seen these papers and is involved.

AB: Might want to talk to Philip Krausse who builds with really tiny devices. Philipp Krause (SDCC: http://sdcc.sourceforge.net/)

Ben: Been aiming at the 10K crowd. The smaller you go even the core language stuff doesn't work. Functions like qsort can get a little bit big. How small does the implementer get it. If you don't call it you shouldn't have to pay for it. If it doesn't fit you don't use it.

Jens Maurer: These are all pure functions and not including functions that don't use errno. If you don't use it, it doesn't appear on the device. The burden is on the implementor to provide these functions. Might as well get them from BSD or somewhere so you don't need to do all this yourself.

Aaron Ballman: Very motivated.

Ville Voutilainen: Philosophically C has optional parts. Wonder if you can achieve goal by allowing rather than requiring?

Ben: Hosted and freestanding. Put everything under a feature macro like minimal library. So we can say C++ requires at least the minimal library of C.

P1949R7 C++ Identifier Syntax using Unicode Standard Annex 31

Steve: Set of slides from EWG C++ working group.

C++ identifiers using UAX 31

This is going to plenary shortly. Adopt Unicode Standards that identifies the pattern to identify valid identifiers.

(XID_Start + _) + XID_Continue

Problem that it fixes NL 209 allowed characters that are zero-width and control characters that lead to impossible to type names, indistinguishable names, and unusable code and compile errors.

Status quo: we allow other weird code points. The middle dot that looks like an operator, many non-combining modifiers and accent marks that don't make sense on their own. Swift was using the same white list that C++ uses from the late 1990s. Tone marks, box drawing questions, greek question mark that looks like a semicolon and dingbat symbols. Very little consistency in what's allowed and what's not.

UAX31-Unicode identifier and pattern syntax follows the same principles as originally used for C++, actively maintained, stable.

Going to remain backwards compatible and not take away identifiers.

XID_START and XID_Continue are Unicode database defined properties, closed under normalization of all four forms, once added won't be removed.

The emoji problem. The emoji-code points that we knew about were excluded, we included all unassigned code points, status quo emoji support is accidently included and broken.

Male and female symbols are disallowed so the generic variants are not allowed. Can't have female construction workers.

Emoji are complex not just code points, need grapheme cluster analysis, not stable, that that were false and now are true and the emoji property could be removed.

Making identifiers not identifiers would be bad. Detecting emoji is difficult. The Unicode standard provides a regex that will reject non-emoji, but does not guarantee a valid emoji sequence. Not clear how much of the Unicode database would be required.

Some surprising things are emoji including digits and asterisk. These can start emoji sequences.

We would have to be inventive in an area outside our expertise. Adopting UAX31 is conservative approach. Another area of concern is scripts. Some scripts require. Apostrophe and dash are not in identifier set. Programmers are used to this and don't notice anymore.

Zero-width characters are excluded by UAX 31. Status quo allows invisible characters. However, zero width joiner and non joiner are used in some scripts. People accept that zero width separators can't be used in names. Expensive solution to dealing with scripts. Identifiers can be checked for what script the code points in the identifier are used and the rules for allowed characters can be tailored. SG 16 does not recommend this.

Jens G supports this broad list.

Jens M Annex D normative.

Steve: NFD leads to bigger symbols. NFC for comparison functions does more to differentiate different identifiers. NFK would tend to make different symbols compare the same.

Steve: C++ developing some new components in the lexing.

Jens G: Basically this is in the lexing so in early phases of compilation would almost be in favor of pull this out and have a common document for preprocessing in C and C++. The text is almost the same and could find a common superset and spare a lot of headaches.

Steve Downey: Actually have some stuff in progress that would underpin the model of how text is processed. Everything except the source code set is converted to U form (which no one ever does). If we shift to the C model there is probably more we can share.

Aaron Ballman: limits on length of identifier might have three glyphs showing up on the screen. Will this expand.

Steve: Code points. Characters up in astral plains that people needed for some reason. There will be characters outside the BMP.

GCC 9 finally fixed their UTF-8 bug. Until recently there wasn't portable code that uses characters outside the basic character set. Clang has always supported UTF-8. Most style guides are likely to strongly limit the general capabilities.

Robert: Annex D

From Miguel Ojeda to Everyone: 02:52 PM

MSVC has /utf-8, so I assume internally they can handle anything

Wrapup

Aaron: I'll schedule the next meeting and get minutes for this meeting out shortly, thanks everyone for coming.

End at 3:00 pm EST