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

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2707
SG Meeting Date: 2021-04-09

Fri Apr 9, 2021 at 1:00pm EST

Attendees

Aaron Ballman	WG14 WG21	chair
Tom Honermann	WG21	
Jens Maurer	WG21	
Clive Pygott	WG14 (21)	
Michael Wong	(14) WG21	
Hubert Tong	(14) WG21	
Hans Boehm	(14) WG21	
Jens Gustedt	WG14 (21)	
Miguel Ojeda	WG14 WG21	
Robert Seacord	WG14	
Will Wray	(14) (21)	
JF Bastien	WG21	
Ben Craig	WG21	scribe
Nevin Liber	WG21	
Martin Uecker	WG14	
Damien Lebrun-Grandie	WG21	
Ronan Keryell	WG21	

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

Agenda

Discussing the following papers:

P1152R4 Deprecating Volatile (http://wg21.link/P1152R4)
WG14 N2655/P2311R0 Make false and true first-class language features (https://wg21.link/P2311R0)

P1152R4 Deprecating Volatile

Champion: JF Bastien

Hubert: Volatile atomic in C, there is a compound assignment with volatile atomic. Understanding the semantics we didn't need to do for C++. I think we should deprecate it.

JF: C has compound assignment for volatile atomic, and we should deprecate that? It's non tearing. I think it has the semantics people expect it to have, so I don't think we want that behavior.

Hubert: compound assignment being confusing is multiple accesses, so if you have a hardware loop, then that is confusing.

JF: I think you are right there, I agree with you.

Clive: The simple assignment text seems like it might be a problem.

JF: When you have a = b = c and the b in the middle is volatile, then that's a problem. This has changed over time.

Hubert: I think for C, we shouldn't change it. In C++ we translate the lvalue'ness over, where we don't do that in C. The confusion in C++ isn't translated to C. The rationale would be more C++ compat vs. there being a reason.

Clive: It sounded like it was deprecating `int i = something;`.

Maurer: The discarded value part addresses that.

Clive: In that case I don't have a problem.

JF: I think we want to ban the same thing in C, because when I see a = b = c, I expect it to not to do weird volatile things.

Clive: Or make it clear that it's the same as a = b and a = c.

Hubert: That's already clear in C.

JF: C has a very clear wording, I agree on that. It's confusing to readers though.

Aaron: The semantics in the standard is clear, but the semantics aren't clear to developers.

JF: The C++ standard has examples with dinosaurs.

Aaron: Is the intent that op++ and op-- be UB or ill-formed?

JF: It won't compile.

Gustedt: atomic and volatile, would be good if this could be clarified for the C standard. We all have these volatiles in the interface for atomic stuff, and I'm not sure that they are needed. Not sure if we should have atomic with and without volatile. For assignment, the difference in C++ is that you get an l-value back that you can assign to, so for volatile, this is something really weird. In C, it's just a normal value that you can't assign back into it.

JF: The behavior in C++ changed between 98 and 11. Instead of writing `a = b = c;`, they should write either `b = c; a = b;` or `b = c; a = c;`. I don't know as a reader which the author intended, even though the C semantics in the standard is clear.

JF: We do deprecate volatile atomic when it isn't lock free. That may resolve some of the issues. You are either going to have an embedded lock or lock table somewhere. Shared memory non-lock free atomic makes no sense. If it's not lock free, it will give you a deprecation warning, with an eye towards making it not possible in the future.

Martin: Normally, we don't deprecate a feature just because we don't want people to use it. We'd expect compilers to step in and warn as a QoI issue. Maybe in 10 years we would do something about it.

JF: Agree that is how WG14 normally works. Hoping that C will do more warnings. I think WG14 should have a different approach here. I'm not trying to give you a nicer tool. I'm trying to reduce the number of bugs in code. You may be able to reduce this in terms of product recalls and lives saved. A newer string.h would be nice, but we wouldn't want to deprecate an old one to keep compatibility. This is in a different category.

Martin: long term, you wanted a template load thing. You couldn't do that in C. Maybe we could do something similar with a different syntax. Will we later diverge anyway.

JF: C++ and C should have different syntax for some things. In this case, we are operating at the same abstraction levels. Paul McKenney's paper on this is something that he wants to use in the Linux kernel. So whether we end up using generic in C, or having different syntax, I'm not sure. Maybe the 'T' is passed as a parameter to a macro.

Martin: What is the motivation behind getting rid of the qualifier?

JF: At point of use, you don't see the qualifier. Having a function that does the read and write makes it clear at the point of use, and not just the point of declaration. We also want to be able to say that certain memory locations always need to be volatile accesses.

Aaron: I'm in favor of doing this in C, and I would recommend that the C paper have very strong motivation for doing that deprecation. Feedback from actual users helps motivate WG14. WG14 is likely to balk at doing this deprecation without strong motivation.

JF: Won't be sending this to WG14 myself, but would be glad to have someone else do it.

JF: Function signature stuff never effected mangling, so even the C++ side of thing never affected ABI stuff. Also, it is currently just deprecated and not removed, so it is just uneven warnings. Coincidentally, that is also the source of the compound assignment issues. A C header doesn't diagnose compound assignment, so the vendor doesn't fix it. But you build it in C++ and it diagnoses.

Gustedt: Unrealistic to get this into C23, unless we have a physical meeting in Oct, or have agreement on the direction of this change. Unlikely to go through in one session.

JF: A volunteer to write the paper would help, but polls wouldn't help right now.

Seacord: I think the paper as is today is probably too weak to get through. We would need to see more clear explanations, as well as what might break existing code.

JF: Just a diagnostic, so technically, no breakage. If you turn warnings into errors, then yes, code will break. What I would do with a paper is take what we deprecated and have the concrete examples.

WG14 N2655, P2311R0 Make false and true first-class language features

Champion: Jens Gustedt

Maurer: Are these now really keywords, or not? Or can I not detect it because I can't write ifdef true / false anymore? Keywords are reserved tokens in the token stream

Gustedt: But not in the preprocessor

Maurer: named constants in the proposed changes, does a table of reserved words exist in C?

Gustedt: The tokens would be added to the list of keywords

Maurer: That means after preprocessing phase, they would be given as proper keywords to the later phase 7 parser that parses C.

Gustedt: There's a possible option in the proprocessor that may cause the true to be replaced by itself.

Maurer: So it would be a bad program that does ifdef true?

Gustedt: I'm not sure.

Maurer: In C++ we allow constant expressions in the preprocessor. C and C++ differ here. We have a constant-expression grammar that can be used in the preprocessor. You don't get identifiers there yet though. True and false are not identifiers in C++, they are keywords there.

Gustedt: This sounds similar to what C is doing, but the results should be mainly the same. For `#if true`, it would be special in the preprocessor as it would need to be replaced by literal 1.

Maurer: I think c++ handles that case by making all the expressions fit in intmax_t. I think what you have is compatible with what we do in C++.

Aaron: Was wondering how the constant-expression hookup works.

Ben Craig: C++ removed op++ and op-- from bool. You don't need to change it here, but it's something that may be worth considering while messing with bool.

Hubert: C preprocessor wording is very similar to the C++ wording. It uses constant-expression as well. I may have missed the part where it tries to put it through further phases of translation. The wording may be able to be very similar in the two standards. If you make them keywords and treat them as constant expressions, then true and false already exist, so you could just evaluate true or false without

turning them into a number.

Aaron: We need the numerical value for conversions anyway, don't we? In case I do `a + true`?

Gustedt: This is new land for C because we haven't had keywords with values. I would prefer to have it explicit.

Hubert: Different wording may cause more problems. If you don't touch true and false and let it be converted to a token, then it is all well defined.

Gustedt: If you have it in an if expression, it's not converted to tokens?

Hubert: p7 of 10.1.7, conditional inclusion.

From Jens Maurer to Everyone: 01:09 PM
C++ cpp.cond p11: After all replacements due to macro expansion and evaluations of defined-macro-expressions, has-include-
expressions, and has-attribute-expressions have been performed, all remaining identifiers and keywords, except
for true and false, are replaced with the pp-number 0, and then each preprocessing token is converted into a token.

Maurer: C says "all remaining identifiers are replaced with pp-number 0", we need to say "except for true and false". Then true and false are turned into tokens, which is what we want. We have very similar phrasing in C++.

Gustedt: I think I had that wording once...

Hubert: There's usual arithmetic conversions and promotion in C and C++ that need to take place. We want to make sure that those rules are the same.

Hubert: Expression as type of int, but it has a different range in the context of preprocessing.

Aaron: True and false are no longer going to be integer constant expressions unless we change something. And that feels in the same general vein. The shall have integer type wording may be causing us some pain.

Maurer: 6.6p6 defines a technical phrase integer constant expression, and true doesn't fit that definition anymore.

Hubert: C has some confusion whether bool is an integer type or not.

Gustedt: It is an unsigned integer in C.

Aaron: It is a standard unsigned integer type.

Hubert: We need to be more concerned then. The PP case is where it is more obvious with the range

problem. Usual arithmetic conversions may act badly. Need to check how bool promotes between the two languages, and I don't think we will get to the bottom of it today.

Gustedt: In C it's the lowest integer range and promotes, just not in the PP. Types don't exist in the PP, which are signed or unsigned which have the range of intmax.

Maurer: That's not how I read it. We have a difference of opinion in what the #if wording says in C. I think that you have type correct behaviors in #if statements with all the type rules, except they all have the same range. The alternative is that all the types are actually intmax and uintmax.

Gustedt: This didn't occur before now is that we haven't had literals with narrow type. Promotion didn't occur before now.

From Aaron Ballman to Everyone: 01:22 PM
all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, intmax_t and uintmax_t

Hubert: character literals do differ between C and C++ (int vs char).

Hubert: Outside of preprocessing, it's still an issue. bool is allowed to have a gigantic range in C.

Gustedt: Not more than int or char.

Hubert: char is allowed to have the same range as int / unsigned int. So bool promotion differs because C++ says it goes to int. On some platforms, C could go to unsigned.

Seacord: Current wording requires it to be unsigned. All unsigned integer types act as if they are uintmax_t.

Hubert: C could have a profile where booleans promote to unsigned, but in C++ always promotes to int.

Seacord: promotion rules would always need to promote to int. It says that all the values can be represented by int.

Hubert: 6.25 type doesn't say that.

Hubert: Indeterminate values of bool can be more than 0 or 1.

Uecker: Should change bool to not have any other values.

Hubert: That's a huge change, because indeterminate values don't have to behave as true or false.

Uecker: C doesn't have indeterminate values.

Gustedt: Trap representations?

Uecker: Everything other than 0 or 1 would be a trap representation by definition.

Hubert: I guess that's fine? But that's a bigger change than what the paper is presenting.

Maurer: Have the same problem in the phase 7 language as in the PP though.

Hubert: _Bool is already a type in C. We are now calling _Bool -> C++ bool. Now we are trying to say that _Bool -> C bool. That might be a problem because of the unsigned-ness of _Bool.

Maurer: C++ bool is a special type that always promotes to int.

Seacord: I don't think C should promote to unsigned int, it should always promote to int.

Maurer: If C fixes that, then that is fine, and would fix a lot of C++ interop. Then you wouldn't need the special preprocessor wording. It also fixes the odd word width situation.

Gustedt: Not convinced that we shouldn't fix it in the preprocessor. Can bool have different values than 0 and 1 or false and true?

Maurer: If they can, you can't observe it.

Aaron: as an implementation perspective, I am opposed to things that would diverge the preprocessor. Having one behavior in C and a different in C++ wouldn't fly, I would pick one and just be non-conforming on the other.

Aaron: Only concern is breaking existing C code.

Seacord: We should just use the C++ one.

Gustedt: Changing the language part, I would like to narrow bool to have the values false and true, and everything else traps. Then we see what comes out of it. Also worried about the procedural side of things. Which conversions we hit may not get things voted in this year. May have to wait until the next release to get true and false.

Aaron: I would rather see true and false go in C26, rather than introduce an incompatibility in the promotion rules.

Hubert: The macro that exists in the C++ version of the header, and in C, the macro may need to be retained by the header, or become a predefined macro.

Gustedt: That is one of the C questions.

Hubert: The bool_false_defined macro.

Gustedt: stdbool.h shouldn't disappear immediately with no content (or less content). It will survive for some time.

Hubert: The macro for bool_false_defined should remain.

Gustedt: If we want to remove the thing at some point, then someone needs to do that.

Aaron: Removing it could break code that is checking for that macro.

Gustedt: Do people use that with ifdef or if? Never understood what it is for.

Hubert: With feature test macros in C++, we always ask how it would be used. This feels like a feature test macro, but don't know how it would be used.

Ben Craig: Could test for bool_false_defined to undef true, then make a function called true.

Aaron: 7.18p5 lets users redefine true and false, need to fix that.

Gustedt: We would be making true and false keywords, so we will be breaking those use cases. So users shouldn't be undefing and redefing true, false, bool. Otherwise we would just leave it as is.

Gustedt: Should ask wg14 about changes to conversions in C++ is sufficiently unambiguous to be carried over to C.

Seacord: Need to change this paper for the June meeting to answer this question.

Hubert: C standard should clarify that the promotion of a boolean to unsigned does not occur, however they do it.

Aaron: Anyone that disagrees with that point? (The room agrees)

Hubert: So do we want to use the C++ version of the PP wording. After that we have the open question of the header. There's the one macro that C++ defines bool_true_false_are_defined. Maybe this is the one liason item that is up in the air. Could say it's not defined, defined in the header, or pre-defined. C++ chose to put the define in the header.

Gustedt: stdbool.h header is a compat header with C. Now, driving this back into C as a requirement of what to do sounds backwards to me.

Aaron: I don't see it as a driving requirement, so much as being worried about code breakage. Regardless of how we address that, we should have some discussion in the paper, and ensure that wg21 needs to do the same thing, because it would be silly for C++ to retain it if wg14 removes it.

Seacord: Shouldn't we deprecate the macro?

Gustedt: Saying that the obsolete header provides the obsolete macro would make people happy?

Aaron: Other action is to address P4 of 7.18, and saying what we want to do with those provisions.

Gustedt: In C you have the right to redefine your keywords, but why not these? But the idea is to basically not allow this for true and false and all the other keywords we are changing. Change 4 asks the

question if we should let false and true be defined and undefined.

Hubert: If we let them be defined, then that would be news to WG21.

From Robert Seacord to Everyone: 01:54 PM
so this is C++? A prvalue of type bool can be converted to a prvalue of type int, with false becoming zero and true becoming one.

From Aaron Ballman to Everyone: 01:57 PM
[conv.prom]p6 for those following along at home

Wrapup

Aaron: Will send meeting minutes for Apr and agenda for May shortly.

End at 2:59 pm EST