Draft Minutes for 15 November – 19 November, 2021

MEETING OF ISO/IEC JTC 1/SC 22/WG 14
WG 14 / N 2914

Dates and Times:

Monday,	15	November,	2021	14:30 – 18:00 UTC
Tuesday,	16	November,	2021	14:30 – 18:00 UTC
Wednesday,	17	November,	2021	14:30 – 18:00 UTC
Thursday,	18	November,	2021	14:30 – 18:00 UTC
Friday,	19	November,	2021	15:30 – 18:00 UTC

Meeting Location

Teleconference

1. Opening Activities

1.1 Opening Comments (Keaton)

1.2 Introduction of Participants/Roll Call

Name	Organization	NB	Notes
Aaron Ballman	Intel	USA	C++ Compat SG Chair
Alex Gilding	Perforce / Programming Research Ltd.	USA	Scribe
Barry Hedquist	Perennial	USA	PL22.11 IR
Bill Ash	SC 22	USA	SC 22 Manager
Bill Seymour	Seymour	USA	Prospective Voting
Clive Pygott	LDRA Inc.	USA	WG23 liaison
David Keaton	Keaton Consulting	USA	Convener
David Svoboda	SEI/CERT/CMU	USA	
David Vitek	Grammatech	USA	
Elizabeth Andrews	Intel	USA	
Fred Tydeman	Tydeman Consulting	USA	PL22.11 Vice Chair
Freek Wiedijk	Plum Hall	USA	
Lars Bjonnes	Cisco Systems	USA	
Maged Michael	Facebook	USA	
Martin Sebor	IBM		
Paul McKenney	Facebook	USA	
Rajan Bhakta	IBM	USA, Canada	PL22.11 Chair
Robert Seacord	NCC Group	USA	
Victor Yodaiken	E27182	USA	
Aaron Bachmann	Austrian Standards	Austria	Austria NB
Dave Banham	BlackBerry QNX	UK	MISRA Liaison
JeanHeyd Meneide	NEN	Netherlands	Netherlands NB
Jens Gustedt	INRIA	France	France NB
Joseph Myers	CodeSourcery / Siemens	UK	
Marcus Johnson		USA	
Martin Uecker	University of Goettingen	Germany	
Miguel Ojeda	UNE	Spain	Spain NB
Nick Stoughton	Austin Group, ISO/IEC JTC 1	USA	Austin Group Liaison
Peter Sewell	University of Cambridge	UK	
Philipp Krause	Albert-Ludwigs-Universitat Freiburg	Germany	
Roberto Bagnara	BUGSENG	Italy	Italy NB, MISRA Liaison
Ville Voutilainen	The Qt Company	Finland	Finland NB
Wenge Rong		China	

1.3 Procedures for this Meeting (Keaton)

We hold straw polls instead of formal votes; guests may vote.

We do *not* want multiple conversation threads; do not split the audio and chat. Do not have more than once conversation at once.

1.4 Required Reading

1.4.1 ISO Code of Conduct

1.4.2 IEC Code of Conduct

1.4.3 JTC 1 Summary of Key Points [N 2613]

1.4.4 INCITS Code of Conduct

1.5 Approval of Previous WG 14 Minutes [N 2803] (WG 14 motion)

Tydeman: some corrections were needed. Typos were sent to the editor. Svoboda is fixing in the background.

Tydeman moves, Ballman seconds, no objections.

1.6 Review of Action Items and Resolutions

DONE: Keaton: Notify the author of N2684 of WG14's poll.

DONE: Seacord: Submit a paper standardizing that if the arguments to `calloc()` wrap when multiplied, `calloc()` shall return `NULL`.

DONE: Svoboda: Write a proposal to clean up C standard on wording on integer overflow (especially for unsigned integers

1.7 Approval of Agenda [N 2866] (PL22.11 motion, WG 14 motion)

Not counting the extra days.

Tydeman moves, Pygott seconds. (both motions)

Typo in the agenda, Friday begins at 1430 as well.

Keaton: I received many objections to the Dec 9th as well as my own conflict, so propose Monday-Wednesday only in December.

Tydeman: CFP cannot do Wednesday.

Krause: *huge* backlog, we need the extra days anyway, have to expand somewhere.

Gustedt: we have a really tight schedule with effective deadline of end of the year. Need to give original work a chance to meet this.

Ballman: cannot justify the extra time: a week of preparation for a week of work. May need to tell users it’s not happening. How long until the next C? 3 years or 10? Will influence the rush.

Bhakta: agreed, mailing one month before to give time, rushing work in does disservice to authors. Don’t shovel stuff in, make the Standard good. Alternatives include extending the deadline to do justice to the work.

Seacord: favour December, Spring will be in-person and cannot extend as easily, much harder to arrange.

Gustedt: do not agree about prep time – all papers are known about now. The fixed deadline of Oct 15th must be respected, otherwise is unfair.

Bachmann: agree that new proposals met the deadline and must have a chance.

Ballman: I don’t have time to attend at all.

Keaton: this is sufficient objection to not do December. I have proposed an extra week for January/February (non-consecutive because of C++) as way to clear the backlog. N2864 revises the C23 schedule and puts the deadline after January. We can do this much without permission, would need an extension from ISO to do more.

We will not have the extra days but we *will* make sure the new papers get time.

Ballman: would an extension still ship in 2023?

Keaton: yes, but not in August. I would worry about the public perception of a release slip. Officially an extension is 9 months, but we can finish early.

Krause: is there any majority for two days in December?

Bhakta: strongly no, what’s the need to rush if we can still get it into C23?I want time to gather feedback.

Ballman: the agenda makes this hard to support (“very aspirational”), there’s a lot of editorial stuff on the schedule. Not all of this needs committee time.

Voutilainen: hard clash with Monday, Finnish independence day.

Keaton: Monday-Wednesday of next week?

Krause: schedule clash, that is too soon.

Ballman: not available.

Keaton: so we need to extend the schedule. I didn’t anticipate so many submissions – I do not want to step on people. We will combine etra days in the future with the ISO extension as needed.

No objections to just five days. Agenda approved.

1.8 Identify National Bodies Sending Experts

Austria, Canada, China, Finland, France, Germany, Italy, Netherlands, Spain, Sweden, United Kingdom, United States.

1.9 INCITS Antitrust Guidelines and Patent Policy

Guidelines were observed.

1.10 INCITS official designated member/alternate information

Check with Bhakta if unsure.

1.11 Note where we are in the C23 schedule [N 2864]

Second element removed.

Fifth element proposed.

No papers will be discarded.

Gustedt: authors also need time to integrate comments.

Keaton: Dec 31st is *no longer* the hard deadline – unsure when, but it will be later.

2. Reports on Liaison Activities

2.1 ISO, IEC, JTC 1, SC 22

Keaton: JTC1 plenary ended today; resolution for a new adhoc group asking to restore free availability of documents according to old rules of free if meeting criteria. (not C) Problems for WG23 as now only a IS and with tight criteria. Keaton is in the group and pushing back.

Ballman: are you pushing for ISO C to be free?

Keaton: no chance, wouldn't be allowed even under the old criteria. Only applies to Standards that actively drive other Standards.

There are business model discussions, years away; hard because Nbs sell documents, which discriminates against less wealthy groups. Will change but must be gradual.

Gustedt: JTC1 can state a principled position w.r.t programming language Standards; they are special within ISO and not like other Standards, still valuable.

Keaton: we are doing that, we are a "troublemakers" group asking for this as a separated issue from rolling back changes.

Myers: w.r.t business model – JTC1 distinguishes between Standards like languages that are even useful to the average reader, not equipment-based.

Keaton: JTC1 has a culture of openness making it different from the other groups. ISO is not happy but will eventually have to change the model.

2.2 PL22.11/WG 14

Bhakta: PL22.11 has two votes that will take place later this week.

2.3 PL22.16/WG 21

Ballman: we continue on track for C++23, stuff has been approved in October. C++23 will not be a giant change, but is growing.

There have been four SG meetings; WG21 is rolling up the TS18661 CFP work, which CFP attended to ensure compatibility.

Thanks to CFP; thanks to all WG14 attendees, who have been really useful to the WG21 members.

2.4 PL22

Keaton: nothing to report.

2.5 WG 23

Pygott: we are translating into a IS. Nothing is happening in the C area, working in the language-independent area. A work item is out for a vote.

2.6 MISRA C

Gilding: TC2 will be finalized next month, AMD3 will be frozen in February but is not final or edited.

2.7 Austin Group

Stoughton: a new revision of POSIX is being developed for 2023. POSIX 23 builds off C17; now on the final draft.

2.8 Other Liaison Activities

None.

3. Study Groups

3.1 C Floating Point Study Group activity report

Items scheduled for Wednesday.

Bhakta: no new proposals for C23 except to fix broken things. Not adding Infinity macros, large change that implementations have already handled.

Good discussion with the C++ group; edge cases with differences, some due to valid language differences and some naming falls on C++.

Tydeman: many CFP papers are editorial – is there no way to process these through the reflector and not waste Zoom time? e.g. the `remquo` response could have been done by mailing list.

Keaton: you can send them straight to the editor.

Bhakta: if there's a chance it has semantic meaning, it's not editorial even if it's small.

Gustedt: some of these won't take long. Often discussed on the reflector, could we schedule vote-only papers? We have seen these shrink together in the past.

Keaton: if we dispense with them quickly we pick up others from the backlog. Don't want to preclude people from studying under the two-week rule. Well-discussed papers won't take up time.

Myers: typos have the option to be handled by GitLab MR. This can be very helpful to the editors.

Gilding: CFP papers have high semantic density even when there's little pushback.

3.2 C Memory Object Model Study Group activity report

Not much going on at this time. Discussion tomorrow.

Ballman: when the group meets, there is an invitation from SG1 in C++ who would like the Memory Object TS discussed there too. Please meet with Olivier. Need to require Sewell.

LLVM DevConf is interested in implementing the TS and have scheduled to meet the SG.

3.3 C and C++ Compatibility Study Group activity report

Discussed above.

3.4 Undefined Behavior Study Group activity report

Svoboda: we are set up, with a mailer, a wiki, and 12 members. We are soliciting more members.

Steenberg is working on an educational TR about “surprises”.

Annex J.2 to add code examples, taking from TS 17961, inspired by C++ p1705. A draft and work request expected by January.

Myers: expanding J.2 – are you annotating each UB as a property of the whole program or just one execution, determines whether the program can be rejected?

Svoboda: some are program, some are portable. For instance signed overflow is an execution property, can’t reject at translation; conflicting `extern`s can be rejected during linking. Part of the work is a table for J.2 – this can be a field. A lot of other different properties are already noted, there will be many columns.

4. Future Meetings

4.1 Future Meeting Schedule

Keaton: is the extra week after C++ in February possible? (virtual)

Krause: hard to do the Friday but if we need it, it must be done.

No objections.

Keaton: this gives January and February equivalent capacity to a face-to-face meeting. Strasbourg is still face-to-face.

Ballman: are the dates tentative?

Gustedt: moved because of Bastille Day, can be moved again. University is OK but unsure about Covid, really tentative.

Krause: want to add days in the summer (there will not be free food). Two days before or after.

Gustedt: expect doubled time.

Keaton: can we use the Saturday for face-to-face?

Krause: sure.

Gustedt: will January/February be enough?

Keaton: no, it covers new topics and not all are continuing. Strasbourg is for continuing topics.

After that Perforce is still offering Minneapolis for Fall 2022.

4.2 Future Mailing Deadlines

With only three weeks difference, the mailings are to remain merged.

Ballman: for off-list questions, will the website still be doing document bundles? The last was October 2020.

Keaton: yes, it became confused due to fragmentation because of non-appearing dates in the registry. We don't have to bundle any more; this is a legacy of postal mailings. Grabbing the Agenda would be better.

Ballman: can I change the page to promote the full Document Log?

Keaton: please do.

Myers: what about the C-document system?

Meneide: I am still working on it. Giving myself until December 31st to complete it. Apologies for the progress but it is ongoing.

Keaton: it is greatly appreciated, thank you for the work.

Stoughton: will it issue empty papers for unsubmitted documents?

5. Document Review

Monday

5.1 Working draft updates [N 2731] [N 2733]

Working draft now covers all changes up to March/April meeting plus editorial changes. Moving to incorporate the changes from the August/September meeting.

No diffmarks at this time, will fix and issue them separately. All editorial suggestions have been logged and handled now.

Yodaiken: there was a change between C17 and this draft on side effects – a clause was dropped without attestation.

Meneide: couldn't identify this change, the wording was moved during the atomics work; will continue to search old diffmarks.

Myers: did we agree to include n2566? Thought this was agreed to include, GitLab shows as not-agreed.

ACTION: Keaton to investigate status of n2566.

Tydeman: looking at the document, the table of contents has mojibake.

Meneide: bookmarks are inserting an extra byte order mark. Cause is unknown but the issue is open.

Uecker: I recompiled the document from source and this disappeared.

Meneide: maybe this is a build issue – libiconv inserts byte order marks without asking. Hopefully I can update or someone else can rebuild.

Gustedt: can we replace an n-numbered document?

Meneide: requesting a new n-number is not difficult. This will include the other pending changes within two weeks.

Uecker: there was also a DR about Yodaiken's observed issue with the wording for lvalue-modification of objects.

Keaton: thanks to Meneide for this hard work!

5.2 Tong, `_Thread_local` for better C++ interoperability with C (C++ liaison) [N 2850]

Deferred.

5.3 Johnson, Length modifiers for Unicode character and string types [N 2761]

This is version 5 of the change. Width and precision affect the codepoints, not the code units (value rather than byte representation).

Keaton: we must have an n-document to discuss, because of the two-week rule. We can review and vote "along the lines" for future revisions.

Bhakta: (not following the update) I saw 16-bit mentioned UTF-16 as the only character set – Windows and other platforms do have other 16-bit character sets.

Johnson: those are very implementation-defined and I want `l16`/`l32` to be specific to Unicode.

Seacord: is there a reason this ignores UTF-8?

Johnson: mostly because `char8_t` is already supported well and didn't feel it needed more coverage.

Myers: the new versions address some of these issues, but – w.r.t wording, precision handling needs to be consistent with other format specifiers. It's inconsistent with every other handler.

Ballman: confused by the details of printing – if the execution character set is ASCII, and I print a `char16_t`, what happens? Or a `char32_t` out of the BMP-range?

Johnson: this is based on codepoints not code units, so there are no surrogate pairs to consider.

Bhakta: the modified description has "the array" without reference to an object.

Johnson: this is the string being accessed, but need to clarify wording.

Seacord: for the literals, `l` corresponds to a wide char – would `u` make more sense?

Johnson: originally we used `U16`/`U32` but found that these are reserved, so chose `l` to match `wchar_t`.

Myers: w.r.t ASCII, the paper needs to amend 7.21.3p14 “encoding error” which does not correspond to a multibyte character, this is the same issue as the other functions failing to convert. It’s a new kind of encoding error and should amend the definition.

Bhakta: `wchar_t` specifies the list must be a compatible type; this should require a conversion as the encoding in the same type might be different.

Johnson: agreed, it cannot just be the type but needs to be the meaning as well.

Seacord: w.r.t revision 5, this provides definitions for codepoint and code unit – not sure the definitions should be reproduced here, they are pulled from ISO 10646. If it’s normative should reference there and include in the paper to reduce confusion, or this should be informative but not part of the suggested wording.

Johnson: will remove from the wording.

Ballman: version 5 mentions wanting to retroactively add this to C11/C17. Hard to make this into a defect report.

Keaton: ISO doesn’t allow modifying past Standards. Old ones are withdrawn. We did vote to track “changes of interest”, but this would be the first one.

ACTION: Gilding to add n2761 to the papers-of-interest list.

Myers: to clarify, this is a new feature and not a defect, and cannot apply to old versions. Only unambiguously non-new features are defects. C++ is too aggressive here.

We do need a normative definition of codepoint/code unit though. If we import ISO 10646 it needs to be mentioned in clause 3, not just 2, to be normative in C.

Seacord: w.r.t *changing* old Standards – C17 is current, is it repairable? Though this is a new feature and shouldn’t qualify regardless.

Keaton: we do have the option to make minor revisions to C17, but those don’t have technical changes. It would interfere with C23, so we will not.

Johnson: I don’t mind either way.

Keaton: it should still go on the list.

Seacord: rehashing the wide char argument – nobody wants them, should they be removed? It’s a large change but removal is always easier. Does anyone actually want the wide char stuff?

Johnson: I avoid it but my code is new.

Ballman: I wouldn’t say I “like” it but it’s extensively used on Windows. The scale makes it impractical to deprecate.

Myers: there is a huge API for wide characters – are these APIs useful, and which should be added for `char16_t`/`char32_t`? Did anyone review the full API? Are there any missing Unicode APIs, such as Unicode classification functions? Are enough people interested to add them?

Bhakta: I know of a lot of uses of wide characters – used in conversion functions to do translation between character sets. A lot of code depends on it.

Keaton: personally used wide characters recently. They were developed with the Japanese National Body and would need their involvement to consider removing them. Not part of this proposal.

Straw poll: (opinion) Does WG14 want something along the lines of n2761 in C23?

Result: 8-1-15

Seacord: I view the types as useless, can only be converted to multibyte. This is little improvement for something not truly supported at all. Unnecessary and incomplete, support should be complete or not present. There are no C APIs for these so they are not usable in C at all and only usable by third-party libraries.

Johnson: there is a lot of code, but it’s third-party.

Seacord: I would prefer comprehensive support.

Keaton: WG14 asked internationalization experts if they wanted this, and they said no, they would prefer to use specialized non-Standard libraries. Experts prefer to provide the APIs themselves.

Myers: wanting extra functions is similar to wanting more general equivalent APIs. Meneide is working on a conversions API for C23 (n2620).

Keaton: can the Abstainers explain?

Krause: the facilities are not a problem but I don’t see them as useful without the conversion and other APIs. Will check use after the API is available.

Svoboda: support is poor, want a decision to improve the support first.

Gilding: see a piecewise API as useful, we can start to build on it if each piece isn’t a problem.

Meneide: this is a problem in C++ and in other work. You can semi-trust UTF-8, but printing is more difficult for 16/32. Doesn’t cover everything but moves in the right direction. To echo the expert opinion, the existing APIs aren’t useful as designed, we shouldn’t standardize stuff that will break. No fundamental problem with the specification of printing and no need to redesign it.

Ojeda: like Svoboda, I want to focus more on UTF-8. 98% of web pages are Unicode, but 96% are UTF-8, so we should focus on the useful part.

Seacord: agree with Meneide that APIs for narrow characters are problematic, for wide characters are problematic, and that they don’t fit the Unicode use case anyway. The whole library needs to be rethought. We would be adding to a base we don’t understand and may have to deprecate. Neither API works for string handling.

Myers: I would like a poll on whether we want a larger Unicode API, so long as it makes sense (not a literal copy).

Bhakta: new proposals can submit new APIs.

Was this consensus? There was a mixture of support and abstention, nothing prevents returning the paper with a revision.

Keaton: considering the replies the position is in favour of seeing a new version of the paper.

5.4 Wiedijk, Types do not have types [N 2781]

This is a small editorial change to the previous version.

Gilding: does breaking up the italicised term of art matter?

Seacord: yes, is this introductory or important? It seems fine as it is.

Banham: this changes the meaning: “underlying type with qualifiers” implied by the original sentence is lost.

Seacord: compatibility only really matters for enums.

Keaton: 6.2.7 is a list, and this is just one option.

Wiedijk: “type of a type” is something that doesn’t exist, it is never introduced. I don’t see this as having different meaning.

Banham: what is being said here? This is meaningless.

Keaton: compatibility, if identical, is not exclusive.

Wiedijk: this is an inductive definition: the *default* reason for compatibility is that they are identical. This is just the lead-in, there are other reasons. I don’t want an intrusive change, should keep “the same”.

Seacord: the wording isn’t worse. The next sentence is untrue because the additional rules don’t exist. Would “identical” be better? The footnote uses it. But it’s fine.

Bhakta: this is an improvement. We should remove the italics because it’s not a definition. Change “their” to “the”.

Wiedijk: “have” still implies ownership?

Bhakta: it would be a way to keep the term of art there.

Keaton: the problem is in the first half.

Myers: the new wording is improved; compatibility is reflexive, symmetric, not transitive.

Gilding: `typeof` may confuse this. This looks like a reuse of old text.

Gustedt: italics are important; a semicolon might make the start of the options clearer.

Voutilainen: the change is correct if we drop the italics and not correct otherwise. There are other uses of “compatible type” that don’t mean this anyway.

Banham: is this talking about type aliases?

Keaton: it includes that but is not limited to it.

Ballman: I don’t want to lose the italic definition, as it’s referred to in e.g. generics, and it’s in the heading, so it should be defined.

Wiedijk: I propose keeping the italics and adding a semicolon which makes it clearer.

Straw poll: (decision) Does the Committee accept the changes in n2781, with `.` substituted for `;`, into C23?

Result: 17-2-4

Voutilainen: so now we have a term of art for “compatible”, but not for “compatible type”.

Krause: the term only applies to types.

Voutilainen: that’s not how terms of art work.

Wiedijk: we could italicise “type” separately?

Keaton: encourage editorial suggestions for changes without meaning.

Bhakta: does this violate the ISO rules?

Keaton: I don’t think so.

5.2 Tong, `_Thread_local` for better C++ interoperability with C (C++ liaison) [N 2850]

This moves to undo the change to keywords that brought in the lowercase spellings, because `thread_local` means something different in C and C++. In C, it is static to the thread, its lifetime matches the thread object, and it only needs to store. In C++, constructors and destructors require special code paths. C can always generate fast paths, but C++ can only do this if the constructors are trivial; nontrivial constructors require it to call out.

We want the visible semantic difference to remain. We want inclusion to indicate explicit buy-in by the user to the fastpath semantics implied by the `_T` spelling. C++ will have a way to specify both, while C will remain the same. Otherwise it is confusing to users which semantic is intended.

Gustedt: this was discussed in the liaison SG: the problem already exists and an incompatibility is not added by changing the spelling, as the macro is already there and already used in existing code. The possibility for C++ to detect fast paths is also already there – C++ can easily detect constant initialization and use fast paths for those initializers. The model differs in that C++ can make *some* declarations different, but that’s their problem.

Bhakta: agree that we want to make this more visible, we want users to see when they’re invoking something different, want to force visibility. You can’t always tell from the declaration line if construction is trivial. Users cannot tell by reading it, so this is a usability issue.

Gilding: we already encourage users to use the lowercase spelling by providing the header, so there is no visibility.

Bhakta: there is lots of existing code that dis-applies or undefines the header because it *wants* to draw the distinction. This is not just for existing code, it provides an option for new code. This restores the status quo and doesn’t take anything away.

Myers: the previous vote was “along the lines”, which would change to direction only.

Krause: I don’t see why this is special. Constructors and destructors apply to everything, including global variables.

Bhakta: the point is to make it clear C++-side that lowercase implies the expensive code path.

Ballman: are there any C++ implementations that support C semantics for the `_T` spelling, and not just an alternative keyword?

Bhakta: I know of implementations using the syntax but they provide C++ semantics. I expect this to mean “with the addition of constant initialization” – we *assume* existing compilers would change their behaviour.

Ballman: I would expect this to be seen as an ABI break.

Bhakta: IBM is used to ABI breaks on the C++ side.

Straw poll: (opinion) Does WG14 want to remove thread_local from the list of revised spellings in n2654?

Result: 9-5-9

Keaton: having checked, the ISO directives actually say *not* to italicize words.

Tuesday

5.5 A Provenance-aware Memory Object Model for C

Sewell: clarifying uninitialized reads. Discussed many options. Result is clear there is no clear preference within WG14 for any single alternative. Compilers give various different options, sometimesuser-controlled.

Covered last votes, but they were not super clear. Wobbly values had some significant support in the first poll, but not in the preference poll. a (allocation time nondeterministic choice), c.1 (plain UB), and c.2 (implementation per-instance choice to diagnose at compile time or give runtime error, else fall back to c.1) had some significant support in the preference poll.

It doesn't seem like we can reduce this set of options down to 1, or eliminate the distinction between address taken and address not taken. What can we possibly do?

Perhaps enumerate several distinct semantics and make it be implementation-defined in different circumstances. Leaving aside of which alternatives, would like a straw poll on the premise.

Seacord: sounds like a horrible idea because we're inventing here and letting implementations decide. Implementation-defined is for multiple implementations not agreeing on a common solution.

Sewell: that's the circumstance we're currently in today.

Seacord: so is this out there in existing implementations?

Sewell: not exactly these, but some form of them.

Seacord: this still feels like invention.

Sewell: but we're not really inventing.

Seacord: but only some of this are not in existing implementations, aren't those invention?

Sewell: we're inventing a specification to encompass what exists in the wild. We're not trying to invent features.

Seacord: perhaps we're getting too much choice and the SG should give us the preferred approach instead of options?

Sewell: we'll get to the cohesive set once we validate the premise.

Seacord: this feels like design by committee because we're repeatedly picking from series of options.

Sewell: the SG is doing what WG14 asked us to do. The SG wishes there was a single choice they could present, but the fundamental tension is between existing implementation behaviour vs things that are not terrible for programmers.

Voutilainen: curious, from your perspective, how is this materially and practically different from plain UB? Does this buy us anything if an implementation picks "plain UB"?

Sewell: the hope is that implementations provide (a) or (c.2) rather than (c.1).

Voutilainen: they already can do so today, so does normative encouragement matter?

Sewell: we could ask compiler vendors what they plan to do.

Gustedt: not in favour of (c.1) as that's the current status for variables where the address is never taken. If we don't have (c.1) then this is a big accomplishment because then implementations have to at least diagnose missed initialization (with some false positives), but this tells users that their code looks fishy. That's progress over plain UB.

Voutilainen: overall concern is that if we have one option or a buffet of options, an implementation that wants to do something else that happens to be useful for diagnostics in practice. The plain UB approach leaves more implementation freedom to diagnose.

Sewell: we believe the options should allow all possible configurations.

Voutilainen: when we have the UB options, it allows implementations to just do whatever they did before. Would prefer we're be damned sure that list of choices really does encompass all possible configurations.

Worries about nonconformance. I don't know what implementation approaches are in the wild though, so I can't help narrow that down.

Gustedt: implementations do this already. floating-point nonconformance modes for optimizations. Important to fix the standard to *something* that's relatively easy to comprehend, then vendors can add their own modes which may or may not conform. But if something nonconforming is used a lot in practice, we can standardize that.

Bachmann: I think requiring diagnoses is not a good idea because there could be different code paths; false positives are a concern. Those make warnings no longer useful.

Sewell: important to note we're not mandating diagnoses. c.2 is either diagnosed or some stable deterministic choice.

Keaton: this kind of question is one where the only people to talk to you about it are the ones who object. So it's possible that the majority don't object. Could we see if people are generally in favour, and then look at the alternatives to see where that leads?

Sewell: fine by me.

Gustedt: too much false positives -- going to (c.2) where the address is never taken, and in that case, static analysis should be sufficient to cover that and not have false positives. This case should be rare (for variables where the address is never taken), so I don't think this will have many false positives in practice.

Bachmann: conditional with a function in it; without whole program optimization, the compiler won't see whether the path is taken or not.

Gustedt: yeah, users might need to mark code paths.

Keaton: I'd like to see where this goes.

Sewell: which approach do you think would be most fruitful?

Keaton: try the first straw poll?

Straw poll: (opinion) Does WG14 support the approach of defining several alternative semantics for uninitialized reads, so that compilers can document, in terms of these, which semantics they provide in what circumstances (depending on compiler options, on the storage duration, on whether the type is a character type or not, and on whether the address is ever taken)?

Result: 10-3-8

Keaton: general leaning is in favour, but a lot of abstentions. Could move on to the next phase which clarifies some of those abstentions.

Sewell: should it be made not just implementation-defined but checkable via a feature test macro? Is that uncontroversial? We also discussed removing (c.1) altogether. That would be a strengthening of the standard. Perhaps that now has support? We also discussed automatic zero init or automatic sentinel value init (which are both admitted by (a)), but automatic zeroing may give extra strength. Or we could follow the C++ direction of allowing uninitialized values to be read and written (with the target becoming uninitialized), but gives UB if used in any other way. However, `memcmp` of a `memcpy` may not be guaranteed.

Krause: the C++ direction might not be as bad if we used a variant if you read and write them as character types, that should work. Any other type would be the C++-like behaviour. That could be an option.

Sewell: that's a good point; it would be more complicated, but it could work.

Krause: I'm against removing (c.1) altogether, that makes too much of a burden on implementations.

Sewell: you might not get false positives; if the implementation cannot decide, it needs to at least be a deterministic behaviour.

Ballman: I think zero and sentinel init has a lot of implementation experience; it's the option I think we have the most experience with.

Krause: but that does have clear cost to implementations, is worried about initializing large objects.

Sewell: these options are not proposed as being mandatory, so that seems fine.

Keaton: I've used this option and personally have found it can be an optimization rather than a pessimization, at least in some cases. But if it's UB, then people can do automatic init if they want to.

Sewell: but it does give strength for code to depend on (at least for zero init).

Seacord: can we have automatic zero init, but add a per-instance opt-out kind of option? This would solve the "worry about init of a big object" concerns.

Sewell: we could, but that would be inventive. I think it could be useful though.

Uecker: gives some guarantees about not accidentally leaking sensitive information.

Sewell: interesting point, but that wouldn't apply to any padding, so it can still leak sensitive information.

Sebor: I don't think we'll get any one ideal choice that the *whole* community will accept. It will disadvantage somebody.

Sewell: the goal is to narrow down to the smallest set of options that would not disadvantage folks

Sebor: is the question do we outline all of these as possible implementation choices? So implementations pick the specific choice and document it?

Sewell: we outline some set of these as possible choices and the implementation documents which they pick.

Sebor: all of these are useful and are being used. Consensus here doesn't reflect community or implementor consensus.

Sewell: this is consensus on a set of options, not a choice of option.

Sebor: one option is removal; removing one affects implementations. Are all of these options, does an implementation choose and document? Providing choices should improve portability but this doesn't facilitate that. Providing a feature test macro doesn't help in the presence of `pragma` or LTO. A wide range of options doesn't help portability.

Sewell: I can ask a porting target if it provides an option I depend on.

Sebor: not all implementations care about porting to others, our goal is to facilitate that. Outlining options doesn't do that.

Sewell: why not? A user writes new code depending on zero-init, they ask the implementation if it provides it.

Sebor: to me, "portability" means moving code without change. "Let me read the docs to see what changes are needed" isn't that kind of portability. Implementation-defined is useful, but not the same as portable.

Seacord: agree with Sebor that having multiple options doesn't help. Option F1 is like Rust's direction, with `[[uninit]]` you can opt out of it; we seem to have moved on but I would like to see a preference poll and think the consensus would favour multiple.

Sewell: we preference-voted last time. We also did an acceptability poll, A and C2 won with non-trivial support for C1. F wasn't in the vote but implementations use it.

Seacord: 18-3 is consensus?

Sewell: it wasn't a poll to adopt.

Seacord: but we have a solution?

Sewell: not a mandatory option.

Seacord: might as well make it UB and give up if we're going to have 12 options.

Sewell: let’s look at support for option subsets.

Bachmann: A was optional last time – why not poll for making it mandatory since it had strong support?

Multi-way poll: Does WG14 accept [this] as the only allowable set of options?

Results:

(a, c1, c2)	12-10-3
(a, c2)	8-11-5
(a)	9-10-5
(a, c1, c2, f1, f2)	10-12-3
(a, c2, f1, f2)	8-13-4
(f1)	6-15-3
(c1)	13-8-2

Voutilainen: toying with an implementation and proposal that zero-inits everything that is not an array, an F1-variant also supporting opt-out with an attribute. Might carve out a palatable niche but needs discussion in the C++ context. Large objects incur a cost, and it is non-trivial to work out where the annotations go, so I think F1 is not feasible in the field.

Ojeda: F1 should come with explicit opt-in.

Sewell: we already have that in the language as `= { 0 }`.

Ojeda: I want the opposite, a default init with opt-out.

Sewell: that’s F1.

Ojeda: then F1 has support, if it’s an attribute.

Sewell: I don’t care about the mechanism.

Yodaiken: existing code assumes there is no init and that can be important. Should this imply that code needs to be rewritten? F1 or F2?

Sewell: only F1 alone needs that.

Ojeda: so opt-out wasn’t there before?

Sewell: it was added after feedback.

Sebor: what are we converging on? A straw poll?

Sewell: Yes/No on each option set.

Sebor: so does a set reflect practice or is it a wishlist? Do implementations do these and what does the exercise do, what are we specifying?

Sewell: these are in use.

Sebor: arbitrary subsets must exclude some practice.

Sewell: the choice is over the strength of the guarantee.

Sebor: I have trouble with what this poll achieves.

Sewell: I am trying to align the Standard with practice and some sets strengthen its provisions.

Sebor: it is possible that *some* subset of this will emerge over time – GCC and Clang just added F1 and F2 with very surprising results. Over time implementations will converge as they gather data.

Sewell: all of these are legitimate options and it is useful for the user to be told which the compiler is giving them.

Myers: we really need a diagram showing the relationships and which subsumes which. The order shows the order we came up with them and not the subsuming order.

Sewell: I didn't want to change the numbering scheme. Specifying other things gives the guarantee more strength.

> a: an allocation-time nondeterministic choice of a concrete value (stable if re-read)
>
> - c.1: plain UB
>
> - c.2: at the implementation's per-instance choice: diagnosed compile-time or run-time error or (a)
>
> Note that an implementation could trivially conform by picking (c.1) in all cases. That's weaker than the current standard in some cases - we could conceivably mandate a non-UB lower bound in those cases.
>
> These options don't include wobbly values (as there was little support in the preference vote), so that might have to be stronger than the current standard.
>
> - f.1: automatic zero initialisation (with some per-variable opt-out as in existing practice), or
>
> - f.2: automatic sentinel-value initialisation

Sebor: so the first poll includes C1, “plain UB”, plus A and C2 – what are we asking here? To document “UB or A or C2”? What does that do?

Sewell: they would have to document which of them they provide. So an implementation can say “UB” if it can't say anything better.

Seacord: I would like the solution to be teachable. “Every implementation initializes every different type of object differently” is not teachable.

Wiedijk: why are A and C2 options?

Sewell: some would like C2 over A, there are program benefits.

Gustedt: A and C2 were majority favoured.

Uecker: phrasing “in terms of these” – is it constrained to *only* these?

Sewell: yes, it must use one of these.

Uecker: what if I think “address taken” is good for one set but not for another?

Sewell: the compiler documents the circumstances, including addressing. Not all of these are generic enough for the current Standard. A compiler can say it does different things for address-taken vs address-not-taken.

Uecker: I would like to vote on these differently.

Sewell: there are too many choices there to achieve consensus.

Wiedijk: “implementation defined if UB” gets my favourite. What’s the difference between a runtime error and UB?

Sewell: a runtime error is intended to mean a trap, signal, or something else a user will see. Whether there is output is out of scope.

Gustedt: *bounded* UB, not unbounded UB.

Wiedijk: is that concept new to the Standard?

Gustedt: arithmetic overflow is similar, it has rules about traps vs. signals.

Sebor: there’s a lot of discussion of the options themselves but the goal is to force the implementations to document them. Not all implementations do document implementation-defined behaviour, including major ones, so this may be useless.

Sewell: they are choosing not to conform to the Standard.

Sebor: GCC and MSVC outline their choices. Clang and Intel do not.

Ballman: Clang doesn’t have this because nobody has written it yet – “read the source” etc.

Meneide: if I wanted F1 and F2, would C2 cover that with a runtime diagnostic? Are they a subset of A?

Sewell: they are specializations of A.

Bhakta: relying on Godbolt as a survey is not a great way forward – lots of compilers, including six from IBM, are not on there. *All* of mine document their implementation-defined behaviour.

Sewell: so C1 has the strongest support. Not surprising as it is the status quo.

Wiedijk: can we poll A and C1 together?

Straw poll: Does WG14 accept (a, c1) as the only allowable set of options?

Results: 13-9-2

Sewell: I have no idea what to do next. There is genuinely no clear direction here for *any* proposal.

Voutilainen: spend your energy elsewhere.

5.6 Uecker, Variably-Modified Types [N 2778]

In the previous meeting we discussed arrays, which are underdeveloped in C. Feedback was to break out small proposals from that paper. Right now, VLAs are optional, but they are useful in many ways, including bounds-checking, without overhead.

Their use in signatures supports the Charter and it is strange for the Standard to rely on an optional feature.

VLAs themselves have problems especially w.r.t stack usage, but we can break up the feature, keeping the arrays optional and making their types mandatory. This keeps the useful parts with no downsides, avoiding all of the stack issues.

Sebor: I am in favour, but more in favour of making the whole feature mandatory. Otherwise this sneaks in under the `__STDC_NO_VLA__` macro – implementors have the ability to opt out of the cost of implementation; subsetting the feature under a single macro seems wrong. I would like to check support for mandating the full feature, or adding another macro.

Uecker: I also like making the whole thing mandatory, but the implementation cost of this is much lower than full VLA support, felt more reasonable.

Ballman: I support this and like the direction. Clang has an ongoing RFC for a diagnostic users wanted, to tell them that VLAs are in use – users *want* this feature split because the distinction is too coarse right now. Unsure about the wording of automatic duration implying parameters though (6.2.4p5).

Uecker: parameters are never arrays, but that could be improved.

Bhakta: to echo Sebor, if this is just about types, it sounds valuable, but hidden by a change to existing macros is wrong. I would prefer a second mechanism or to make it fully mandatory.

Uecker: mandatory implies no macro. What is the alternative?

Bhakta: so I want to keep the old macro *and* add a new macro that only describes object support. `__STDC_VERSION__` is a mandatory positive macro.

Uecker: old code can rely on one, new code can check both.

Gustedt: the wording needs smithing – the declaration is a VLA but not the object. I am not in favour of the extra macro but like the direction.

Myers: why were VLAs made optional? (ref: n1460, n1493, n1542) The discussion in 2010 is unclear. Was it about the complexity of the feature or defining the objects? Types seem more complex, and may influence the answer.

Uecker: I think it was not widely provided at all. I disagree about complexity, types are not trivial but stack-handling is the more complex part.

Gilding: I like the direction and definitely want the type system part to be mandatory. I also think this is clearer than the status quo w.r.t user expectation, which is that object allocation might be optional but types exist in the compiler.

Meneide: older code will choose not to use the VLA – now users would need to check for types separately. In practice users are not good at checking feature macros correctly. We should add a macro for better user experience.

Bachmann: I like this a lot; I would prefer the full feature to be mandatory, the user can’t do much without the macros anyway, but if you need them you need them. This gives most of the feature back. We can make the arrays themselves mandatory too later.

Uecker: maybe vote on making the whole thing non-optional?

Krause: I oppose Meneide here – users already have the C version macro, and checking the version is enough for this. Don't need another one.

Tydeman: need to update the 6.7.6.2 examples of valid and invalid VLAs. Not wrong as-is, but assume full support. We would want additional examples for conditional support.

Sebor: good arguments have been expressed both ways for a new macro. There seems to be little opposition to mandating the types. What data do we have about implementations that don't provide them?

Tydeman: Microsoft don't and never will. People use warning flags, but those will continue to exist anyway.

Banham: in n2660 you have a section on flexible array members?

Uecker: didn't have time to extract this one yet.

Banham: flexible array members are used extensively in system programming. The "fortify" approach is also increasingly used, would like to standardize it.

Straw poll: (opinion) Does WG14 want to make VLAs fully mandatory in C23?

Result: 7-9-8

Ballman: we haven't had an answer to Myers's question, want to know why these were made optional.

Keaton: some people felt C99 was too ambitious and features should be made optional. A collection of "complex" features was created and this was included in it.

Meneide: n1542 "Implementability concerns", didn't know whether it was possible in embedded. Many hadn't implemented it yet at all so it was unknown. I do not think this is actually impossible on any platform. I don't think the types were the problem.

Gilding: do we know of any targets where this would be impossible? Not including Microsoft.

Sebor: the heap strategy is not viable in embedded. GCC still supports it for all targets.

Myers: GPUs and PTX maybe, no other problems. BPF might be another weird architecture like GPUs with limitations on what can be supported, but it isn't really expected to fully support all of C.

Bhakta: our freestanding implementation has a user-stack – so users know the requirement for the entire program at compile-time. VLAs are rarely used because they would alter this. Often there is no heap either. Even then, they're still supported.

Yodaiken: are they problematic on a GPU? Is it reasonable to exclude MSVC like this?

Uecker: they said it on the internet, which is not binding. Their stated concern is still stack safety and not the types.

Ballman: there are no Microsoft reps here. Microsoft does also support freestanding for device drivers and esoteric hardware.

Krause: Microsoft said they don't consider C important enough to do the work.

Keaton: Microsoft were explicitly invited to join WG14 and said they didn't have the resources.

Uecker: I would expect any technical issues to come from the embedded side.

Ballman: they will notice or they won't.

Keaton: we cannot let them block us after they were invited to join the group.

Krause: types are implementable, that's just work. Objects could go on the heap, if there is one; some targets do have a reasonable stack, but others don't and by default we don't even stack-allocate automatic variables (functions are non-re-entrant by default). MCS51 is a hugely popular. The types are OK, but there is not user demand to add the objects themselves.

Straw poll: (decision) Does WG14 want to make variably-modified types mandatory in C23 as specified in n2778?

Result: 10-5-9

Ballman: "in C23" is why I voted no.

Gustedt: yes, this would be rushed through.

Bhakta: what is the base of consensus? Yes greater than Abstain plus No?

Keaton: not prescribed. This has strong support between the Yes/No sides. There is significant abstension.

Ballman: would have voted yes for "along the lines", because there are known wording issues.

Myers: I would go from No to Abstain, also concerned by wording issues.

Keaton: let's put it in for now.

5.7 Uecker, Remove UB for incomplete types of function parameters [N 2770]

This is just wording cleanup, after removing K&R functions, some text needs fixes. The text about complete objects appears twice, this is partly editorial. The text still refers to definitions without a prototype which no longer makes sense.

Gustedt: I support this completely.

Bhakta: in favour, but want to point out that the inconsistencies show the original change was not complete enough. Can't just like a change, it must be correct too.

Straw poll: (decision) Does WG14 want to adopt n2770 as-is into C23?

Result 20-0-1

5.8 Uecker, C23 Atomics, Issues and Proposed Solutions [N 2771]

Atomics are good but can be annoying – they don't just need improvement, they are *unusable* for some portable code without relying on non-portable things. This may affect the ABI and implementations with no atomics yet.

The good: aiming for C++ compatibility is helpful.

The bad: lack of atomic accesses to existing data structures because an existing pointer cannot be converted to be `atomic`-qualified. To access an atomic-element matrix requires copying it. If it worked like a qualifier, it would “just work”. Torvalds in particular was opposed to this.

As a qualifier, it isn’t one. It is the same as the specifier. The lack of control over placement makes some sense in C++ but not what users expect of C. e.g. embedded locks inside arrays can blow out the size. Because implementations didn’t choose common representations there is an ABI issue. Generic functions rely on compiler magic because there is no `atomic void *`.

We do not expect C++ to change.

Alternative 1 is to separate out a new qualifier from the specifier to work like `atomic_ref`.

Alternative 2 is to no longer require the specifier and the qualifier to produce the same type, which is potentially confusing.

Only in some cases do the representations differ right now. The alignment issue is already free to be altered by the compiler. We may require runtime alignment checks for pointers, or always fall back.

Myers: reusing the qualifier and making the keyword mean two things is a bad idea. There is maybe a use case for atomic access to non-atomic objects; this should follow C++ as closely as possible. The ABI question is as issue, it was suggested on the reflector that we should track ABI choices and who is making which decision. This sparked little response and there doesn’t seem to be interest in tracking. Making atomics always the same wouldn’t get rid of all ABI issues, maybe in practice but it is not a complete definition.

Uecker: w.r.t C++, `atomic_ref` is similar to a qualifier, but the alignment requirement makes it non-portable to all existing datastructures. We could always fall back to a library call though.

Gilding: please don’t reuse the keyword to mean something else. I do agree about qualifiers and user expectations regarding them. We also have the lock-free macro tests to inform the user which types could be used this way.

Uecker: macros do not fix the portability problem.

Krause: it is a definite usability improvement for a qualifier to work as a qualifier. I would rather rename the specifier. Lock-free is mostly important for signal handlers; we can’t implement anything efficient in a lock-free way, but do use the multi-lock pattern that works for signal handlers. On our architecture locks are efficient and *faster*, contrary to user misconceptions.

Uecker: it’s at worst one branch to check that we can access a pointer without a call.

Krause: The qualifier should have the good name because the specifier is implied on definition.

Bhakta: I like alternative 1. C adoption is *slow* – there are few users even of C11 yet, and it helps with slow adopters to have new syntax that doesn’t risk breaking existing code.

Sebor: several issues are outlined but not all are serious. Long previous discussions converged on solutions that didn’t work because of lack of implementation experience (struct member assignment, initialization, etc.) – we will not adopt without implementation experience of the semantics. It’s really important that features have experience behind them.

Ballman: the comments saying the specifier and qualifier are identical are wrong. Specifiers create a type, the qualifier does not. But these tripped people up, so they should be unified.

Uecker: no, alignment is the same; they are the same.

Ballman: the amount of confusion I have seen online shows the issue exists.

Krause: the lack of existing code is good, it gives us a chance to fix it. With Alternative 2, existing C code will still work correctly, unless the specifier really gives different types. I don't think Alternative 1 serves existing code.

Voutilainen: what is the problem with a required alignment for `atomic_ref`?

Uecker: for some types, a stricter alignment is required than for the underlying type, which means qualification can fail for some pointers. Would rather use a runtime check.

Voutilainen: and a lock?

Uecker: we do that for other types.

Straw poll: (opinion) Does WG14 want something along the lines of n2771 in C23?

Result: 12-6-4

Wednesday

5.9 Thomas, C23 proposal - 5.2.4.2.2 cleanup (N2672 update) [N 2806]

(Consider only the change to 5.2.4.2.2p4. The rest is left over from N 2672 which was already voted into C23.)

Bhakta: n2672 was already voted in, so this is the same text with a one-line change based on feedback from outside the working group. This is: Strike out #4: ... and all possible *k* digits and *e* exponents result in values representable in the type) ...

K is an index; it has no other meaning.

Further feedback from Myers: p4 defines FP classifications, which break up into paragraphs for normal, unnormal, etc.; separate paragraphs for each. This is an editorial change with no semantic impact.

Straw poll: Does WG14 want to adopt the changes to 5.2.4.2.2 p4 added by n2806 into C23?

Result: 14-0-3

Myers: it's also unclear what it means, future text for the definition of "normalized" is needed.

Bhakta: future papers go into depth on this; not perfect text right now. There will be further changes but they are not in scope. This keeps the model description as small as we can. We do have another paper that's editorial but also large – we're not sure it's effect-less without review, and want wider eyes on the non-trivial change.

Keaton: in this case the paragraph is complex enough that a paper is a good idea.

5.10 Thomas, C23 proposal - overflow and underflow definitions (N2746 update) [N 2805]

This is a seemingly simple topic at the surface level but actually more complex. There have been numerous revisions and RFCs already on this topic, requests to change the model to fit formats (accommodate `double double`); this doesn't cover every model existing in the wild.

Looking for acceptance as-is, plus changes later; or to be deferred until the final definition is in the body: value is too big for the type you have; going below precision is underflow. But wording is hard.

w.r.t "mathematical result" not fitting the evaluation model, C doesn't require correct rounding in general; implementation can overflow a result that is mathematically in range. Doesn't fit numbers that get normalized back into range by rounding. We remove "mathematical result" so that infinite precision of the implementation is not implied, similar to IEEE "ordinary accuracy".

Possible concerns include `double double` where implementations are known to differ. Where models don't fit these changes and concerns exist, we will bring text later. We prefer to incrementally improve the feature.

Gustedt: hard to understand the new term "ordinary accuracy".

Bhakta: not an original term, comes from the IEEE standard.

Gustedt: needs a definition, not in a footnote.

Tydeman: it means "what the hardware does".

Bhakta: it's not intended to be a new term of art, but it does look like one.

Uecker: similarly, fine with incremental approach, but does the C Standard guarantee anything about rounding? Good or bad?

Tydeman: all floating point operations are implementation defined for accuracy.

Uecker: so the implementation has to document what it does; so this should refer back to that clause.

Bhakta: C does specify modes, certain functions specify and deal with them; but in the general case, as Tydeman said.

Myers: "The implementation may state that the accuracy is unknown." (5.2.4.2.2p8). "Ordinary accuracy" tries to cover a couple of things – implementation result over mathematical result, and also rounding after going out of range – trying to be more general than IEEE.

w.r.t `double double`, has a different spec, lo+hi rounds to high; discontiguous bits do not lose accuracy. Other variants do have lower precision than (all `double + double`); may need supernormal values rather than huge values. POSIX has variable-precision format; may need supernormal if anybody is using Unum (https://en.wikipedia.org/wiki/Unum_(number_format)#Unum_III).

Bhakta: we plan to put all of this in part 5, discussing what we had before.

Gustedt: really uncomfortable with the wording – we must have a clear definition for the accuracy first.

Bhakta: so for accuracy, do you want to change p8 to give a better specification for implementation accuracy?

Gustedt: if the term is needed it must be defined. If the implementation is allowed to not know, this becomes meaningless.

Bhakta: for many FP formats there isn't a single accuracy – e.g. there is more accuracy between 0 and 1 than there is outside that range.

Gustedt: they still say what happens near 0 and at the edges – we must elaborate.

Bhakta: that's a bigger change than we want here.

Ballman: we have two other "overflow" papers – an integer one on Friday, I'd like reification of the overflow concept so we don't have different definitions. We might want to see if there's a preference for a unified approach.

Bhakta: this is an apples-to-dogs comparison – these are really very different things; integers are very simple and a common definition would be worse. They don't even have underflow. This would make floating point even harder to understand.

Ballman: primarily – which users? Some users think under-wrapping *is* underflow, so we want to avoid confusion. If it's different we should call that out.

Bhakta: I don't agree, floating point looks at magnitude. The CFP is not best to do this as we have a very different worldview.

Ballman: content with this.

Keaton: historically, the Standard placed no requirement on accuracy. Users should know but we never had it outside Annex F. This paper does say the minimum it needs to, and adding accuracy is a wider-reaching change.

ISO 24747 - special math functions - does have wording for specific functions making their values implementation-defined for argument ranges where they are hard to compute.

Bhakta: "ordinary accuracy" is used as it exists.

Myers: the question of accuracy of an operation depends on the operation, defining it for arithmetic doesn't mean we can define it for e.g. `cpow`.

Bhakta: that's why we mentioned cosine – it's very implementation-dependent, it would triple the size of the Standard.

Uecker: accuracy is a huge topic that we shouldn't tackle now. Can we have an editorial change to the footnote to point at Myers's reference?

Bhakta: great idea, and it is editorial, I agree we can change the footnote. With the understanding that we will examine other models, `double double` is already not handled and this is not worse.

Straw poll: Does WG14 wish to adopt n2805 as-is into C23?

Result: 10-1-5

Bhakta: we assume the CFP can bring more editorial papers?

Keaton: yes, on-topic.

5.11 Thomas, C23 proposal - Annex F overflow and underflow [N 2747]

The base C Standard tries to fit every model. Annex F fits IEEE specifically. The current definition only fits the 1985 spec, and must be updated. There are consistency issues with the rest of the document and should put the same kind of wording into the Annex F definitions.

CFP considers *all of this* editorial. Annex F incorporates IEEE as a normative reference, with precedence, so this can't change the semantics, but is easier to read for users.

Krause: so, sentence 9 – why can't we use the "underflow" wording here that we put in a moment ago?

Bhakta: this is what's in IEEE – "tiny" has a mechanism and definition that doesn't exist in the general C model which fits other models. This is a refinement of the model, not a diff from it. Floating-point exceptions are also a IEEE thing.

Straw poll: Does WG14 want to adopt n2747 as-is into C23?

Result: 12-0-5

Bhakta: this is all editorial but non-trivial, hope the Committee agrees this warrants input.

Keaton: this is complicated and glad you solicited it.

5.12 Thomas, C23 proposal - Normal and subnormal classification [N 2842]

"Normal" is used as an English word, but it should be a term of art. It *can* be read as "normalized", but can also fit other things. Could be confused, so we should give a definition. Also this helps to classify things that are outside normal, like subnormals – adding "normal" allows the definition of "subnormal".

There is no semantic change to the model, but *may* result in implementations needing to change their value classifications. We do make the model better defined; there are remaining problems, especially with `double double`, like before, but changes for that are to follow.

We will expand this text in future versions. This reflects the p4 classes split.

Myers: does this agree with existing implementations for decimal floating types?

Bhakta: I have not heard otherwise. We have not surveyed exhaustively.

Tydeman: I tested this and do not recall any problems with any existing implementations.

Bhakta: we are confident but not certain.

Gustedt:I understand this paper much better – could the overflow/underflow paper be expressed in such terms?

Tydeman: there's the issue with rounding.

Bhakta: rounding is a complicating factor; also, overflow and underflow have special cases beyond these; exact/inexact exceptions make a simple description difficult. Did debate on the balance point between understandability and completeness. Any similar definitions seemed to cause implementation divergence.

Gilding: like the incremental approach, it’s very comprehensible and still an improvement.

Straw poll: Does WG14 want to adopt n2842 as-is into C23?

Result: 17-0-2

5.13 Thomas, C23 proposal - Clarification for max exponent macros [N 2843]

Based on feedback from out of the group: macros don’t fit everything if they’re larger, not normalized. We considered splitting the macro for normal and non-normal; Standard sets E*max* as a value, which wouldn’t work in the model.

We explicitly call out that macros are only for normalized exponent. *Again*, this doesn’t work with all models, especially `double double`. Will again need more work for the remaining concerns: one concern that we don’t have wording for, CFP will look at but we have no idea how big it will be.

Straw poll: Does WG14 want to adopt n2843 as-is into C23?

Result: 16-0-2

Bhakta: maximum integer such that `FLT_RADIX` raised to one less than that power is a representable finite floating-point number. If that representable finite floating-point number is normalized, `emax`.

Keaton: future papers should add a note that they refine these topics and are not new content.

5.14 Tydeman, `remquo()` [N 2790]

Tydeman: this addresses an issue from the Austin Group of three unidentified cases returning `NaN`, and generalizes it.

Straw poll: Does WG14 want to adopt n2790 as-is into C23?

Result: 15-0-5

Bhakta: good – did we ask Stoughton or the Austin Group about this?

Stoughton: yes, we’re happy with this.

5.43 Tydeman, `*_HAS_SUBNORM==0` implies what? [N 2797]

Tydeman: originates from external question about ARM architecture – what should flushing subnormal do? We decided to remove the macros that describe the support for subnormals and make it implementation-defined in new text. On ARM, the FPU can change this at runtime, so these are not build-time constants.

Myers: I am wary of removing macros added in C11 without going through Obsolescent. They don’t cover all cases, but they’re not useless – are people using them? What does the C++ liaison think?

Tydeman: we removed `gets` without warning. These can only be constant sometimes.

Gustedt: removing macros which are used invalidates code. This needs investigation

Bhakta: similar to Myers, this isn't unuseful in general. It makes sense to keep them for implementations where the value doesn't change. Agree that it is not adequate as-is; this value is useless, but it doesn't mean all are. If we could tighten the text to only refer to operations that are constant, and add a runtime function to query the others?

Gilding: the existence of the question seems to imply use in the wild?

Tydeman: no, this came from an implementor.

Ballman: this *is* a liaison issue as C++ defines these. Need to coordinate. Do we have sentiment to deprecate?

Tydeman: so we need C++ to move first?

Bhakta: we need an action item, not just to leave it to C++. Need an opinion on deprecation vs. further specification vs. removal; deprecation alone is not good enough.

ACTION: Tydeman to take n2797 to the liaison SG.

Ballman: will schedule it – is this for C23? Only have two meetings.

Tydeman: this is easy to determine at runtime and removal is not painful.

Myers: `issubnormal ((float)(FLT_MIN/2))` might be how someone could determine at runtime ... except that runtime flushing to zero might not always be respected by constant folding etc.

Gustedt: problem that we want an Action internally and also for the SG.

Bhakta: I disagree that this is knowable at runtime – it depends on the operation. If we keep it runtime it needs to be more fine-grained for zeroes and operations should have replacement functions. It could be possible to repurpose the defines to functions.

Keaton: so is this needed in C23?

Tydeman: it would be nice but is not necessary.

Keaton: we'll try.

Bhakta: the issue is really answering the user, but they have a solution and the main driver is resolved.

5.44 Bhakta, Proposal to update CFP freestanding requirements [N 2823]

Changes applied to C++ to make freestanding more comprehensive; are concerned by the C-side expansion which is incorporated into C++ by normative reference. C23 integration of the TS added stuff that WG14 thought was needed but WG21 did not agree. Trying to reduce the scope to something WG14 and WG21 can agree on.

This is a CFP perspective and does not represent IBM's or Bhakta's opinions.

CFP has examined IEC 60559 and required parts for conformance. `errno` for instance pulls thread-local storage into freestanding; could remove the thread-local requirement. Could remove the exception model from the `matherror` content. Doing the first leaves in the possibility of thread-local storage, but minimally forces conformance.

Alternative 1 removes global changes to state, but breaks conformance to IEC 60559. Some things are left in – query functions do not set anything, could also be added. “Without the requirement to set `errno`” is an operative, global change. Put here and not in 7.5 in order to keep freestanding together.

Alternative 2 uses the `math_errhandling` route – use the floating point exception mechanism instead of `errno`, additional permission to reduce the thread-local requirement.

Aim is to make it easier for implementations to conform.

Krause: this only applies to freestanding implementations claiming IEEE conformance? So C23-style implementations can still be freestanding; if there are no threads no change is needed. Removing something makes some implementations non-IEEE conforming? So now there are four sets of conformance. Would like Bhakta’s personal opinion before a vote.

Bhakta: Individually I do not want freestanding to require IEEE conformance. Correct that changes only apply when claiming conformance. Global `errno` is a separate problem for freestanding implementations, which usually want *less* state. Trying to balance the needs of IEEE and freestanding, may be impossible.

Bachmann: if we support IEEE754, subnormals are not optional, conversions are a lot of code – one integer state is so little, this is totally unbalanced in the face of the scale of conformance. N2095 says that if we support it, decimal float is not optional, adding even bigger weight to implementations.

Bhakta: the paper is trying to reduce the scope of what C23 brings in, not to remove conformance. CFP would oppose removing the requirement for freestanding to support IEEE.

Tydeman: the changes to `strtod` are independent, should be aligned with the rest of the family.

Bhakta: that’s a glaring omission… fine with piecewise approach.

Banham: does freestanding imply single-threaded?

Bhakta: not in the Standard. Maybe in practice.

Banham: smaller processes use an RTOS for threading, but not `thread.h`.

Bhakta: it’s a general idea – the environment doesn’t have to be thread-specific.

Banham: `errno` can only be thread-specific if threads are in the language. What about the exception-handling mode?

Bhakta: Alternative 1 says: the floating point environment doesn’t change with freestanding application of these; alternative 2 says it does.

Gustedt: difficulties with this paper… it lacks rationale and context; the changes are not a diff, so it’s hard to say which changes are OK; there are changes to functions that are independent of freestanding; not a mature change.

Bhakta: we can add wording; independently of `strtod` this fits into the idea of allowing `errno` to not be present. Separability is nice but doesn’t fit the alternatives. Are diffmarks required? We felt the changes were more confusing as a diff.

Gustedt: not required but a readability requirement. Including the original text can improve readability.

Myers: w.r.t Alternative 1, the text says conversion functions, then lists – excludes `strfrom`, `atof`; would expect `strfrom` at least as a number conversion function.

Bhakta: `strfrom` was not added because it is locale-sensitive, needs a separator etc. This avoids pulling localization into freestanding, which currently only needs the C locale.

Banham: Alternative 1 goes too far by removing the floating point exception modes – `errno` is fair enough but exceptions are important and should be an option.

Bhakta: freestanding only sets a minimum requirement.

Banham: the market will settle on a minimum but we should let it decide. Maybe Alternative 2 is better for this.

Bhakta: some users want every function; a portable subset is good though. Alternative 2 requires state, which may not be an option. The choice is between freestanding not claiming support, and guaranteeing support.

Banham: so that means two levels of support within freestanding.

Bhakta: anyone who wants to fully support IEC 60559 will be hosted, given the scale of what is pulled in.

Banham: not convinced, many DSPs are conforming now.

Bhakta: in my experience DSPs and GPUs are only partially conforming.

Straw poll: Does WG14 want to adopt Alternative 1 from n2823 into C23?

Result: 5-3-10 (no consensus)

Tydeman: I want to poll `strtod` separately if needed.

Keaton: this is not sufficient consensus.

Straw poll: Does WG14 want to adopt Alternative 2 from n2823 into C23?

Result: 4-4-10 (no consensus)

Straw poll: Does WG14 want to adopt the changes in 7.22.1.6 p7 of n2832 into C23?

Result: 9-1-9

Myers: this proposes a change *only* for decimal floating point. Will we propose the TS 18063 binary annex?

Bhakta: Yes, decimal is all that was added, the others would be a bigger change. We could change the others the same way.

Keaton: any abstainers who were not “don’t care”?

Krause: I am comfortable with the change to decimal floats, but not the others.

Bhakta: we will provide papers for the binary and extended.

Gustedt: it would be better in the other paper: I oppose inclusion without a rationale, this is not traceable and is bad for users.

Myers: papers should have an analog to `matherrhandling = 0`, to make consistent with `math.h` functions.

Ballman: is this for any C Standard or for C23? C++23 cares about this.

Straw poll: Does WG14 want to limit the scope of freestanding C23 implementations w.r.t IEC 60559 conformance?

Result: 9-2-7 (sentiment)

Bhakta: would like to help feedback to the C++ side, I want an idea to proceed that gains acceptance.

Ballman: I abstained because of lack of rationale, readability or preparedness.

Keaton: typically we pull papers from later on – hoped to increase the chance that people understand it.

Myers: floating point papers are fast, maybe we should schedule more.

Ballman: CFP papers being so small, maybe omnibus papers would help the scheduling.

Keaton: the Committee always finds small pieces easy to approve – separable concepts are easy to consider. It throws off the schedule, but in a good way.

Ballman: WG21 does omnibus issue processing; it does wording review but for floating point we trust the CFP.

Gilding: "trust the CFP" does inform my votes.

Bhakta: we don't want to restrict the voices of others. Slightly different meeting plan this time with two sections, this would normally be in section 7.

Gustedt: I like the idea of a fast-track. WG21 has a wording study group, maybe we should too.

5.45 Thomas, C23 proposal - `feraiseexcept` update [N 2845]

Annex F has a description for the function; this is out of date since 1985. There are no "coincident exceptions" any more. IEC 60559 removed because of overflow/underflow, as there was no way to tell the ordering of the exception in C programmatically; now possible to know which came first and this detectable difference can change program behaviour.

Additionally "atomic operations" in Annex F is unrelated to the C11 term. Changes the wording to give ordering and remove "atomic".

This is written as editorial but might not be, gives binding and removes confusion.

Straw poll: Does WG14 want to adopt n2845 as-is into C23?

Result: 17-0-1

5.46 Thomas, C23 proposal - Clarification about expression transformations [N 2846]

This was brought from outside the working group; Annex F talks about transformations, but extensions added invalidate them; footnote is wrong. This is not normative.

C didn't bring in alternative exception handling and it shouldn't be invalidated.

Straw poll: Does WG14 want to adopt n2846 as-is into C23?

Result: 18-0-2

Gustedt: non-required features are called extensions?

Bhakta: is this editorial?

Tydeman: signalling `NaN`s are not an extension.

Bhakta: alternative exception handling is outside of that.

Tydeman: changing the text changes that.

Gustedt: word as "extensions or"? The text should be explanatory and easy to understand.

5.47 Thomas, C23 proposal - Contradiction about `INFINITY` macro [N 2848]

Also originates outside of the working group; the text of the expansion is problematic. If a type doesn't have an Infinity it cannot be implemented.

The evaluation format must be larger than `double` in order to overflow, so can't implement the "else" part of the description. This is only useful in implementation-specific ways: `INFINITY` can't be used as a feature test, unlike `NaN`, which should be possible.

Implementations could already be making sense of this another way and this may change the two options presented.

Myers: this only proposes a change to `float.h`, not `math.h` – is this deliberate? `math.h` is older and might have more impact.

Bhakta: thought there was only one definition… we don't want different definitions and did not intend to split it. Hadn't checked the `math.h` text.

Tydeman: we thought it was moved, not copied.

Myers: moved signalling-`NaN` because it was new; we copied quiet-`NaN`.

Tydeman: we should change both.

Bhakta: maybe if it's obsolescent we should leave it.

Banham: is the intended meaning of the struck text a literal with the `f` suffix?

Bhakta: that is one option that could satisfy "translation time"; it doesn't always work – usually compiler magic is needed to avoid issues with exceptions, but it could be.

Banham: is the intention to have an object? Or would an overflow-object break translation?

Bhakta: the macro can be used in a non-static context and the user may want an exception there. There is no way to tell the difference.

Banham: unclear what the objective is – what is required if infinity is not implemented? Should some equivalent be provided or should the compiler error?

Bhakta: the fix is to make `INFINITY` a feature test: if it's there, it will work.

Gustedt: I am in favour of something along these lines, but this is very surprising – this is a normative change that can invalidate existing programs.

Bhakta: the alternative is provided for this reason – gives likely existing behaviour validity. We have made other normative changes.

Gustedt: this invalidates application code rather than implementations.

Bhakta: application code that depended on `INFINITY` without infinity was specified wrong.

Krause: the suggested change is fine, but maybe people used `INFINITY` instead of huge values? But they should have used huge values. A feature test is better than Alternative 2.

Myers: application code in previous versions was using `math.h`. We could make the changes separately to each header?

Gustedt: please do not.

Krause: second Gustedt that we should make the change everywhere.

Bhakta: this is my (not CFP) preference, and to keep it the same (CFP didn't know about that). We could ask CFP, or we could seek consensus to change one or both.

Straw poll: Does WG14 want to adopt the suggested change in n2848 as-is into C23?

Result: 19-1-0

Bhakta: can I ask about the No vote?

Stoughton: it's possible for this to break existing implementations.

Ballman: what about `math.h`?

Straw poll: Does WG14 want to ensure the definition of INFINITY in maht.h matches the definition in float.h?

Result: 18-0-0

Bhakta: CFP will look into whether a paper is needed for this.

Keaton: this resolves all new floating point issues.

Thursday

5.15 Múgica, Memory layout of union members [N 2788]

Ojeda: (standing in for Múgica) this can still go into C23 even if it is revised?

Keaton: yes.

Ojeda: in unions we do not say that members overlap; we do not fix their starting offset to 0. Quotes were pulled from 6.2.5 and 6.7.2.1p16 that do force the offset to be 0; Múgica feels that this doesn’t force them to be zero, only to be the same, and that padding is not considered.

Gustedt: so Ballman’s point was that there is a sequence of deductions in the Standard that force this, but it cites four different places and is not easy to see. Clarification is therefore useful and the wording from the reflector is good.

Uecker: so in the part about 6.5.8 “Relational operators” discusses conversions (but doesn’t specify unions by themselves) … didn’t find 6.7.2.1 in the paper.

Bagnara: what is the purpose of “in the same order” in the new text?

Ojeda: that they are not rearranged?

Gustedt: the formulation was originally for both unions and structures; in structures the first member has a special role.

Ballman: thank you Ojeda for presenting this. I agree with Gustedt that the spec is all over the place but we can unify that in a footnote, not normative text. This minimizes the chance of confusion between two places. I didn’t understand the comment about a union storing a pointer to itself – to me the text says there cannot be leading padding because the pointer is equal.

Ojeda: I agree that “a pointer suitably converted” implies offset zero, but Múgica seems to think the conversion allows an offset.

Wiedijk: I find it hard to have an opinion – the relationship between union members is subtle and an issue for the Memory Object Model SG. Taking pointers will cause effective-type issues, compilers may ignore the relationship.

A union *can* be bigger than its members, and have padding, and it doesn’t say that those bytes change value. My understanding is that other members stop existing after a store, how can they have a value? An over-long tail becomes uninitialized, etc. I support the idea of a clarifying footnote.

Gustedt: it’s *only* C++ where only one member is active at a time. In C union punning is explicitly allowed, and a write to one member is a write to all members.

Bhakta: I agree with Ballman that normative text should not be in multiple places. Clarification should be non-normative. Has anyone got this wrong? The chain of deductions seems fine in practice.

Myers: w.r.t pointer conversion implying offset 0 – as in the reflector message, not much is specified about pointer conversion, and it might not preserve the value intact. What people assume about pointer conversion is not the same as the normative text.

Tydeman: 6.7.2.1 allows padding at the end of a structure or union. We should clarify that there is none at the front. When one member is stored the others take unspecified values.

Gustedt: no, they don’t.

Gilding: new users have great difficulty with this topic. Clarification is definitely needed. Would padding bytes be treated as a structure submember?

Ballman: there is no definition for a “structure object”.

Straw poll: (opinion) Does WG14 want to add a footnote along the lines of n2788 to clarify that union members start at offset 0 and overlap?

Result: 16-0-7

Straw poll: (opinion) Does WG14 want to change paragraph 17 in 6.7.2.1 to clarify that “structure object” includes or applies to unions, explicitly?

Result: 9-3-12

Keaton: so this is direction, but it will depend on the text.

Bhakta: support for just the footnote was stronger.

5.18 Krause, Allow 16-bit `ptrdiff_t` again [N 2808]

This partially reverts a C99 change. Even today, `ptrdiff_t` *is* 16-bit, it is just non-conforming.

The raised translation limit for object size to 65535 only applies to hosted. This concern was raised but ignored. We are not convinced that the object size limits make sense. Nobody restricts their memory use against the limits. Embedded systems often come close to full hosting anyway. We want to lower the object size limit.

Gilding: users *do* care about the size limits. There is broad divergence between hosted and freestanding here, as hosted users really don’t.

Ballman: this may affect C++, as it incorporates `stdint.h` by reference and doesn’t specify its own values for the widths etc.

Myers: widths are from C23 and won’t be in there yet.

Krause: the Standard doesn’t mandate that `ptrdiff_t` is big enough for universal pointer difference. It is commonly the same as the pointer size, though the wording allows for a 32-bit `ptrdiff_t` and a 64-bit size, which implicitly restricts the object size.

Bhakta: is the chain reaction with `size_t` considered, or is it allowed to just fall out of sync?

Krause: `size_t` should be unaffected – pretty weird because of the 17-bit requirement that nobody follows in reality.

Gustedt: the `WIDTH` macros are new to C23, but the `MAX` macros would change and this would affect C++.

Sebor: I favour the change, but is this the only conformance problem on such systems?

Krause: it’s the only *intentional* divergence. Non-conforming `double` is a thing, but not a priority.

Keaton: can we gauge sentiment and then go to the liaison SG?

Straw poll: (decision) Does WG14 want to adopt the first proposed change in n2808 into C23?

Result: 23-0-1

Bhakta: I would rather vote indirectly and revert the pending objections.

Keaton: great.

Wiedijk: can you get overflow in pointer difference?

Krause: that's not a new problem, it's already expressible with 17 or 64 bits. Realistically, no on 32K systems.

5.16 Krause, `unsigned long` and `unsigned long long` bit-fields [N 2774]

Currently these are implementation-defined "other types". But bit-fields with greater than 16 bits are useful and would be good for portability. We consider adding these two types specifically to be most useful, to allow 23-bit `unsigned` etc. These are very widely used and supported for this purpose. It is also clearer how to implement these than other types, especially the signed types. Does raise open questions for other types.

What is the type of a bit-field? The base type, or "bit-field of"? GCC and Clang diverge on the type. Divergence is also seen on what the type promotes to when used. But this is already a problem for `int` bit-fields.

Myers: w.r.t the type of the bit-field, we should make clear in the integer promotions that a bit-field of type `unsigned long` but narrower than `int` will promote to `int` – this happens explicitly in C++ and avoids compatibility issues.

Krause: I left that out because we have divergence.

Myers: would hope that Clang follows C++.

Krause: in Clang it has `unsigned long` type – it doesn't promote, and matches `unsigned long` in `_Generic`.

Gilding: what was the original rationale for restricting the types in C90? Couldn't find this in the document log.

Bhakta: I do think we need text for the promotion behaviour, or we introduce a hole.

Banham: what is the benefit? Can you not oversize an `unsigned int`? If so this isn't very useful.

Krause: I don't remember if this is allowed, or if it is, whether all bits are value bits. Compatibility issue maybe.

Gilding: Constraints, paragraph 4 in 6.7.2.1.

Bhakta: `long` by itself is problematic – customers find problems because of changes with the address model. `unsigned long long` doesn't have this problem. Portability issue.

Myers: Clang does apply the integer promotions to small bit-fields, I just tested it.

Gustedt: you can't oversize a bit-field because the type is unclear in the expression. Adding `unsigned long` and `unsigned long long` avoids this problem.

Seacord: there's a long history of problems here, MISRA disallows them outright. What about the bit-precise types?

Krause: on a constrained system the space packing actually matters, but bit-precise types expand to the full storage unit.

Bhakta: what about smaller types like `short`?

Krause: if you didn't like `long`, why? They're not really useful anyway. `unsigned long` and `unsigned long long` are widely used in practice.

Bhakta: the Perl source code uses `short` and `unsigned short` extensively.

Gustedt: w.r.t bit-precise types – these were proposed in Core and would be sensible. The bit-field ensures space compression and the type makes the conversions clear. We could say a bit-field converts to a precise integer type?

Bhakta: that changes existing types.

Gustedt: an alternative would be to allow exact-width types and use them directly.

Krause: we do ask this in the paper.

Pygott: MISRA doesn't ban bit-fields, that is Advisory and there are additional rules about their usage.

Tydeman: allowing bit-fields to have more bits than `unsigned int` means the integer promotion rules must change. 6.3.1.1p2 If an `int` can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an `int`; otherwise, it is converted to an `unsigned int`.

Myers: above that there is the list of applicable cases which explicitly lists the bit-field types. Wider types are not covered. We would need to add wording for wide types with narrow widths to be covered here. w.r.t assignment, C90 DR 120 explains this but there is no wording for the conversions in the Standard.

Gilding: specifically a bit-field of bit-precise type of the same width should be explicitly allowed.

Uecker: is there a use case for that? There is an option for any type but clarify that the bit-field type doesn't exist and always provides the underlying type.

Krause: the use case is that I know the bits supported, and can elide padding bits. It would be elegant to ship a bit-field type, but apparently there is not existing practice.

Uecker: I would want promotion, but to the underlying type.

Krause: that's not what happens right now – `unsigned int` promotes to `signed int`.

Gustedt: I believe this *is* in the Standard, that narrow `unsigned int` promotes to `int`.

Krause: bit-precise types should not promote, because they don't promote.

Gustedt: yes, but that would need to be explicit.

Bhakta: w.r.t "why not other types?", the Standard allows other types, but they're not portable.

Myers: w.r.t promotion to the declared type, currently it *definitely* doesn't do this for `unsigned int`. What about the type of comma, or assignment? Existing practice seems to retain the bit-field-ness. You *can* put it in `sizeof`, but there is divergence on the result, even for `unsigned int`.

Uecker: all of this says we should do something or else implementations will get messier. We should try to find a clear rule and possibly straighten out the divergence.

Bhakta: do we want the issues now to be handled before adding more types and possibly making them worse?

Uecker: before they create divergent practice too.

Gustedt: this is particularly important for bit-precise types – they are new ,with little practice, and we should give rules first.

Myers: do people want a paper on existing bit-field issues? Listing them without proposing any changes.

Bhakta: I would really like to see that before this paper.

Straw poll: Does WG14 want a proposal for bit-precise integer types along the lines of n2774?

Result: 16-2-4

Krause: why the No votes? You don't want these types?

Bachmann: first consider the Standard types and *only then* the bit-precise ones, so this would be the wrong direction.

Krause: there are fewer open questions about promotion.

Bhakta: I want to see the issues resolved first, we can't have higher widths until that is done.

Krause: bit-precise types do not promote, I would not expect them to here either.

Voutilainen: feels like this would be a rush to get into C23. I would want to get the promotions right, and then remove all restrictions, like in C++.

Meneide: if we wait… I know of users who want this *because* they don't promote. This will impact choices the group makes later, after experience settles we risk divergent practice emerging.

5.17 Krause, identifiers for use by users [N 2807]

The new Standard reserves more identifiers. Classes are reserved and WG14 doesn't care, taking away any identifier that looks like it might be right. Unlike C++ with namespaces, there is no protection from clashes. Maintenance is difficult for machine generated code, such prefixes would be useful to use confidently there.

Banham: why can't we have namespaces?

Ballman: they would be nice, but w.r.t this – if WG14 doesn't honour reservations, why do we believe this would stick? Namespaces are the approach we would want to see.

Ojeda: namespaces are useful for all manner of things, like safe/unsafe, or library partition.

Bhakta: what is the motivation? Was this brought by a customer? Did we clobber a name?

Krause: a lot of the new identifiers aren't there yet and won't be in SDCC. This is more about containing the naming scheme. Freestanding doesn't require them, so users may not even notice the clobber. w.r.t namespaces, a lot of users dislike C++ and this is something that's not there yet.

Bhakta: I often hear the argument about the floating-point naming “explosion”, but would really like to see a customer-oriented motivation beyond “that looks like a lot”. It’s a potential problem but not a real problem.

Gilding: reserving the `usr` namespace does seem completely harmless.

Ballman: is there any indication that users want this?

Krause: this is for a code generator, it can generate names in the space. For users it’s not nice, but tools would use it.

Banham: ambiguity in the meaning of “implementation”, in this case, the Library. There is a separation between a public API and visible-but-private symbols. Could use the `__lowercase` namespace for those. Does this extend to POSIX and other third-parties? Or is it reserved only for the consuming application?

Seacord: for code generators, discuss with the person using reserved names for their generator. Perhaps a separate namespace for machine-generated names, as well. I dislike the “anti-reserved” term.

Steenberg: if we do, adding `gen` is really useful. “This is machine generated” has documentation value by itself.

Krause: I went with `prog` to reflect what the user can be: a human or something else. Namespaces would be nice but we don’t have that in front of us.

Straw poll: (opinion) Would WG14 like to anti-reserve some classes of identifiers?

Result: 7-9-6

Krause: well then I hope we put namespaces in C26.

5.19 Gustedt, Types and sizes v1 [N 2838]

It came up previously whether objects could be bigger than `SIZE_MAX`. But we can have differences that don’t fit in the range, `size_t` is similar and could be explicit.

The second change comes from the community, concerns VLAs in `sizeof`. Overwhelmingly (92% polled) users understand that this is wrong, and evaluation is never necessary here. We should restrict evaluation to *only* cases where it is necessary, i.e. a type name, not an expression.

We add one UB. This type of code is rare and easily replaced.

This does change “weird” code that uses VLAs as objects in `sizeof`, but we don’t care much about it and think it should be cleaned up anyway.

Gilding: why UB instead of an error?

Gustedt: not a good anchor for one.

Myers: I dislike the second change, it introduces a new inconsistency and confusion as well as a UB. Currently `sizeof` only has one “kind” of operand and this makes the language less predictable. The evaluation rule for `typeof` is also less consistent and contradicts existing wording. “Nested sets of `typeof` declarators” wording is not necessary. Replacing a surprising behaviour with inconsistency is not helpful.

Gustedt: this adds compatibility with C++, which would never evaluate the `++` operators here. Unfortunate overlap with arrays receiving non-constant expressions, would fix that incompatibility.

Myers: adding compatibility with C++ should consider allowing `const` or `constexpr` variables in constant expressions.

Gustedt: `const`-qualified variables will never have the same rules as C++, even if we adopt `constexpr`.

Seacord: can we just make side effects here a constraint violation?

Gustedt: fine by me.

Bhakta: w.r.t C++ compatibility, this is meaningless, VLAs are unique to C. All compilers I tested get this right, it is only the users who are wrong. I agree with Myers’s point that it is a bad idea to silently change runtime effects in user code.

Gustedt: would the constraint violation help, making it non-compilable?

Myers: this would fix that concern.

Sebor: w.r.t the first change, no opposed but concerned, suggesting an object can be `SIZE_MAX` implies that objects can exceed this at the moment, which they cannot.

Gustedt: perhaps “fits into `size_t`” would be better.

Sebor: we do not want to imply that support for objects of sizes this large is required.

Banham: ambiguous use of “integer constant”, it should be an “integer constant expression”; and integer constant is a literal.

Gustedt: yes this is a mistake in the current text, will fix.

Myers: w.r.t constraint violations, it’s hard to define what counts as a side effect, especially for volatile reads. w.r.t dereferencing a cast pointer, in practice this may be coming from a macro expansion, users are not writing this themselves and it is not easy to rewrite.

Gustedt: agree for expression operands, cannot be ignored. For side effects inside type names I think it’s reasonable to restrict these to appearing in declarations.

Gilding: `volatile` array sizes don’t deserve to work anyway. This is almost certainly an oversight if it appears in user code.

Straw poll: (decision) Does WG14 want to adopt change 3.1 in n2838 as-is into C23?

Result: 15-2-4

Sebor: an allocation isn’t an object – it doesn’t have a type. Change 3.1 only constrains complete types, not allocations.

Gustedt: sizes are not defined for objects, only for types. It could in principle be bigger?

Seacord: skip the vote and bring the wording.

Keaton: this is allowed.

Gustedt: polling for sentiment for 3.2.

Straw poll: (opinion) Does WG14 want to adopt something along the lines of change 3.2 in n2838 into C23?

Result: 8-7-7

Gustedt: worth a try! Not worth voting on TS 6010, we must stay consistent with C23.

5.20 Gustedt, Require exact-width integer type interfaces [N 2821]

Only one sign representation is left so everything is simpler in C23 already. There is a requirement for *some* exact-width types if they exist, but not for all. I don't see any reason why we wouldn't do this.

This definitely doesn't cover bit-precise integer types, only the standard and extended integer types. Extended integer types might be bigger than `long long`; simply exempt the wider types from this.

Bhakta: I really appreciate the non-break of the ABI, thank you.

Myers: is it necessary to indicate that constants of these types are unusable in `#if`?

Gustedt: the effect of this change is that constants for extended integer types are valid *pp-numbers*, but they might not be usable in `#if`. This disappears with the `WIDTH` feature; before people used `MAX` macros but the `WIDTH` macros are small and won't exceed preprocessor arithmetic limits.

Myers: the line about "shall be suitable for use in `#if`" no longer works for `MAX` macros now unless they are in range.

Meneide: the feeling is that we should fix that limitation in the preprocessor so as not to depend upon `intmax_t`. Older code couldn't reference such values and cannot break from this. There is a risk to new code, but strongly in favour.

Gustedt: could add a Recommended Practice to prefer `WIDTH` for this reason.

Ballman: as with `intmax_t`, n2775 is related and runs into the exact same problem with `#if`.

Keaton: for the wording in the second change, N makes sense. In the first change N is not used and reads strangely. "Specific width" would read better.

Gustedt: Agree, was trying to stay close to the original. Think this is editorial.

Banham: it seems p3 tries to say that other widths may exist but must define ones equivalent to the Standard ones?

Gustedt: again, only if they exist. The `least` types are mandatory, but these are not if a matching type doesn't.

Banham: I think this loses that sense of optionality, an implementation may have other sizes.

Gustedt: not what is meant. This refers to common sizes, not to all Standard types.

Banham: do we imply that a two's complement type can have padding bits?

Uecker: the second change defeats the point of `intmax_t` – could we deprecate it?

Gustedt: really there's a need for `int128_t`, everything is in place for a platform to add it and this proposal just lets them. `intmax_t` is still used in ABIs and interfaces and is more complex.

Uecker: this no longer tries to solve `intmax_t` at all.

Gustedt: we're giving up on doing that for C23. We can reach an agreement for now without having solved the general problem.

Uecker: it prevents us from fixing the problem by extending it. We may as well deprecate it right away.

Myers: it may be desirable to ensure that it cannot be `bool` and that an implementation must define `WIDTH > 1`.

Gustedt: thought that follows. `int1_t` should be impossible because of the padding… unsure.

Meneide: w.r.t `intmax_t`, it means the definition isn't true. We could fix it later by changing the definition. It's not given up on, we should have room to fix it; we should solve this first and enable implementations to use the types they have. This gives us something they can work with.

Gustedt: and a perfect excuse.

Keaton: suggested minimal change: specify `intN_t` and `uintN_t` to keep the N.

Gustedt: we have already accepted papers needing more work than this.

Keaton: those were opinion polls.

Gustedt: I will take this as homework for tomorrow then.

Straw poll: (opinion) Does WG14 want to adopt something along the lines of n2821 into C23?

Result: 20-0-2

Bhakta: OK if this is editorial, otherwise I would object.

Krause: this is semantic – if I only have an unsigned type, now I need to provide a signed type.

Gustedt: the homework will address that.

5.34 Gustedt, Pointers and integer types [N 2822]

Very similar covering `intptr_t` and `uintptr_t`. Optional but almost universal, only unsupported by platforms with 128-bit pointers and 64-bit `intmax_t`. If they exist, a conversion from `void *` and back is identity, but does *not* imply a flat address space. They are not there for arithmetic or even for alignment. Following TS 6010 it can be used for ordering, but not arithmetic.

The change itself is super simple, one-word.

Banham: the purpose of having these types is not clear as you can't do much with them. It is difficult to convince developers in a flat addressing environment not to do that.

Gustedt: motivation comes from TS 6010 where they are used in the memory model by imposing an ordering constraint. It helps the SG to make these mandatory.

Keaton: combine the homework on both.

Krause: this is good as-is, though the footnote helps.

Friday

5.21 Bhakta, Clarify the meaning of obsolescent [N 2804]

Question raised in many past meetings why we have deprecated features if they are going to stay that way. This has a different meaning, “obsolescent” is the right word. Ensure we all interpret it the same way.

The term of art definition is in section 2, not just an English word. Means “in use but discouraged”. It is vacuous to say it may be removed in the future because that makes no difference. *Saying* features are in use is important. It does not solely mean that things must be removed, which is true of any feature.

Alternative 2 removes obsolescence from 6.11, as we don’t want to encourage removing functionality without a replacement for a feature. Based on the last meeting we should drop obsolescence from features that have valid uses. Allow different interpretations without risking the features in the Standard.

Myers: I support Amternative 1 and not Alternative 2: features should be removed, not the word. I could find no ISO rule about obsolescence that we need to follow. Because there is no rule there is no requirement.

Ballman: I support neither. Anything can be removed but the marker is a warning that something might be going. Without the warning the term is useless.

Bhakta: it maintains a warning not to use something in new code, the term is not useless because it discourages use.

Krause: w.r.t 6.11.3/6, I want them gone, but while the argument that anything can be removed is correct, there’s still more warning for the user that the risk is higher. Neither wording is perfect.

Bhakta: so what are the issues with the current definition?

Krause: anything can be considered for withdrawal, but we need to mark features as likely to go sooner.

Bhakta: can do this in the revised wording.

Gustedt: agree with Krause that the term is more political, like Recommended Practice – the statement that something is better for removal or replacement is deprecation, not obsolescence. I don’t like removing obsolescent in Alternative 2.

Gilding: it creates a sense of “threat” without commitment. We can determine that a feature needs replacement without settling on what that replacement is, so this is a warning of that feeling.

Meneide: agree that removed features must be replaced with something. We removed K&R definitions in 2019; now we *have* replacements but we didn’t back then; several use cases were lost (VLAs, varargs, assembly) but we still need to say we’ll remove it. We need a new definition of “deprecated” for things we intend to replace.

Bhakta: I don’t like to bank on the future. Don’t assume a replacement will come unless it is part of the removal.

Ballman: while I prefer the second sentence, I was struck that “use isn’t widespread” implies that there is some use of a candidate for removal. Users care about risk when changing Standard versions; will something exist 30 years later?

Bhakta: depends very much on the customer set; K&R is used by some, but being in the Standard doesn’t imply use.

Ballman: sorry – in the Standard implies something is stable and expected to exist in future versions. Marking it means that it may not be there, for things we don’t *want* to support forever. This discussion started because of features that have been obsolescent for 30 years. Importantly they’re “expected” to go away, but unsure when.

Bhakta: agree that we want the implication that it may be removed.

Gustedt: agree with Ballman – “wide use” is very debatable, removing that word is a good idea. Focus on retention.

Uecker: the C99 Rationale touches on this: wide use is irrelevant, any feature used at all shouldn’t be removed without a replacement.

Steenberg: there is value to having obsolescent stuff in the specification, we may choose to implement these. Some things are not how you *should* do something but still implementable.

Straw poll: (opinion) Does WG14 want any change to the Section 2 term “obsolescent” in C23?

Result: 5-11-8

Bhakta: I voted no! But I thought it was obvious. Will not bring a new paper.

Voutilainen: misconceptions are unlikely to resolve with more text.

Banham: I would vote against “deprecate” as well.

5.22 Seacord, Volatile C++ Compatibility [N 2743]

This is taken from the compatibility SG. C++ deprecates some uses of `volatile`. We consider the equivalents in isolation for C.

Implementation-defined whether reading the result of an assignment to `volatile`; the user can easily make this explicit either way.

`volatile` has no semantics in parameters for calling convention and can be ignored. It is nonsense on a return value.

Compound assignment is possibly misleading (though much backlash from embedded developers who want it to be un-deprecated in C++). Pre- and post-increment or decrement are also misleading.

`volatile atomic` is contradictory.

Ballman: de-deprecating compound assignment has consensus from EWG. This will likely be in C++23. I support the rest very much, especially on return and parameter types, as headers are shared most frequently.

Gilding: we should remove `volatile atomic` altogether if it’s truly contradictory. But not if there’s any use case. It never seems to make sense for any non-lock-free type.

Krause: assignment has implementation-defined behaviour; this is not a reason to remove, there was a use case. Implementation-defined is not justification to remove by itself, I would rather define it either way. I agree to keep compound assignment. Post-increment is just syntax sugar and should be the same as compound assignment.

For function types it can always be left out, you can still have `volatile` parameters without exposing that in the header file. It’s an automatic variable so it should be just like any other. This is just the same as banning `const` parameters in definitions. A lot of this is not header stuff and so C++ is less of a concern.

Seacord: the proposal is to obsolete and not to remove. C++ is deprecating because all of these cases are misleading specifically with `volatile`. It implies a very well-sequenced behaviour – the number and order are clear but in these cases it can diverge, and we care about that *and* compatibility.

Myers: I disagree with all of this. The perspective is that `volatile` ensures relations of operations between code and hardware; other uses which *are* true are still valid. It’s not about hardware optimizations, it’s about inhibiting optimizations and forcing sequenced writes, ensures code transformations don’t happen, ensures values are preserved (e.g. in the case of `longjmp`); all of this works very well. It should be OK to inhibit transforms of a value and adding it anywhere should still work. In these cases we don’t care about the number of reads.

Function returns are not covered because they are meaningless; allowing them at all is meaningless and no point obsoleting it. With atomics, the transformation inhibition still makes sense. So as a feature to avoid reordering it is reasonable in all uses excluding return types. Simpler to allow all or none on the return type.

Seacord: was driven to write by the view of `volatile` as meaning externally modifiable, but it does multiple things and some of them are OK. The counter is that these uses can be misleading depending on intent.

[Seacord disconnected, advancing to next topic]

5.19 Gustedt, Types and sizes v1 [N 2838]

Have changed the wording to “particular width” and “corresponding `typedef` names”.

Myers: we still have the issue of the constants being usable in preprocessing expressions.

Gustedt: that is not part of the change. Also the `MIN` and `MAX` values need a change of wording because they may be out of range of preprocessing arithmetic.

Meneide: if someone follows this up with a fix for preprocessing math, is that a continuation for C23 or a new feature? Since it extends the topic to preprocessor evaluation.

Gustedt: take that as an explicit action item if we vote yes.

Keaton: yes, that could be a continuation.

Banham: to clarify, will that fix the constant macros for small integer types? `INTN_C` etc. where the conforming non-magic solution is signed because of promotion rules, choosing cast vs. suffix.

Gustedt: this is an orthogonal change.

Bhakta: I am uncomfortable voting Yes without a fix here. Can the vote be conditional, can't vote Yes without a solution?

Krause: implementations can already provide these. Does this add a problem? Extended integer types are already unusable, I don't think this makes the problem worse.

Straw poll: (decision) Does WG14 want to adopt change 3.1 in n2872 as-is into C23?

Result: 20-0-0

ACTION: Meneide or Gustedt to bring a continuation paper about the use of extended integer constant macros in `#if`.

Bhakta: we already gave opinions yesterday.

5.22 Seacord, Volatile C++ Compatibility [N 2743]

[resuming]

Gustedt: I favour simple assignment but mostly for stylistic reasons. It's bad to remove this for compound assignment. We shouldn't touch function return types. I strongly think we should *not* remove this from atomics – this is used throughout the Standard and we cannot deprecate it without significant action.

Banham: atomics achieve one set of semantics and `volatile` another, with incomplete overlap. WG14 understands `volatile` to be the "old way" but this is not true. Atomics are predicated on hardware threads with cache; a single-core system doesn't have coherency; pseudo-threads can still data race, but not as badly. `volatile` directs the optimizer. ARM only supports `atomic` when using cache, and bypassing the cache breaks.

Krause: the arguments only really make sense for consistency – single opinion poll?

Ballman: I definitely want to poll function signatures separately for C++ compatibility.

Sebor: does this have different semantics in C++?

Ballman: no, but it would cause incompatible headers, the biggest concern being when C++ errors outright on it. I don't want to need `#ifdef`.

Uecker: header compatible is valuable, so is self-consistency within C. This feature does have some use cases.

Myers: the proposed change touches three places: returns have no effect; `volatile` on parameters in the definition does have an effect, probably unshared.

Sebor: this feels firmly in the Quality of Implementation domain – C implementations will start issuing warnings, but if these are errors I would expect implementation diagnostics already. e.g. qualified return types really ought to warn, implementations are already free to do so without community consensus; so not in favour.

Bhakta: for the C++ case, we can vote on the signature case and seek feedback from the SG.

Seacord: the most compelling case is the liaison request.

Ballman: WG21 know about it already and we can vote either way.

Straw poll: (opinion) Does WG14 want to adopt something along the lines of n2743 into C23?

Result: 5-12-5

5.23 Svoboda, Can Signed Integers Overflow [N 2817]

This is close to Seacord's proposal so we should vote after hearing 5.24 as well.

Right now, signed overflow is UB, while unsigned "wraps" and doesn't overflow. These are exclusive. Goal is to clarify the wording and not to change any behaviour. Unsigned integers overflow and wrap around to handle it. This originates with treating unsigned integer overflow as a thing; in GCC there are builtins which led to `ckdint.h`. The argument is that overflow is an error condition requiring handling, differently by the different types; pointer arithmetic also overflows.

The counterargument is that if unsigned integers are a ring, then overflow makes no sense. Users do not view them as a ring; rings don't define division or modulus. This changes "can't" into overflow plus wrapping.

Krause: does conversion to `bool` imply overflow?

Svoboda: yes, there's overflow on unsigned to `bool`, handled by 1-bit wrapping.

Sebor: I was surprised but came around, I understood that unsigned integers *do* overflow. We would also need to change the non-normative notes – does this need a survey?

Gilding: C++ wording intentionally says there is no overflow – is this a problem?

Ballman: yes, accepting that this is the opposite term with no behaviour change, we could present it to WG21.

Tydeman: so some overflows are UB and some are not?

Svoboda: yes, this introduces non-UB overflow. But floating point also has that.

Tydeman: it's not UB in IEEE, but in others it may be. Still, it's consistent.

5.24 Seacord, Clarifying integer terms v2 [N 2837]

We have the same objective – no behaviour change, only terms.

The C99 Rationale defines unsigned integer semantics as always-defined. We want to preserve the same definition used by C++, Ada, Fortran, etc. Four types appear in the Standard: integer arithmetic, pointer arithmetic, floating point, and buffer overflow. So we propose always saying "integer" overflow.

The C++ idea is what we should adopt: define unsigned mathematics as modulo so that arithmetic never produces an unrepresentable value. Our text is wrong, because it says "reduced"; we also spell "wraparound" differently from everyone else. We should add a term of art.

Annex H is descriptive and seems no longer needed in the Standard itself, confusing to users. 6.2.5 changes the Rationale only but not clear what actually happens.

Svoboda: the 6.2.5 change is incompatible with mine. Wouldn't this be the place to use the term "wraparound"?

Seacord: yes, trying to be explicit.

Gustedt: I like most of this – not pointer arithmetic, and I want a separate discussion about Annex H. We don't have a definition for the meaning of pointer arithmetic overflow, so prefer not to touch it yet. We may need the Memory Object Model SG to discuss it first.

Krause: I like defined wraparound and removing Annex H. We should consider these two in isolation even without the rest.

Bhakta: why "unsigned types" and not "unsigned integer types" in 6.2.5?

Ballman: this is an artifact of copying C++'s wording.

Seacord: it's not wrong, it's editorial.

Bhakta: I like the idea of only comparing the main part and separating the Annex H and pointer discussion.

Seacord: agreed, let's split into three.

Sebor: so this is the bigger change. It's incompatible; did you try to converge?

Seacord: yes but we have different views, so we decided to offer both options. Both are teachable, both want to clarify terms and a precise understanding of the meaning.

Svoboda: will be happy if both get positive opinion polls and would try to unify them.

Sebor: what do we lose if we choose Svoboda's proposal?

Seacord: *notional* compatibility with C++. Teaching is slightly more complicated.

Svoboda: there's also a merge conflict on 6.2.5, so we cannot adopt both as-is.

Ballman: thank you both. I am wary of the first for both the notional difference from C++ and the use of "wraparound" without a term of art. The second is largely unifying with C++, which simplifies it for the users and implementors.

Svoboda: mine introduces the term but doesn't define it. The term is compatible. If we favour that paper we should raise it in the liaison SG.

Steenberg – in the UB SG, we discussed defined signed overflow, and also UB for unsigned integer overflow. We could define another type specifier in future for swappable behaviour.

Seacord: it is hard to consider future language features.

Svoboda: accepting this would make it easier to discuss.

Banham: can the wording be tightened based on the adoption of two's complement arithmetic? Sign is now an interpretation, but could be much more precise. For instance an ALU uses the overflow flag for multiplication but the carry flag for addition and subtraction: this tells us something useful.

Svoboda: DEC Alpha has no carry flag.

Banham: the carry flag is useful for extending the result to a wider type.

Gilding: wraparound should be a formal term of art. There is no problem with this for future specification.

Bhakta: I like clarification without change; I like Svoboda's better because it uses terms in the existing Standard, and I prefer simplicity.

Gustedt: I prefer the opposite, want explicitness in changes for clarity. I very much ,like the explicit definition of wraparound as modulo, and the idea of "modulo types".

Bhakta: agreed, I also like "integer overflow" over just "overflow".

Svoboda: w.r.t other footnotes needing changes, I searched and found all the same occurrences Seacord did; didn't consider any of them to need changes. One may exist but I didn't find it.

Straw poll: (decision) Does WG14 want to adopt n2817 as-is into C23?

Result: 7-9-6

Svoboda: poll Seacord's as well before we ask the opinion questions.

Straw poll: (decision) Does WG14 want to adopt the changes in 6.5.6 p9 in n2837 into C23?

Result: 15-2-6

Straw poll: (decision) Does WG14 want to remove Annex H from C23?

Result: 15-0-7

Straw poll: (decision) Does WG14 want to adopt the remaining changes in n2837 as-is into C23?

Result: 19-1-3

Straw poll: (opinion) Does WG14 want something along the lines of n2817 in C23?

Result: 4-9-8

Bhakta: what's different except for the 6.2.5 conflict?

Svoboda: Seacord assumes it can't overflow, mine assumes it does and that it's handled.

Tydeman: Annex H is gone; one of the new floating point annexes should take its place.

Keaton: it's editorial, but preserving the following order is sensible. Thank you both, this was interesting.

6. PL22.11 Business (Agenda 9)

9.1 Approval of Previous PL22.11 Minutes [pl22.11-2021-00007] (PL22.11 motion)

Ballman moves, Pygott seconds, no objections.

9.2 Identification of PL22.11 Voting Members

No new members.

9.2.1 Members Attaining initial Voting Rights at this Meeting

None.

9.2.2 Members who regained voting rights

None.

9.3 PL22.11 Voting Members in Jeopardy

None.

9.3.1 Members in jeopardy due to failure to vote on Letter Ballots

None.

9.3.2 Members in jeopardy due to failure to attend Meetings

None.

9.3.2.1 Members who retained voting rights by attending this meeting

None.

9.3.2.2 Members who lost voting rights for failure to attend this meeting

None.

9.4 PL22.11 Non-voting Members

9.4.1 Prospective PL22.11 Members Attending their First Meeting

Bill Seymour.

9.4.2 Advisory members who are attending this meeting

None.

9.5 PL22.11 Meeting Votes

Bhakta: there are two votes on the agenda.

9.5.1 Systematic Review, ISO/IEC TS 18661-1 - Floating Point Extensions for C - Part 1: Binary floating-point arithmetic. [pl22.11-2021-00010]

Motion to confirm: Tydeman moves, Ballman seconds, no discussion.

Decision poll: 11-0-0

9.5.2 Systematic Review, ISO/IEC TS 18661-2 - Floating Point Extensions for C - Part 2: Decimal floating point arithmetic. [pl22.11-2021-00011]

Motion to confirm: Tydeman moves, Ballman seconds, no discussion.

Decision poll: 11-0-0

9.6 Other Business

None.

7. Resolutions and decisions reached

7.1 Review of decisions reached

Decision: five day meeting only.

Decision: Does the Committee accept the changes in n2781, with `.` substituted for `;`, into C23?

17 / 2 / 4 (yes)

Decision: Does WG14 want to make variably-modified types mandatory in C23 as specified in n2778?

10 / 5 / 9 (yes)

Decision: Does WG14 want to adopt n2770 as-is into C23?

20 / 0 / 1 (yes)

Decision: Does WG14 want to adopt the changes to 5.2.4.2.2 p4 added by n2806 into C23?

14 / 0 / 3 (yes)

Decision: Does WG14 want to adopt n2805 as-is into C23?

10 / 1 / 5 (yes)

Decision: Does WG14 want to adopt n2747 as-is into C23?

12 / 0 / 5 (yes)

Decision: Does WG14 want to adopt n2842 as-is into C23?

17 / 0 / 2 (yes)

Decision: Does WG14 want to adopt n2843 as-is into C23?

16 / 0 / 2 (yes)

Decision: Does WG14 want to adopt n2790 as-is into C23?

15 / 0 / 5 (yes)

Decision: Does WG14 want to adopt Alternative 1 from n2823 into C23?

5 / 3 / 10 (no consensus)

Decision: Does WG14 want to adopt Alternative 2 from n2823 into C23?

4 / 4 / 10 (no consensus)

Decision: Does WG14 want to adopt the changes in 7.22.1.6 p7 of n2832 into C23?

9 / 1 / 9 (yes)

Decision: Does WG14 want to adopt n2845 as-is into C23?

17 / 0 / 1 (yes)

Decision: Does WG14 want to adopt n2846 as-is into C23?

18 / 0 / 2 (yes)

Decision: Does WG14 want to adopt the suggested change in n2848 as-is into C23?

19 / 1 / 0 (yes)

Decision: Does WG14 want to ensure the definition of INFINITY in `math.h` matches the definition in `float.h`?

18 / 0 / 0 (yes)

Decision: Does WG14 want to adopt the first proposed change in n2808 into C23?

23 / 0 / 1 (yes)

Decision: Does WG14 want to adopt change 3.1 in n2838 as-is into C23?

15 / 2 / 4 (yes)

Decision: Does WG14 want to adopt change 3.1 in n2872 as-is into C23?

20 / 0 / 0 (yes)

Decision: Does WG14 want to adopt n2817 as-is into C23?

7 / 9 / 6 (no consensus)

Decision: Does WG14 want to adopt the changes in 6.5.6 p9 in n2837 into C23?

15 / 2 / 6 (yes)

Decision: Does WG14 want to remove Annex H from C23?

15 / 0 / 7 (yes)

Decision: Does WG14 want to adopt the remaining changes in n2837 as-is into C23?

19 / 1 / 3

Decision: Does PL22.11 confirm 00010?

11 / 0 / 0

Decision: Does PL22.11 confirm 00011?

11 / 0 / 0

7.2 Review of action items

ACTION: Ballman to change the WG14 page to promote the full document log.

ACTION: Keaton to investigate status of n2566.

ACTION: add n2761 to the papers-of-interest list.

ACTION: Tydeman to take n2797 to the liaison SG.

ACTION: Steenberg to send commentary on the problems with namespaces to the reflector.

ACTION: Meneide or Gustedt to bring a continuation paper about the use of extended integer constant macros in `#if`.

8. Thanks to host

Keaton: Thanks to ISO for supplying Zoom capabilities.

9. Adjournment (PL22.11 motion)

Seacord moves, Ballman seconds, no objections.

Adjourned.