It has become clear to me that the wg14 as it currently stands are interested in taking the language in a different direction from what this paper proposes. I consider this proposal DOA, and only submit it as a record of my work. Where this leaves IOS C and the extent of my further participation is yet to be determined.

A proposal for C2Y

n3176
By: Eskil Steenberg Hald
Representative of Sweden in the ISO JTC1/SC22/WG14

Note: C2X is the commonly used name for the next major revision after C17, expected to be released in 2024. C2Y refers to the version after C2X. This document also uses the term C2Z in reference to the version expected after C2Y.

At the time of writing The ISO WG14 is completing its work on C2X, and is in the early stages of contemplating what could go into a future C2Y. Due to some changes in the ISO rules, the working group would be disbanded if no project intending to become an ISO publication is ongoing. This is not an immediate concern of the WG14 since many study groups are progressing with works intended for publications besides the major standard revisions of C. However, eventually it is in the interest of the working group to plan for future releases of C. This proposal is trying to help this process by giving the working group a more tangible proposal for how such future work could progress. It is the author's experience that it is easier to make progress when there is a proposal to discuss, even if the resulting progress is to reject the proposal and find a consensus around another direction.

Background

There are multiple different views of the right direction for C to evolve.

Some argue that the next release of C should strictly be a maintenance release, to fix any issues with the standard rather than to add new features. Historically, major C releases have been roughly divided into "feature releases" (c89, c99, c11,c2x) and "bug fix releases" (c90, c17). There is a desire from some to adopt a "tick-tock" release cadence where every other release is a feature release and every other release is a bug fix release. While we are in favor of a conservative approach to Cs development, a straight bug fix release will not be able to address many of the issues C has, even issues that can clearly be identified as bugs in the standard due to backward compatibility.

Yet others argue that C should evolve faster to "keep up with the competition" and add new features like Labdas, operator overloading, name spaces or memory safety.

As its mandate the ISO standard it is tasked with standardizing industry practice, and not engaging in the inventions of new language features. This mandate, while good intentioned, does not offer enough clarity to guide our work. Some argue that a feature being used in a different language does constitute industry practice. This interpretation makes the rule almost meaningless. Another interpretation of industry practice, is that it needs to be implemented in one, or preferably in multiple implementations (although not necessarily in the exact same way, hence the need for standardization). Yet another interpretation is that it needs to have been shown to be widely adopted by its users. This is a test where many proposals fall far short. Many proposals come with implementation proof of concepts, and where the uptake by the community is often non-existent. At the same time, there are common extensions that are widely used (like packed structs) that are not added to the standard. (the authors of this paper are neither proposing nor taking a position on this possible addition) The purpose of standardizing existing practice is not merely to verify that an idea is implementable, but that it is something the wider community is using. Even when the wg14 applies the strictest interpretation of "industry practice", some invention is required by the wg14 to reconcile various similar extensions.

There are those who believe that C needs to become safer, particularly in terms of memory safety. The authors of this paper welcome any effort to make C a better language. Such wide ranging changes to C would require extensive evaluation by implementation, adopted, and then evaluated, before being considered for standardization..

Our vision: C should be for C programmers

C is a successful design. Whatever opinions one may have about it, its design is proven to produce wildly successful software projects. As such it needs to be protected. C is not just a useful language, it is an indispensable language. If the C model will be successful in the future is anyone's guess. Other languages are being developed, and they may (and hopefully will) at some point in the future prove to be valuable innovations that surpases C, but that innovation will and should happen outside of C.

It is pure fantasy that the detractors[1][2] of C will one day abandon the alternatives, and embrace C because of some wg14 added features. C will always be "juggling switchblades", it will always produce "Nasal demons" and it will always lack conveniences. Users who object to C on the grounds of its lack of safety will not change their minds if C adds substantial additional safety. Other languages already occupy this space and are better suited to fulfill these requirements

C will never be a “Modern” Language. This is the nature of C. Trying to change this is to try to change C into something it isn't. For anyone wishing to use a language with these features there is plenty to choose from. In this sense, we are very lucky to have C++ for anyone wishing to have a C like language but with vastly more features. Additionally these features are rarely well designed by committee. Uptake of new features that have been added in the past like _Generic,

VLAs, anonymous structs, bounds checking is limited too. Features that are added aren't being used, taught, or even known by the wider C community. Many new features have to be shoe-horned in, in order to not break backward compatibility and they therefore end up being poorly designed. A lot of these features also significantly complicate further development and implementations since they require a lot of special case attention.

C should be guided by those who like C, and who believe C should retain its identity . The purpose of the C standard should be to serve this C community [3], not to placate its detractors or to be in competition with other languages.

No one is forced to use C, and there is no harm when people choose other languages. The existence of C is not preventing anyone from proposing or adopting new languages. There is however harm when C changes in such a way that those who have chosen C, no longer have the language they chose available to them. With a massive investment already made in to the body of existing C code, C developers have the right to be protective of the language they have chosen. C is not a language that is trying to be everything for everyone, and after 50 years of success it should be comfortable with its position.

The rejection of features.

To choose C is a actively reject features [4][9]. For anyone who wants a more feature rich language, C++ is available and widely implemented. Why does so many people choose to use a subset of features when a larger set of features is available? The simple answer is that the simplicity, and clarity of C is one of Cs greatest assets. C is for people who prefer control and clarity over convenience, and the ability to execute clever syntactic tricks. For the user who wants a language with many conveniences there are many other options. For the user that wants simple clarity without syntactic abstraction there is really only C. It is therefore especially important that the wg14 protects this space, even if a majority of programmers prefer a feature rich language with many conveniences.

This is reflected in the users of C. Many still prefer writing code adhering to the C89 standard and style[5] or adopting newer versions with strict limits[6], where very few features are used. The adoption of new C features is very limited. A language, whose best selling point is its compactness and simplicity, is not best served, by only adding new features, with no mechanism to remove or repair existing features. The wg14 should work to find the mechanisms required to cater to users who chose to program in C.

Many C programmers probably have a pet feature they would like to add to the language to address some deficiency corner case. The problem with such additions is that if all such additions are permitted, then the language is no longer simple. The author of this paper, have previously submitted a proposal titled "break break" [N2859] [7], to address the comparatively common issue of jumping out of multiple loops/switches, with a simple sensible addition. The author also admits that even if said feature would be approved for inclusion in the C standard,

they would not use it. C is the lingua franca of computing, that enables code to transcend time, platform and implementation. Therefore to use new features that are only implemented in the latest versions of some implementations defeats one of the biggest purposes of using C. It is not worth breaking compatibility for syntactic convenience. The value of being compatible with older implementations outweighs the value of the feature.

Someone who uses C in order to be portable, will only adopt features where it's worth writing two implementations: one with the feature and one without the feature for backwards compatibility, and then use the pre-processor to select the one supported by the C implementation. This places a high bar of new features but not an insurmountable one. Any feature that saves the developer typing, are obviously out, but features that lets the user access hardware features like, bit utilities and _BitInt(N) do make sense.

C, A universal format for instructions.

Computer languages are generally seen as tools for giving computers instructions for what to do. With this view language and implementation are closely connected. New features in the implementations are reflected in the language and users are expected to consistently follow this development. C on the other hand is more of a portable format for storing instructions. It is used in order to be independent from implementation. If you want to implement an algorithm once and for all, and want it to be accessible to all, and be usable for decades to come without maintenance, on any platform, then the best way to store your implementation is in C. If you ask what language and dependencies should be used to implement something, there is a wide range of opinions and reasoning, however when you ask what non-native code people prefer to integrate in to their projects an older version of C is almost always preferable, because it is such a known entity, likely to be fast and efficient and is guaranteed to be build on almost any platform.

Implementability

The Implementability of C should get renewed focus. The fact that a reasonably skilled programmer has been able to implement a functional compiler alone has been a great asset to C, and the teaching of computer science. Given the emergence of well designed open-source compiler infrastructure, that facilitates the modular addition of new back and front ends, has reduced the need to write new C implementations in order to support new platforms. However, a lot of the innovation around implementations in the C community revolves around tooling. C has a rich ecosystem of debuggers, linters, analysers and other tools that keeps growing. C has by a good margin the best tooling for analyzing and presenting code. It is the strong belief of the author that over the coming decades, the greatest innovations in C especially in the areas of safety and security will come in the form of tools. Therefore the WG14 should support these

developments by considering the impact on tooling that features have. For example, are unnamed constructs considerably harder to write good error messages for than named ones.

Features unadopted by users, still greatly increase the complexity of the language for implementers. A core value of C is that it is implementable. For C2x several complicated features have been added like new uses of auto, nullptr and constexpr that the wg14 has had difficulty giving a rationale for the users adopting. They are borrowed from other languages where they have features C2x doesnt have and interact with systems that C does not have. "It can have its uses" is not a good enough rationale for adding features to C, a language where the key feature is its simplicity.

The C standard is tasked with standardizing industry practice. From one perspective that means having C embrace things that are the practice in the wider computing industry. In this sense the goal is to narrow the gap between C and other languages. However doing so is widening the gap between standard C and how C is actually used in the industry. By making standard C harder to implement and with fewer implementations able to meet the requirements of the standard, while at the same time meeting the requirements of the vast majority of C code. The wg14 runs the risk of forking the language between ISO C and the classic C used by the community. The inability to come to an agreement with major implementers and important projects like the Linux Kernel on such fundamental issues as tread safety and the memory model, further divides the community.

Issues facing C.

Before discussing the practical question of how C should develop, We would like to start by outlining the issues we think future versions should engage with. A note of caution, we will here in the broadest terms describe the kinds of issues that we should focus on, but avoid prescribing the solutions or even the precise issues that need to be addressed.

-C is today, in some ways, a very simple language. That is its main strength. It has very few constructs and features. However in other ways it can be mind bendingly complex in its details like its types, and memory model. C2x has 3 different ways of expressing NULL that in turn are not necessarily compatible with NULL characters termination. The language guarantees there is no padding before members of a union, but does not guarantee that they share the same address. Anyone who has been a member of the wg14 long enough is likely to have a collection of these strange anomalies. For the millions of people who write C, these anomalies remain opaque. They are rarely mentioned in books or courses and they very rarely actually have real world impact (until they do). In fact, I would argue that the best way to learn about them is to join the wg14, where the few people who truly understand and can explain these anomalies reside. (And for providing this service the author is eternally grateful)

-Because of these anomalies implementing C has a much higher burden than necessary. The difference between an implementation that supports features used by 99% of all programs, and

implementing a ISO compliant C implementation is significant. Ease of implementation is an under rated feature of C, and it deserves our attention. New platforms and hardware designs tend to chose C as the language of choice, because of its implementability and this needs to be a top priority of C. Additionally and perhaps even more underrated is how ease of implementation aids greatly in the growing ecosystem of debuggers, analysers, sanitizers and other useful tools.

-C has a number of features that should not exist like, under declared types, sprintf, bitfield and VLA, that a big majority of the community agrees are bad and should not be used. MISRA and other secure coding guidelines along commonly issued warnings by most major implementations, show that there is much one can do in C that no one thinks one should do. The fact that 5[a] is equivalent to a[5], does not make a C a better language. It makes it legal to do something no sensible programmer should ever do, and deprives the users of useful error messages. C has many ambiguities and hidden meanings that do not clearly convey user intent. The difference between the declaration: function(void) and function() is one such example. This complicates the learning of the language significantly. A complete C course will include many things that the students must know in case they encounter it, but should never use. C also has features that are explicitly declared as deprecated, yet do still remain in the language. Further there are other functionality like sprintf that for security and safety reasons should not be used.

-C has a number of features that misleads the users into thinking that the language is a "portable assembler" and does not accurately reflect the language, such as inline, constexpr, and register, and are in practice nullified by the as-if rule. They do not reflect the plurality of implementation approaches that we would like to foster.

-Text handling is complex, convoluted, has many security issues and has not aged well, as the world has converged on unicode, and UTF-8. Much of this originates from the original sin of null terminated strings. While It is the author's opinion that there are so many valid approaches to storing text, that it would be wrong and is wrong for the wg14 to endorse one, we should work to enable users to implement their own (by for instance providing array/lengths version for many common text interfaces.)

-Cs memory model, is another long running problem, where despite years of work and dare we say an heroic effort by the memory model group, it remains an issue where perhaps the most well known open source C project, the Linux Kernel, explicitly chose to ignore a standard they are unhappy with.[8]

-The atomics that were introduced in C11 have failed to gain broader tracktion and a major platform has declared it unimplementable. Despite this, all major platforms do support near identical atomic operations on volatile types. In hindsight, this was a failure of not enough implementation experience and a failure to adhere to the mantra of standardizing industry practice.

-There are some types of UB that should be addressed. In order to not derail this paper, we decline to elaborate what UB we are referring to.

The main reason that prevents the wg14 from solving these issues, is that it risks breaking backward compatibility. Backward compatibility is further complicated by the fact that implementations want to continue to successfully run important programs that do in some way violate the standard, and depend on non standard behavior. An example of this would be reallocating an allocation to the size zero. It is Undefined behavior, in C2X but only because it allows implementations to continue to support applications that rely on diverging behaviors. The behavior is therefore not only undefined, it is "undefinable" because any definition would break some backwards capability. The issues can therefore from a certain perspective be seen as "unfixable".

-For completeness we would also like to add that a serious issue is that C still lacks vector operations. Vector operations are industry standard hardware features and have been for a long time, and are more common than other data types that C does support like atomics and complex. A wide range of hardware, from CPUs to GPUs are only able to reach their peak performance using vector operations, forcing users to use extensions or other languages. Whereas most proposed additions to the language could be considered "nice to haves", the lack of vector operations prevents the industry from using standard C for a lot of tasks, where C otherwise would be the obvious choice. As this is a large undertaking, a working group should probably be formed to explore how this could be done.

The proposed solution:

Our proposal is to not add any new features to C2Y (with one small exception discussed later). Instead we would make C2Y a subset of C2X. Any compliant C2Y program would be compilable with a C2X compliant compiler. However a C2Y only compiler is not required to be able to compile any previous version of C, as some C2X functionality would be a constraint violation or UB. An implementation should also be able to assume that any C2Y code is not legacy code, and therefore make no effort to compile any code that in some way violates the standard. Our expectation is that all major implementations would retain support for earlier versions of C indefinitely. Our goal should be that 98% of all existing Iso C compliant files written in the last 20 years should also be C2Y compliant without any change whatsoever. C2Y should reflect how the language is being used. Ideally we could produce tooling for the wg14 to use to verify the prevalence of features in large bodies of source code.

The one addition we would make to the language is a new pre processor directive:

#version <language version>

That allows a developer to explicitly say at the top of the file that code is compliant to a specific version of C. We would also add a pragma that would have the same effect, but would allow a

C2Y file to be compiled with a C2X compiler that is not aware of this new feature. This would mean that C2Y is explicit opt-in only. C2Y compiled files would maintain ABI compatibility with C2X, and therefore a project can contain a mix of older C and C2Y files. Users can stay on C2X, and see C2Y as "best practices". The goal should be to consolidate the language, not to fork off yet another version of the language with poor uptake.

Once the user has opted in to C2Y, the implementation is able to assume the program will only use the C2Y subset and issue errors if it is not. The user can choose to use the #version directive, to exclude any compiler not explicitly supporting C2Y, or choose the pragma to still be backwards compatible with implementations supporting earlier versions of C. Someone who is strictly using C89, may still want to enable the pragma in order to get a compiler to enforce the avoidance of problematic areas of the standard. This gives users a real reason to adopt the latest version of C, without rewriting their code, and while still being compilable using old compilers.

The burden of new implementations that only implement the C2Y sunset can be reduced. New implementations that do what to support previous versions of C have a well defined subset they can use as a stepping stone to full backwards compatibility. New implementations that do not wish to implement full backwards compatibility can implement selected features as extensions.

C should have a mechanism for explicitly stating the version of the language that an implementation expects. This form of versioning would truly come into its own in C2Z, that could be a superset of C2Y. In fact work on C2Z could run in parallel with C2Y. To illustrate this let's consider the following example:

In C2X calling realloc with the size param of zero is UB.

In C2Y, we could change the behavior to be a constraint violation. We have now narrowed the language by no longer allowing implementations to insert platform specific behavior here. Users would have to make sure that their programs comply with this more narrow definition before they mark their files as C2Y files. Implementations would also be able to issue errors if the rule is violated.

In C2Z, we could then remove this constraint violation and define a standardized behavior of the wg14s choice, say free or the allocation of a zero bytes.

We believe that there is a strong argument for having C2Z run in parallel with C2Y, because there are work already in progress on proposals for a future version of C, and it is therefore important that this work is acknowledged and that the wg14 is open to considering papers for additions to future versions of C, while engaging with the proposed depreciation effort.

The viability of C2Y as a production language is a topic for discussion. Depending on what the wg14 decides to remove, the language may miss some critical features for some use cases. For instance the wg14 could decide to remove atomics support completely from C2Y so that it can

be reintroduced in a reworked form in C2Z. One can imagine the adoption of an overly strict memory model for C2Y, that allows for a more lenient model in C2Z that would not be compatible with C2X.

Our proposal for how the wg14 develops C2Y aims to be wide reaching and profound, yet have minimal impact on the language people use. The goal is to have the majority of C developers go "oh I thought it always worked like that" when presented with the changes we make to C2Y. We want to make C a stricter and more explicit and clear language, that removes ambiguity, and forces the user to clearly state intent.

Using this method of controlled deprecation, would give C a rare chance to fix issues and to shrink rather than to forever keep growing. This could be transformative, and solve many outstanding issues. It is the strong conviction of the author that the C community wants existing issues addressed rather than new features.

Additional optional proposals for C2Y:

Work assignment:

There are many issues that are not addressed simply because there is no paper to consider. It would therefore be useful for the wg14 to compile lists of these issues, and use some meeting time to see what fixes have wide support, to solicit volunteers to write proposals. Knowing that there is wide support for a fix, may entice members to write papers since they know that the paper has a high chance of being accepted. Additionally, a fix where the wg14 has in advanced expressed a desire to see a change needs to spend less effort arguing that a change is needed, also saving considerable effort.

Consider formalization:

The current standard is expressed in prose and there have been many suggestions to move the standard to a more formal form. Our proposal makes this a possibility, and the wg14 should therefore consider this possibility, its pros and cons and what form it could take.

Conformance tests:

For anyone who tries to implement C, C2Y should reduce the burden considerably. To further help developers implement C, the wg14 should create a repository of conformance tests that

developers can use to verify their implementations. These conformance tests should ideally both contain positive and negative tests, and be freely available.

Vector operations:

The wg14 should consider forming a study group for vector operations.

Text:

The wg14 should consider forming a study group for text functionality.

References:

[1] https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[2] https://twitter.com/markrussinovich/status/1571995117233504257

[3] https://www.humprog.org/~stephen/research/papers/kell17some-preprint.pdf

[4] http://vger.kernel.org/lkml/#ss15

[5] https://daniel.haxx.se/blog/2022/11/17/considering-c99-for-curl/

[6] https://git.kernel.org/pub/scm/git/git.git/tree/Documentation/CodingGuidelines

[7] https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2859.pdf

[8] https://lkml.org/lkml/2018/6/7/761

[9] https://twitter.com/Lauramaywendel/status/1677437443824386048?s=20

[10] https://twitter.com/_herose_/status/1691148684048609294