The C Standard charter

WG14 N3223
2024-02-23

Abstract

Guiding document with overview of the C Committee's mission and expectations

Authors:

- **Robert C. Seacord**; Woven by Toyota; United States; rcseacord@gmail.com
- **Jakub Łukasiewicz**; Motorola Solutions; Poland; me@jorenar.com
- **Christopher Bazley**; ARM; United Kingdom; chris.bazley@arm.com
- **Jens Gustedt**; INRIA and ICube; France; jens.gustedt@inria.fr
- **Martin Uecker**; Graz University of Technology; Austria; uecker@tugraz.at
- **Rajan Bhakta**; IBM; USA, Canada; rbhakta@us.ibm.com
- **Miguel Ojeda**; Spain; ojeda@ojeda.dev
- **Carlos Andrés Ramírez**; Woven by Toyota; Japan; carlos.ramirez@carlos.engineer
- **Ori Bernstein**; Canada; ori@orib.dev
- **Rashmi Jadhav**; Woven by Toyota; United States; rjrashmijadhav3@gmail.com

Prior art:

- [N444] C - The C9X Charter
- [N1250] C - The C1X Charter
- [N2611] Programming Language C - C23 Charter
- [N2986] Interpreting the C23 Charter

Changelog

- N3223 (2024-02-23)
    - initial version

Introduction

The working group ISO/IEC JTC1/SC22/WG14, commonly known as the C Committee, is the steward of the C programming language, first described in 1978 by Kernighan & Ritchie in book *The C Programming Language*, and now by revisions of *ISO/IEC 9899* standard and other technical specifications. The Committee is part of a broader community responsible for the maintenance and evolution of the C language; narrowly, it is responsible for the normative aspects of the C programming language and its interplay with other standards' bodies.

C is a general-purpose high-level programming language suitable for low-level programming, in other words: *system programming language*. Although its development began on the Unix operating system for the PDP-11 computer, it has since been implemented for practically all devices and systems in the world. It is of fundamental importance for many aspects of computing and data processing. C serves as a *lingua franca* to translate between various systems and languages. C is often used as a target language for compilers, as an implementation language for interpreters, for building operating systems, for embedded programming, for teaching fundamentals of computing, and for general purpose programming. It stands out in terms of portability, interoperability, efficiency, and stability.

The work of the Committee is, in large part, a balancing act. The Committee tries to improve portability while retaining the definition of certain features of C as machine-dependent, attempts to incorporate valuable new ideas without disrupting the basic structure and nature of the language and tries to develop a clear and consistent language without invalidating existing programs. All of the goals are important and each decision weighed in the light of sometimes contradictory requirements in an attempt to reach a workable compromise. The C Committee is also in active liaison with C++ committee. While unnecessary incompatibilities between the two languages are to be avoided and some features of C++ may be embraced, the Committee is content to let C++ be the big, ambitious language, while maintaining C's simplicity.

Areas to which the Committee shall look when revising C include:

- Technical Corrigenda and Records of Response;
- Technical Specifications developed by WG14;
- future directions in the current Standard;
- features currently labeled obsolescent or deprecated;
- requirements resulting from JTC1/SC2 (character sets);
- the evolution of safety-critical software development;
- all known software security issues (programming language vulnerabilities);
- the evolution of C++ and other C based programming languages;
- the evolution of other programming languages;
- the evolution of C implementations, including compilers, libraries and operating systems;
- other papers and proposals from members;
- cross-language standards groups' work;
- other comments from the public at large;
- subsetting the Standard;
- other prior art.

Principles

In specifying a standard language, the Committee uses several guiding principles. There are many facets of the traditional spirit of C, but the essence is a community sentiment of these underlying principles upon which the C language is based. The principles serve also an important purpose in preventing falling into the *design by committee* pitfall. That being said, judgment over technical proposals relies solely on Committee members' expertise, therefore submitters are encouraged to keep these principles in mind when making submissions. While there is a tradeoff between the principles and **none of them is absolute**, the more a proposal deviates, the more rationale is needed to explain the deviation.

The following is a list of principles upon which the Committee revises the Standard:

- Uphold the character of the language
- Keep the language small and simple
- Facilitate portability
- Avoid ambiguities
- Uphold the potential for high performance
- Allow programming freedom
- Codify existing practice to address evident deficiencies
- Do not prefer any implementation over others
- Do not leave features in an underdeveloped state
- Ease migration to newer language editions
- Avoid quiet changes
- Enable safe programming
- Enable functional safety
- Ease library independence
- Uphold interoperability

Uphold the character of the language

C programmers attribute considerable value to its syntax and semantics. New features should integrate seamlessly with the existing language. Attention should be paid to the ideas behind design principles, common idioms and habits, and the effect new features might have on the language's economy of expression.

Keep the language small and simple

Features, and the concepts behind them, should be easy to explain in a clear and concise manner. Language issues should be resolved using minimal new machinery. Simplicity enables both programmers and tools to reason about code, allows for diverse implementations, keeps compilation times short, and helps to achieve other principles.

Facilitate portability

While not strictly required, when plausible, code written in C should aim at being highly portable. The language itself, together with the standard library, should be as widely implementable as possible, while meeting its core objectives. The size and complexity of the language and library should not place an undue burden on constrained hardware.

Avoid ambiguities

Undefined behaviors, *unspecified behaviors*, *implementation-defined behaviors*, and other *portability issues* enumerated in Annex J of the Standard should be eliminated or reduced. **These issues might lead to application vulnerabilities.**

Uphold the potential for high performance

Efficient code generation is an important aspect of the language. To avoid code explosion for apparently simple operations, C mimics the target implementation, rather than creating a more general abstract rule.

Allow programming freedom

Non-portable uses of C, such as direct interaction with the underlying hardware, using features specific to implementation, or seizing opportunities for optimization cannot always be satisfied within a sound set of bounds. Programmers need the flexibility to explicitly bypass checks to do what needs to be done. However, these bypasses should not be required to the extent their use becomes habitual, as they detract from readability, safety, security, and analyzability.

Codify existing practice to address evident deficiencies

Prior art may come from other languages, although C implementations are naturally more compelling. Unless a new feature addresses a significant deficiency, no new inventions should be entertained. Avoid standardizing workarounds instead of long-term solutions..

Do not prefer any implementation over others

C is a language with a wide variety of individual dialects. No single implementation is the exemplar by which C is defined; it is assumed that all existing implementations must change somewhat to conform to the Standard. However, it should be possible for existing implementations to gradually migrate to future conformance.

Do not leave features in an underdeveloped state

Incremental change towards a full solution is a common approach to introducing new features. However, care should be taken that features are not left in a state which could reduce their overall usefulness, hindering adoption and further development.

Ease migration to newer language editions

Developers should be able to mix and match code from different language editions. The bulk of existing codebases should be largely accepted by a translator conforming to a newer language revisions, and the programmer's burden to change code just to have it accepted by a conforming translator must be limited.

Avoid quiet changes

Changes that alter the meaning of existing code causes problems. Breaking changes that require diagnostic messages are easily detected. Avoid silent changes that cause a working program to behave differently without requiring a diagnostic message. Where this principle is violated, informative notes should be added to the Standard.

Enable safe programming

The language should take into account that programmers need the ability to check their work. While not guaranteeing program correctness, properties such as *portability*, *unambiguity*, *memory safety*, *type safety*, *thread safety*, etc. are prerequisite to reasoning about security and reliability. Software interfaces should be analyzable and verifiable. The language should allow programmers to write concise, understandable, and readable code.

Enable functional safety

C is frequently used in the development of safety-critical systems. *Functional safety* is the systematic process used to analyze that a fault does not prevent a program from performing its required function. An analyzable subset of the language is used to create a safety argument; this subset should be enlarged. Unbounded undefined behaviors (that represent a single point of failure) should be eliminated.

Ease library independence

Fundamental language features should be operational without the standard library. Many programmers value supplying the library, or its parts, independently from the compiler vendor. Standalone implementations may serve specific needs such as safety, decreased size, improved efficiency, or compatibility with multiple compilers on diverse architectures. The burden and difficulty of matching implementation details for such use-cases should be minimal.

Uphold interoperability

C serves an important role as the *lingua franca* of the programming world. It is a primary target for *foreign function interfaces* and other languages often expose bindings for C interactions. Such communicative design allows C to exist as a part of larger systems, often providing the means for improving performance, being a gateway to the underlying platform, or translating between components written in different languages.