EDITOR PHIL LAPLANTE
Penn State, plaplante@psu.edu

Effective C

Robert C. Seacord, NCC Group

The world runs on code written in C, yet more can be done to help developers learn to write professional, secure, and effective C programs. This article describes why the C programming language has succeeded and what's next for the language from the perspective of a long-term C Standards Committee expert.

Carl Sagan once said, "If you wish to make an apple pie from scratch, you must first invent the universe." Dennis Ritchie and Ken Thompson did not invent the universe at Bell Telephone Laboratories in 1972, but they did create a highly successful system programming language that can work with a wide range of computing hardware and architectures. After 50 years, the C Language they invented remains as vital and popular as ever.

System languages are designed for performance and ease of access to the underlying hardware while providing high-level programming features. Although other languages offer newer language features, their compilers and libraries are often written in C. C is layered directly on top of the hardware, making it more sensitive to evolving hardware features, such as vectorized instructions, than higher-level languages that usually rely on C for their efficiency.

According to the TIOBE index, C has been either the most or second-most popular programming language since 2001; it was also TIOBE's programming language of the year in 2019. C is a small, simple language that has remained successful by staying true to its principles. C doesn't prevent the programmer from doing what needs to be done. C programs should be fast, even if they are not guaranteed to be portable. C strives to provide only one way to perform any operation. More recently, C has worked toward making support for safety and security demonstrable.

Many developers no longer formally learn how to program in C. When I attended Rensselaer Polytechnic Institute in the early 1980s, all students learned Fortran, while computer science students, who were thought to be capable of understanding while loops, learned WATFIV-S. Many universities transitioned to teaching C or C++ as entry-level languages, although the dean of the School of Computer Science at Carnegie Mellon University (CMU)

Digital Object Identifier 10.1109/MC.2020.3016369
Date of current version: 21 October 2020

felt C++ was too complex for undergraduates to study. The current advanced placement computer science exam tests students on their knowledge of Java, which, in turn, became a popular first language to teach at the university level. More recently, Python has become popular as an entry-level language at CMU and other universities. Unfortunately, this means it is now increasingly common to get an undergraduate degree in computer science or related field without learning how to program in C.

The need exists for an introduction to C that is widely accessible but not so oversimplified that it promulgates the development of incorrect and insecure code.

Brian Kernighan and Dennis Ritchie published *The C Programming Language*[4] in 1978. Frequently referred to as *K&R C* (after the authors), this was the first widely available book on the subject. In 1988, the second edition of the K&R book[5] was published to cover the then-new ANSI C standard, particularly with the inclusion of reference material on the standard library. There have been many C language programming books published since, but none that stand out as a proper introduction to modern, professional C language programming.

Even though the need for C programmers who can develop software for embedded systems, Internet of Things (IoT) devices, and other applications is on the rise, the availability of C programmers is decreasing. The learning path for new C language developers is twisted and runs through murky waters. The need exists for an introduction to C that is widely accessible but not so oversimplified that it promulgates the development of incorrect and insecure code. I published *Effective C: An Introduction to Professional C Programming*[16] to provide a direct instructional path to writing professional quality code.

WHY C?

People commonly ask if they should learn a new language and where and when it is appropriate to use that language. This is particularly true for mature languages like C that have lost that "new car smell." Learning a new language requires significant effort; mastering a language can take years of practice. Consequently, it is worthwhile to research the advantages and disadvantages of a language before committing to the effort.

Fundamental tradeoffs exist in computer science, where improving quality attributes such as security, performance, usability, safety, and robustness can result in diminishing other properties. The C language is the natural consequence of the language designer's goals to produce a small and optimally efficient language. The ability to write machine-specific code is one of the strengths of C, but this also means that programmers are not required to write portable code. Languages such as Java prioritize portability over performance.

Most programming languages manually allocate memory and other resources. C and C++ programmers manually deallocate unused objects for greater control. Many programming languages, such as Java, C#, D, and Go as well as most scripting languages, use garbage collection to automatically reclaim memory that is no longer in use by the program. The Boehm–Demers–Weiser conservative garbage collector can be used as a garbage-collecting replacement for C `malloc` or C++ `new`, although its use is not common.

LANGUAGE APPLICATIONS AND STRENGTHS

C is known as a *systems programming language*, so it is not surprising that most operating systems—including Unix; the Microsoft Windows kernel; Linux; the macOS kernel; and the iOS, Android, and Windows Phone kernels—are largely written in C. Most embedded system and IoT devices are programmed in C. Many of the Google open source community's 2,000-plus projects are written in C. Numerous desktop applications are written in C.

C is commonly used in the development of embedded systems because of the efficiency of the generated code, the simplicity and availability of the compilers, and the availability of development tools. C is a good choice for highly constrained environments. For example, microcontrollers are notoriously space constrained, ranging from megabytes of random-access and read-only memory (ROM) to bytes of ROM and only registers for mutable state. The engineering of these systems can be extremely cost sensitive, particularly in the case of mass-produced consumer devices.

Real-time environments that must guarantee their response within specified time constraints or deadlines are frequently written in C. Real-time systems frequently need to ensure that the worst-case execution time in certain code paths is below a certain threshold for a system to be correct. The thresholds are often on the order of hundreds of microseconds. A wide disparity in execution times makes budgeting difficult. Developers need to minimize the maximum runtime, making the use of garbage-collected languages problematic.

Support for arbitrary pointer arithmetic, manual memory management, unchecked memory access, and the lack of a built-in string type shift much of the burden for ensuring the correctness, safety, and security of C language systems to the programmer. To some degree, this is a natural outcome of the language's design goal of not preventing the programmer from doing what needs to be done. Some aspects of the language, such as pointer arithmetic, may just be a historical artifact that cannot be eliminated. Unchecked memory access is largely an artifact of the high cost of checking these accesses in the presence of pointer arithmetic.

C is frequently disparaged with respect to memory-safe languages such as Rust, Go, Ada, and D. I served on the front lines of the C and Ada wars back in the 1980s. I'm glad C won, but I would rather not revisit the conflict as we lost many good people. All of these other languages are relatively new and generally have roughly a 1% market share. This has disadvantages in the number of experienced programmers available to develop code in these languages and the maturity of the ecosystem and tools. There are numerous, mature, sophisticated tools available for C language programmers, including static and dynamic analyzers; if used correctly, they can significantly reduce the number and severity of defects found in C language code.

It is a general misconception that there is any such thing as a "secure language." One language I can speak about extensively is Java. Along with my coauthors, I began the development of *The CERT Oracle Secure Coding Standard for Java* toward the end of the first decade of the 21st century.[10] What we thought was going to be a small pamphlet evolved to the point where it had to be separated into two volumes.[11] In this context, the term *secure* is akin to "not well understood." Although Java does successfully address memory safety issues, it introduces a large attack surface that is susceptible to a broad range of security issues, including deserialization vulnerabilities.[15] All languages appear to be susceptible to defects, particularly with respect to input validation, which can lead to vulnerabilities.

WHAT'S NEXT FOR C?

A major revision to the C standard, referred to as *C2x*, is currently under development. The goal of the C Standards Committee is not to innovate but rather to standardize on existing practices. Before new features are incorporated into the language and library, there has to be sufficient implementation experience to show that these features are being successfully incorporated by C language programmers and that their benefits outweigh their costs.

New innovations can come from academia and industry. One new proposal for C2x, "Defer Mechanism for C," is a collaborative proposal between several researchers and members of the C Standards Committee.[2] This proposal describes an attempt to adopt the defer statement from the Go programming language to the C programming language. Peter Sewell and Kayvan Memarian have been exploring C semantics and pointer provenance[12] at the University of Cambridge Computer Laboratory. Intel has developed a set of special arbitrary-width integer types spelled as `_ExtInt(N)`, where `N` is an integral constant expression representing the number of bits to be used to represent the type.[3] Aaron Ballman labored tirelessly to introduce attributes to C2x, which are a mechanism by which the developer can attach extra information to language entities with a generalized syntax, instead of introducing new syntactic constructs or keywords for each feature.[1] This work is, in turn, based on implementation experience from the same feature in C++, which itself was based on experience gained using the Microsoft `__declspec` and GNU `__attribute__` features. This flow of innovative proposals from academia and industry shows how the C language continues to evolve and improve in a steady and deliberate fashion.

A major revision to the C Standard, referred to as *C2x*, is currently under development.

C has been successful for a long time, and there is no indication that this will change any time soon. Memory-safe languages such as Go and Rust are gaining in popularity but combined still represent less than 2% of the marketplace. C has a considerable advantage in existing code and compiler support for a wide variety of architectures and embedded platforms. Significant ways remain in which the C language, library, and ecosystem can be improved. There is still work required to more precisely define the behavior for parallel execution in C. The future evolution of the C language requires maintaining a balance between preserving existing code and adopting modern language features that can benefit C language developers. *Effective C*[16] helps newcomers to the language write professional-quality code and circumvent many of the perils of C language programming. This book modernizes one aspect of the C ecosystem and helps propel the entire ecosystem forward. ■

REFERENCES

1. A. Ballman, "Attributes in C," WG14 N2335, ISO/IEC, Geneva, Switzerland, Mar. 2019. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2335.pdf
2. A. Ballman et al., "Defer mechanism for C," WG14 N2542, ISO/IEC, Geneva, Switzerland, July 19, 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2542.pdf
3. M. Blower, T. Hoffner, and E. Keane, "Adding fundamental type for N-bit integers," WG14 N2534, ISO/IEC, Geneva, Switzerland, June 9, 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2534.pdf

4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall, 1978.
5. B. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, Mar. 1988.
6. *Programming Languages --C*, ISO/IEC, 9899:1990.
7. *Programming Languages—C, 2nd ed.*, ISO/IEC, 9899:1999.
8. *Programming Languages—C, 3rd ed.*, ISO/IEC, 9899:2011.
9. *Programming Languages—C, 4th ed.*, ISO/IEC, 9899:2018.
10. F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *The CERT Oracle Secure Coding Standard for Java*, 1st ed. Reading, MA: Addison-Wesley, 2011.
11. F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*,1st ed. Reading, MA: Addison-Wesley, 2013.
12. K Memarian et al., "Exploring C semantics and pointer provenance," in *Proc. ACM Programming Languages (POPL)*, 2019, pp. 1–32. doi: 10.1145/3290380.
13. D. M. Ritchie, "The development of the C language," in *Proc. 2nd ACM SIGPLAN Conf. History Programming Languages (HOPL-II)*, New York, 1993, pp. 201–208. doi: 10.1145/154766.155580.
14. R. C. Seacord, *Secure Coding in C and C++*, 2nd ed. Boston: Addison-Wesley, 2013.
15. R. C. Seacord, "Combating Java deserialization vulnerabilities with Look-Ahead Object Input Streams (LAOIS)," NCC Group, San Francisco, White Paper, June 15, 2017. [Online]. Available: https://www.nccgroup.com/globalassets/our-research/us/whitepapers/2017/june/ncc_group_combating_java_deserialization_vulnerabilities_with_look-ahead_object_input_streams1.pdf
16. R. C. Seacord, *Effective C: An Introduction to Professional C Programming*. San Francisco: No Starch Press, Aug. 2020.

ROBERT C. SEACORD is a technical director at NCC Group. He is on the advisory board for the Linux Foundation and a technical expert for the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language. Contact him at robert.seacord@nccgroup.com.

Erratum

In the article "Inference Acceleration: Adding Brawn to the Brains,"[1] which appeared in the June 2020 issue of *Computer*, an incorrect URL was given for reference [1] due to a production error. The correct reference is

[1] K. Johnson, "OpenAI releases curtailed version of GPT-2 language model," Venture Beat, Aug. 20, 2019. [Online]. Available: https://venturebeat.com/2019/08/20/openai-releases-curtailed-version-of-gpt-2-language-model/

REFERENCE

1. M. Campbell, "Inference acceleration: Adding brawn to the brains," *Computer*, vol. 53, no. 6, pp. 73–76, 2020. doi: 10.1109/MC.2020.2984870.

Digital Object Identifier 10.1109/MC.2020.3026508
Date of current version: 21 October 2020