Analysis of Undefined Behavior in C: The Pedagogical Issues of Using Multiple Pre- and Post-Increment Operators Across Compilers

Author: Khalil Abd AlMageed Khalil Mohammed

Abstract

This paper provides an in-depth analysis of why expressions in C that combine multiple pre- and post-increment operators on a single variable exhibit unpredictable and compiler-dependent behavior. The analysis shows that this phenomenon is a direct consequence of "unsequenced modifications," a specific form of undefined behavior sanctioned by the C language standard. The historical rationale behind this design choice, which was intended to enable aggressive compiler optimizations for performance on early machine architectures, is explored. It is argued that this historical relic has become a significant pedagogical and software engineering challenge. Through case studies of divergent compiler outcomes, the practical risks of relying on such expressions are illustrated, including silent data corruption and non-portability. This paper concludes that these expressions, while a fascinating study in language design and compiler theory, are pedagogically unsound and should be actively discouraged in any foundational C curriculum, advocating instead for teaching principles of clarity, defensive coding, and a deep understanding of the language's formal rules.

1. Introduction

1.1. Context and Background

The C programming language was devised in the early 1970s at AT&T Bell Laboratories as a systems implementation language for the nascent Unix operating system.[1] Its design was a pragmatic response to the limitations of earlier languages like B and BCPL, aiming to create a general-purpose language that was both portable and efficient enough for low-level tasks such as operating system development.[1] C achieved this by providing a small, concise syntax and a close relationship with the underlying hardware, essentially acting as a portable assembly language.[1]

This design philosophy, prioritizing performance and low-level control, has been a defining characteristic of the language throughout its evolution and standardization.[4] While this approach has made C an enduring and powerful tool for a wide range of applications—from embedded systems to high-performance computing—it also introduced certain complexities and ambiguities that have become a subject of ongoing debate in the software development community.[1] The central focus of this analysis is one such ambiguity: the use of multiple increment operators within a single expression.

1.2. Problem Statement

Individual increment operators, pre-increment (++x) and post-increment (x++), are fundamental and well-defined elements of the C language. The timing of the side effect—the increment of the variable—is what distinguishes them, and this behavior is predictable when they are used in isolation.[5] However, when multiple instances of these operators are applied to the same variable within a single expression (e.g.,

y = i++ + ++i;), the results become inconsistent and unpredictable.[7] The same code, compiled with different compilers or even with different optimization settings on the same compiler, can produce divergent outcomes, a phenomenon that runs counter to the expectation of deterministic computation.

1.3. Thesis and Scope

The central thesis of this paper is that this apparent inconsistency is not a flaw in compiler implementations but a direct consequence of a deliberate design choice in the C standard known as "undefined behavior" (UB). Specifically, the expressions in question fall under the category of "unsequenced modifications." This paper will argue that, despite the historical justification for this design choice—to enable aggressive compiler optimizations—this ambiguity represents a significant pedagogical challenge and a source of subtle, hard-to-find bugs. The analysis will proceed by first outlining the technical mechanics of the operators, followed by a formal explanation of UB and its relationship to sequence points. Subsequently, a discussion of compiler-specific outcomes and the historical rationale for this behavior will be presented. The paper will conclude by outlining the pedagogical pitfalls these expressions present and providing recommendations for a more robust and effective approach to teaching C.

2. The Increment Operators: Fundamental Mechanics

To understand the core issue, it is first necessary to establish the precise mechanics of the pre- and post-increment operators when they are used in isolation, a context in which their behavior is fully defined and predictable.

2.1. Pre-increment (++x)

The pre-increment operator, denoted by two plus signs before the operand, first increments the value of the variable and then uses the new, updated value in the expression. The variable is modified in place.[5] For example, in the expression

int b = ++a;, if a starts with the value 5, it is first incremented to 6, and then the value 6 is assigned to b. Both a and b will end with the value 6.[5] This operation is generally considered to be slightly more efficient than post-increment because it avoids the need to create a temporary copy of the variable's original value.[5]

2.2. Post-increment (x++)

The post-increment operator, with the plus signs following the operand, operates differently. It first uses the current value of the variable in the expression, and only after the expression is evaluated does it increment the variable.[5] In the expression int b = a++;, if a starts with the value 5, the original value 5 is assigned to b. After the assignment is complete, a is incremented to 6. Consequently, b will have the value 5, while a will have the value 6.[5] This operation necessitates the creation of a temporary copy of the variable's original value to be used in the expression, which can result in a minor performance penalty compared to pre-increment, particularly for complex data types.[5]

2.3. A Comparison of Distinct Behavior

The distinction between these two operators is clear and deterministic when they are used in isolation. The following table summarizes their key differences.

Feature	Pre-increment (++i)	Post-increment (i++)
Increment Timing	Before the expression is evaluated	After the expression is evaluated
Return Value	The updated value	The original value
Syntax	++i	i++
Performance	Generally slightly faster	Generally slightly slower
Usage	Preferred when the updated value is needed immediately and for efficiency	Used when the old value is required for the expression

This table illustrates the well-defined, predictable behavior of the operators when they are not combined in ways that violate the language standard.[5] This foundational understanding is critical for appreciating why their combination creates a problem that transcends simple operator precedence.

3. The Formal Definition of Undefined Behavior in C

The unpredictability of expressions with multiple increment operators is not a bug or an oversight; it is a direct consequence of a formal concept within the C standard.

3.1. Defining the Spectrum of Behavior

In the C programming community, undefined behavior is a widely discussed topic.[9] It is defined as behavior for which the C standard imposes no requirements on how the code behaves.[9] When a program invokes UB, the compiler is free to do anything it chooses, ranging from generating a predictable result to causing a program crash, corrupting data, or even silently producing incorrect output.[9] This extreme latitude is often humorously referred to as "nasal demons" in the programming community.[9] It is important to distinguish UB from other categories of behavior defined by the standard [9]:

- **Unspecified behavior:** The standard specifies a set of valid outcomes, but the choice among them is not defined. The compiler must choose one of the specified behaviors.
- **Implementation-defined behavior:** The standard requires that the behavior be documented by the compiler vendor. While the behavior may vary across platforms, it is predictable and documented for a given implementation.

The expressions in question fall firmly into the UB category, a formal condition that the program must not meet.[9]

3.2. The Role of Sequence Points

The formal mechanism that governs the determinism of expressions in C is the "sequence point".[14] A sequence point is a specific point in a program's execution where all side effects from previous evaluations are guaranteed to have been performed, and no side effects from subsequent evaluations have yet taken place.[14] The C standard dictates a critical rule related to sequence points: between two sequence points, an object's stored value can be modified at most once by the evaluation of an expression.[14] Furthermore, its prior value can only be accessed to determine the value to be stored.[14]

Examples of sequence points include the semicolon at the end of a full expression, the end of the first operand of the logical AND (&&), logical OR (||), and comma operators, and before a function is entered in a function call.[14]

3.3. Unsequenced Modifications as a Source of UB

The unsequenced modification of a variable is the precise reason why expressions like i++ + ++i result in undefined behavior. The + operator, unlike the comma operator or the logical operators, does not introduce a sequence point.[8] In an expression such as

int j = i++ + ++i;, the modifications to i from both the post-increment (i++) and the pre-increment (++i) are unsequenced with respect to each other.[7] This means the compiler is not required to perform the two side effects in any particular order.[7]

The problem is not one of operator precedence, which only governs how an expression is parsed and grouped, not the order in which its sub-expressions are evaluated.[7] For example, in the expression

int j = a + b * c;, precedence dictates that the multiplication b * c is performed before the addition a + (b*c). However, the order in which a, b, and c are evaluated is unspecified.[16] Similarly, in

i++ + ++i, the compiler is free to evaluate the two operands of the + operator in any order, or even interleave their operations, as long as it adheres to the "as-if" rule for a strictly conforming program.[4] Because the variable

i is modified twice without an intervening sequence point, the behavior is formally undefined by the C standard.[7] This is a specific instance of a broader class of UB cases, such as signed integer overflow, division by zero, or accessing an array out of bounds, all of which stem from the standard's lack of requirements for certain erroneous or non-portable constructs.[9] The core reason for this behavior is that the language provides no means for the programmer to guarantee a specific order of side effects.

4. Compiler Interpretations and Divergent Outcomes

The C standard's decision to leave certain behaviors undefined is not arbitrary. It is a fundamental element of the language's design that provides a vast amount of freedom to compiler developers.

4.1. The Freedom of Optimization

The allowance for undefined behavior gives compilers a powerful tool for optimization. A compiler can assume that a conforming program will never encounter UB and, based on this assumption, can make aggressive transformations to the code.[9] This principle, often called the "as-if" rule, allows the compiler to reorder, simplify, or even remove code entirely, as long as the user-visible side effects of a program that

doesn't contain UB remain the same.[4] For instance, a compiler can remove a conditional statement if a previous operation, by not invoking UB, guarantees that the condition will always be false.[9]

4.2. Case Studies: GCC vs. Clang

The practical consequence of this is that different compilers, pursuing different optimization strategies, will handle UB in their own unique ways, leading to divergent outcomes. For example, consider a simple division-by-zero expression: int ub = argc / 0;.[18] When compiled with different optimization levels, GCC might translate this into an

ud2 instruction, which causes the program to crash, making the UB immediately obvious to the developer.[18] Clang, on the other hand, might simply translate the code into a

ret instruction, effectively removing the problematic code entirely and allowing the program to continue running with an unpredictable return value.[18] This demonstrates that both outcomes are valid under the C standard, yet one crashes the program while the other does not.

A similar divergence occurs with the increment operators. The following table illustrates the unpredictable results from a commonly cited example.

Code Example	int i = 1; printf("%d", ++i + i++);
Compiler	**Output**
Clang	4
GCC (G++)	5

Resulting Behavior	Unspecified and non-portable

The table clearly shows that the exact same line of code, executed on two different compilers, produces different results.[7] One compiler may evaluate

++i first, resulting in a value of 2, and then i++, also using a value of 2, leading to a sum of 4. Another compiler may evaluate ++i first, using a value of 2, and then apply its side effect. Then, it may evaluate i++, which uses the new value of 2 to return the original value of i, which is 2. This would be the logic 2 + 2 = 4 with i ending up as 3. However, the behavior on the second compiler leads to a different result, which could be 5, as some implementations may apply the side effect of i++ before the value of ++i is evaluated, leading to 2+3 = 5 with i ending up as 3.[7] This non-deterministic outcome is a direct consequence of the C standard's refusal to specify the order of operations for unsequenced side effects.[7] The divergent behavior reveals fundamental philosophical differences between compiler teams: one may prefer to make UB obvious by crashing, while another may opt for a more lenient approach that removes the offending code.[18]

5. The Historical Rationale for C's Design

To fully grasp why these issues exist, it is essential to consider the historical context of the C language. The design decisions made in the 1970s, which appear problematic today, were rooted in the engineering constraints of that era.

5.1. Performance as a Primary Directive

The C programming language was created at a time when computing resources were extremely limited.[1] Dennis Ritchie and Ken Thompson's goal was to build a language that could effectively replace assembly language for systems programming while being portable across different machine architectures.[2] Every byte of memory and every CPU cycle mattered. The design of C was thus a series of deliberate trade-offs, with a strong bias toward performance and simplicity of compilation.[9]

5.2. Compiler Portability over Strict Semantics

In the early versions of C, the primary advantage of undefined behavior was its role in producing efficient compilers for a wide variety of machines.[9] Different hardware architectures had varying conventions for things like function calls and argument passing, such as pushing operands to the stack in forward or reverse order.[4] To avoid forcing compilers to generate "redundant" copies of variables or add complex, inefficient code to enforce a uniform, strict order of evaluation, the language standard simply left the behavior of expressions with unsequenced side effects undefined. This pragmatic approach simplified the task of compiler writers and ensured that the generated code would be as lean and fast as possible for any given machine.[4]

5.3. A Historical Relic

While this historical rationale was valid in the 1970s and 1980s, modern computing environments have fundamentally changed.[4] Today's compilers are far more sophisticated and can apply advanced optimizations under more strictly defined rules without a significant performance penalty.[4] A document from the C standards committee acknowledges that the historical performance concerns related to compiler limitations have "evaporated with the passage of time".[4] The performance benefit gained from leaving these expressions undefined is now considered minimal, with the trade-off being increased programmer confusion and the potential for non-portable code and security vulnerabilities.[4] What was once a rational design decision is now, in many ways, a relic of a past era of computing.

6. Pedagogical Pitfalls and the Case for Avoidance

Beyond the technical and historical aspects, expressions with multiple increment operators present significant challenges in a pedagogical context.

6.1. Misleading Intuitions

Beginners often confuse these expressions with deterministic behavior, assuming that operator precedence or associativity will dictate the outcome.[7] When a student runs a program with

y = i++ + ++i; on their personal machine and observes a specific output, they may incorrectly conclude that the language's behavior is predictable and that they have "solved" the problem.[23] This reliance on "what works on my machine" is a dangerous habit that leads to non-portable and fragile code, as the behavior may change with a different compiler or optimization level, or even with a different version of the same compiler.[23] This creates a false sense of security and obscures the true nature of UB.

6.2. Masking Core Concepts

A more profound issue is that teaching with these expressions masks the fundamental principles of the C language. Students and even experienced programmers often conflate operator precedence and associativity, which are syntactic rules for parsing an expression, with the order of evaluation, which is a run-time semantic rule.[16] The

+ operator has left-to-right associativity, which only matters when multiple operators of the same precedence are used.[16] It says nothing about the order in which the operands are evaluated at run-time.[16] By using these complex expressions as a test of knowledge, instructors inadvertently reinforce this misconception rather than clarifying the core principle of unsequenced side effects.[16] The real lesson to be taught is not a specific, obscure rule about a single expression, but the overarching importance of sequence points and the distinction between value-based operations and side effects.

6.3. Fostering Poor Coding Habits

Relying on complex, side-effect-laden expressions promotes a coding style that is antithetical to the principles of clarity and maintainability.[25] Such code is difficult to read, debug, and reason about, even for experienced developers, and can lead to silent data corruption or security vulnerabilities that are hard to detect.[23] A much safer and more effective approach is to break down complex expressions into multiple, simpler, and more readable statements. This principle of "one side effect per statement" ensures that side effects are explicitly sequenced by the semicolon, eliminating any ambiguity and making the code more portable and robust.[25]

7. Recommendations for Safer and More Effective Teaching

To address the pedagogical challenges posed by undefined behavior, it is imperative to shift the focus of C instruction from memorizing obscure rules to adopting a safer, more principled coding methodology.

7.1. Focus on Clarity and Predictability

The most direct solution is to teach students to write clear and predictable code from the outset. Rather than relying on terse, complex expressions that invite ambiguity, a better pedagogical approach is to advocate for a single assignment per line and a clear separation of concerns. This makes the code easier to read and debug and ensures that side effects are explicitly ordered by sequence points, eliminating the possibility of unsequenced modifications.[25]

7.2. Teach the 'Why' of Undefined Behavior

Instead of merely telling students to avoid certain constructs, educators should explain the underlying reasons for undefined behavior. This includes a discussion of its historical context, its role in compiler optimization, and the practical implications for software engineering and security.[9] By understanding that UB is a deliberate design choice with real-world consequences, students can develop a more mature and responsible approach to programming. The

++x and x++ problem can be used as a case study to illustrate how a historical design decision for performance has evolved into a modern source of confusion and bugs.

7.3. Introduce Defensive Coding Practices

A modern C curriculum should integrate the use of tools and best practices that help developers identify and prevent UB.[13] This includes:

- **Compiler Warnings:** Encouraging the use of flags like -Wall, -Wextra, and -Wpedantic to catch potential issues early.[24]

- **Static Analysis Tools:** Introducing tools such as Clang Static Analyzer, which can identify issues like uninitialized variables or null pointer dereferences without running the program.[13]
- **Sanitizers:** Discussing the use of runtime tools like AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan), which can detect UB during execution.[9]

By fostering a mindset of secure and defensive programming, instructors can equip students to write robust, portable, and reliable code that will stand the test of time.[29]

8. Conclusion

The analysis demonstrates that expressions combining multiple pre- and post-increment operators on the same variable within a single statement are a specific and well-defined instance of undefined behavior. This unpredictability is not a compiler error but a consequence of a design choice rooted in C's historical origins, where a permissive standard was prioritized to simplify compiler development for a wide range of architectures.

While these expressions are a fascinating subject for theoretical study in language design and compiler optimization, they are fundamentally at odds with the principles of robust, predictable, and portable software development. Their use in a foundational curriculum creates misleading intuitions, obscures core language concepts, and fosters fragile coding habits. Therefore, they should be actively avoided and, more importantly, explicitly discouraged in any pedagogical context. A more effective approach to teaching C involves a clear focus on the distinction between value and side effect, an understanding of sequence points, and the adoption of modern defensive programming tools and practices.

References

1. Unstop. (n.d.). *History of C Language | Timeline Infographic, Working & More.* Retrieved August 19, 2025, from https://unstop.com/blog/history-of-c-language

2. Nokia. (n.d.). *The Development of the C Language*. Retrieved August 19, 2025, from https://www.nokia.com/bell-labs/about/dennis-m-ritchie/chist.html

3. Wikipedia. (n.d.). *The C Programming Language*. Retrieved August 19, 2025, from https://en.wikipedia.org/wiki/The_C_Programming_Language

4. Open-Std.org. (n.d.). *explicit-order*. Retrieved August 19, 2025, from https://www9.open-std.org/JTC1/SC22/WG14/www/docs/n3203.htm

5. upGrad. (n.d.). *Pre-increment and Post-increment in C*. Retrieved August 19, 2025, from https://www.upgrad.com/tutorials/software-engineering/c-tutorial/pre-increment-and-post-increment-in-c/

6. Quora. (n.d.). *What is the difference between pre and post increment?*. Retrieved August 19, 2025, from https://www.quora.com/What-is-the-difference-between-pre-and-post-increment

7. Stack Overflow. (n.d.). *Undefined behavior in C/C++: i++ + ++i vs ++i + i++ ...*. Retrieved August 19, 2025, from https://stackoverflow.com/questions/39900469/undefined-behavior-in-c-c-i-i-vs-i-i

8. jambit GmbH. (n.d.). *Order of evaluation in C, Objective C and C++*. Retrieved August 19, 2025, from https://www.jambit.com/en/latest-info/toilet-papers/order-of-evaluation-in-c/

9. Wikipedia. (n.d.). *Undefined behavior*. Retrieved August 19, 2025, from https://en.wikipedia.org/wiki/Undefined_behavior

10. Learn C++. (n.d.). *6.4 — Increment/decrement operators, and side effects*. Retrieved August 19, 2025, from https://www.learncpp.com/cpp-tutorial/increment-decrement-operators-and-side-effects/

11. Solid Sands. (n.d.). *A Study on Undefined Behavior in C*. Retrieved August 19, 2025, from https://solidsands.com/wp-content/uploads/Master_Thesis_Vasileios_GemistosFinal.pdf

12. Stack Overflow. (n.d.). *Undefined, unspecified and implementation-defined behavior*. Retrieved August 19, 2025, from https://stackoverflow.com/questions/2397984/undefined-unspecified-and-implementation-defined-behavior

13. Medium. (n.d.). *Understanding Undefined and Implementation-Defined Behavior in C Programming*. Retrieved August 19, 2025, from https://medium.com/@python-javascript-php-html-css/understanding-undefined-and-implementation-defined-behavior-in-c-programming-44b44c3f204e

14. Wikipedia. (n.d.). *Sequence point*. Retrieved August 19, 2025, from https://en.wikipedia.org/wiki/Sequence_point

15. GNU. (n.d.). *Sequence Points (GNU C Language Manual)*. Retrieved August 19, 2025, from https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Sequence-Points.html

16. Reddit. (n.d.). *Precedence of logical operators and parentheses : r/C_Programming*. Retrieved August 19, 2025, from https://www.reddit.com/r/C_Programming/comments/11jywku/precedence_of_logical_operators_and_parantheses/

17. Clang. (n.d.). *Clang Compiler User's Manual — Clang 22.0.0git documentation*. Retrieved August 19, 2025, from https://clang.llvm.org/docs/UsersManual.html

18. Diekmann, L. (2024). *Statically Known Undefined Behaviour*. Retrieved August 19, 2025, from https://diekmann.uk/blog/2024-06-25-statically-known-undefined-behaviour.html

19. Wikipedia. (n.d.). *Undefined behavior*. Retrieved August 19, 2025, from https://en.wikipedia.org/wiki/Undefined_behavior#:~:text=In%20the%20early%20versions%20of,the%20side%20effects%20to%20match

20. Stack Overflow. (n.d.). *Incrementing in C++ — When to use x++ or ++x?*. Retrieved August 19, 2025, from https://stackoverflow.com/questions/1812990/incrementing-in-c-when-to-use-x-or-x

21. GeeksforGeeks. (n.d.). *Pre and Post Increment Operator in C/C++*. Retrieved August 19, 2025, from https://www.geeksforgeeks.org/cpp/pre-increment-and-post-increment-in-c/

22. MC++ BLOG. (n.d.). *C++ Core Guidelines: Rules for Expressions*. Retrieved August 19, 2025, from https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-expressions/

23. Project Nayuki. (n.d.). *Undefined behavior in C and C++ programs*. Retrieved August 19, 2025, from https://www.nayuki.io/page/undefined-behavior-in-c-and-cplusplus-programs

24. Medium. (n.d.). *What Are the Most Common Mistakes Beginners Make in C ...*. Retrieved August 19, 2025, from https://medium.com/@rukminisantoshi/what-are-the-most-common-mistakes-beginners-make-in-c-programming-5e97112e1c13

25. Stack Overflow. (n.d.). *A cleaner if statement with multiple comparisons [closed].* Retrieved August 19, 2025, from https://stackoverflow.com/questions/20952325/a-cleaner-if-statement-with-multiple-comparisons

26. Software Engineering Stack Exchange. (n.d.). *When to use else in conditions?.* Retrieved August 19, 2025, from https://softwareengineering.stackexchange.com/questions/364295/when-to-use-else-in-conditions

27. Reddit. (n.d.). *Jens Regehr: A Guide to Undefined Behavior in C and C++.* Retrieved August 19, 2025, from https://www.reddit.com/r/programming/comments/1ja6cza/jens_regehr_a_guide_to_undefined_behavior_in_c/

28. MoldStud. (n.d.). *Overcoming the Frustrations of Undefined Behavior in C ....* Retrieved August 19, 2025, from https://moldstud.com/articles/p-overcoming-the-frustrations-of-undefined-behavior-in-c-development

29. Software Engineering Institute. (n.d.). *Secure Coding in C and C++*. Retrieved August 19, 2025, from https://www.sei.cmu.edu/training/secure-coding-c/

30. Carnegie Mellon University. (n.d.). *CERT Secure Coding in C and C++ Professional Certificate*. Retrieved August 19, 2025, from https://insights.sei.cmu.edu/training/cert-secure-coding-professional-certificate/