Feature Article

Understanding Variable Length Arrays In The C Language

Zhiwei Sun, *Beihang University, Beijing 100191, China*
Wenge Rong, *Beihang University, Beijing 100191, China*
Nanxiang Jiang, *Beihang University, Beijing 100191, China*
Yuxi Bai, *Beihang University, Beijing 100191, China*
Jing Li, *Open University of China, Beijing 100039, China*
Zhang Xiong, *Beihang University, Beijing 100191, China*

Abstract—The C language has been a mainstream programming language and widely used since its inception. Arrays represent one of the most fundamental constructs in C, yet they have long been considered a challenging topic to fully master. It has been observed that significant ambiguities persist in the interpretation of variable length arrays (VLAs). Key challenges include the precise understanding of the VLA definition, particularly the distinction between integer constants and integer constant expressions, as well as the behavioral differences between VLAs and ordinary arrays when used as operands in the `sizeof` *operator. Through practical examples, this paper illustrates the underlying mechanisms of VLAs and underscores the importance of avoiding side effects in their use. We hope that this work will offer clearer guidance for understanding principles related to VLAs and contribute to more accurate and effective usage.*

Since its inception, the C language has remained one of the most popular programming languages. However, during the learning and usage of C, certain concepts prove challenging for programmers of all levels, such as pointers and arrays [1], [2], as both require a comprehensive understanding of memory management mechanisms. Meanwhile, the adoption of Variable Length Arrays (VLAs) in C99 standard [3] has undoubtedly added further complexity to both the understanding and practical application of array programming.

VLAs does offer certain conveniences in programming. By allowing the size of an array to be determined at runtime, the implementation of small, short-lived arrays becomes cleaner and more straightforward. However, it is important to note that the concept of VLAs is highly controversial. They carry inherent risks, such as potential stack overflow when the runtime-specified array size is excessively large. Meanwhile, VLAs are optional in the C language standard, and some compilers (e.g., MSVC) choose not to support them, which introduces portability challenges. Furthermore, C and C++ have different mechanism for VLA, using VLA needs carefulness and this paper mainly focuses on the C language.

Proper understanding of VLA is non-trivial and has practical implications. For example, a case reported in the community involves the Linux kernel, where an array was defined using a `max` macro to compute the maximum of two expressions and set as the array size. Unintentionally, this usage caused the array to be interpreted as a VLA, which subsequently led to unexpected runtime behavior. The issue was ultimately resolved by refactoring the `max` macro [4].

The introduction of VLAs has presented two key challenges for all programmers. The first pertains to their definition, while the second relates to their behavior, particularly that associated with `sizeof`. The distinction between VLAs and ordinary arrays hinges on whether the expression specifying the array size is an **integer constant** or an **integer constant expres-**

XXXX-XXX © 2025 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

sion. If the size is specified by an integer constant or an integer constant expression, the array is classified as an ordinary array; otherwise, it is deemed a VLA. However, few materials provide explicit definitions for these two concepts, nor do they clearly delineate the differences between them.

While clarifying the definition of VLAs is one aspect of the challenge, a more significant issue lies in the additional complexities VLAs introduce regarding certain behaviors associated with the `sizeof` operator. Prior to the introduction of VLAs, the operand of `sizeof` will not be evaluated. For example, given `int i=0`, after `sizeof(i++)` is executed the value of `i` remains 0 rather than becoming `1`, because `i++` is never actually evaluated. Some compilers, such as Clang, will issue warnings in such cases, as any side effects within the operand will be disregarded.

However, the introduction of VLAs complicates this rule: if the operand of `sizeof` has a VLA type, it must be evaluated. As a result the common teaching emphasis (i.e., operand of `sizeof` is not evaluated) now requires careful qualification to reflect the specific case of VLAs. Listing 1 provides an illustrative example where `A[++j][++i]` in the first `sizeof` should not be evaluated but `A[++j]` in the second `sizeof` must be evaluated. That means the value of `j` should be `1`. A survey among C enthusiasts revealed that only 7.8% of responses were able to correctly identify the value of `j` [5]. This suggests a widespread misunderstanding of the evaluation mechanism for VLAs.

Listing 1. Examples of VLAs Misunderstanding

```
int main(int argc, char *argv[])
{
    int n=10;
    int m=10;
    int i=0;
    int j=0;

    double A[n][m];
    sizeof(A[++j][++i]);
    sizeof(A[++j]);
    printf("j=%d\n", j);

    return 0;
}
```

To the best of our knowledge, the literature lacks a rigorously organized and pedagogically coherent treatment of VLAs. Consequently, this paper addresses two foundational questions: 1) How do VLAs fundamentally diverge from ordinary arrays in their definition? 2) In what ways do their behavioral characteristics diverge during program execution? Using the recently released C23 standard [6] as specification, We aim to establish a systematic framework that elucidates the essence of VLAs, offering programmers, students, and educators at all levels an insightful perspective to examine subtle and often overlooked conceptual distinctions, such as those between an integer constant and an integer constant expression. Additionally, we seek to highlight the nuanced behavior of the `sizeof` operator when applied to VLAs, underscoring the importance of avoiding side effects within its operand.

Definition of VLAs

To understand VLAs, we must first clarify what an Array Type is. An array type is a derived type formally defined as `T[N]`, where `T` is referred to as the element type and must be a complete object type. Meanwhile, `N` is an expression that represents the length of the array, i.e., the number of element types.

It is important to note that `N` is optional; in such cases, `T[]` is also a valid array type, but it becomes an incomplete object type. After the introduction of VLAs, the complete array types are categorized into ordinary array types and variable length array types (VLA types). The criteria for distinguishing them are as follows: for an array type `T[N]`, if either of the following two conditions is met, `T[N]` is considered a VLA type; otherwise, it is an ordinary array type (Section 6.7.7.3 in [6]):

1. `T` itself is a VLA type.
2. `N` is not an integer constant or an integer constant expression.

From the definition, we observe that understanding VLAs involves two critical concepts: **integer constants** and **integer constant expressions**. Here we provide a few examples to establish an intuitive understanding. For instance, `int[10]` is an ordinary array type because `10` is an integer constant and `int` is not a VLA type. Similarly, `int[4+6]` is also an ordinary array type because, although `4+6` is not an integer constant, it is a valid integer constant expression. On the other hand, given `int n=10`, `char[n]` is a variable length array type because, while `char` is not a VLA type, `n` is clearly neither an integer constant nor an integer constant expression.

Furthermore, the element type `T` in `T[N]` can be any complete object type, including another array type. Thus, we can treat `int[10]` and `char[n]` as `T` to derive further array types. If the new derived array type lengths are `m` (given `int m=5`) and `5`, respectively, the resulting array types would be `int[m][10]` and `char[5][n]`. According to the specification, the array length, i.e., `m` and `5`, must be inserted before `[10]`

and `n`. For `int[m][10]`, although the element type `int[10]` is not a VLA type, the length `m` is not an integer constant or integer constant expression. For `char[5][n]`, although the length `5` is an integer constant, the element type `char[n]` is a VLA type. Therefore, both of these array types are VLA types.

In the above discussion, `10` is referred to as an integer constant, `4+6` is considered an integer constant expression, and `n` is neither. But what exactly are the definitions and distinctions between these two concepts? The notion of integer constants is indeed covered in all major textbooks. However, different sources present inconsistencies in their descriptions. For example, while all books unanimously refer to `10` as an integer constant, some textbooks also consider `+10` as an integer costant [7]. However, in the C23 standard `+10` is not an integer constant. It is an integer constant expression.

The notion of integer constant expressions receives limited coverage in popular textbooks. Some sources [8], [9] mentioned the term "constant expression" but did not provide a formal definition or specific syntactic rules governing its formation. After investigating the definition of constant in the literature, we find that most treatments distinguish constants primarily based on whether they have fixed values [10]. Thus, in some sources, given `const int m=10`, `m` is also referred to as a constant [7] because currently its value cannot be modified by the programmer.

Definitions of Integer Constant and Integer Constant Expression

From the discussion above, we understand that integer constants and integer constant expressions are key to understanding VLAs.

An **integer constant** belongs to the the primary expression category in the standard, and can be defined as: `prefix+number_sequence+suffix` (Section 6.4.4.1 in [6]). The optional `prefix` specifies the encoding strategy of the number sequence (e.g., none for decimal, `0x` for hexadecimal). The optional `suffix` determines the numerical type (e.g., `L` for `long`). For example, `10` represents a decimal-encoded constant with a value of `10` and type `int`. `0x10L` represents a hexadecimal-encoded integer constant with a value of `16` and type `long`. From the definition, it is clear that `4+6` and `+10` are not integer constants.

In contrast, an **integer constant expression** refers to an expression will be viewed as integer constant expression if 1) it can be evaluated at translation time, 2) its result has integer type, and 3) its operands satisfy the following constraints[1], namely the operands should be:

1. **Integer constants**, e.g., `10`, `0x10L`.
2. **Character constants**, e.g., `'a'`.
3. **Named constants**, which include: 1) **Enumeration constants**, e.g., `East`, given `enum {East,West,South,North}`. 2) **Predefined constants**, i.e., `true`, `false` and `nullptr`. 3) **Object Identifiers** declared with storage class specifier `constexpr`. e.g., `a` given `constexpr int a=10;`
4. **Compound literal constants**, which is a compound literal with storage class specifier `constexpr`, e.g., `(constexpr int){10}`.
5. **`alignof` expressions**, e.g., `alignof(int)`, given `int n=10`, `alignof(int[n])`.
6. **Cast expressions**, 1) as part of an operand to `sizeof`, `alignof` or `typeof`. 2) otherwise whose operands are floating constants, named constants, and compound literal constants.
7. **`sizeof` expressions**, whose operand cannot have VLA type.

The rules governing these behaviors are formally specified in the C23 standard (Clause 8 in Section 6.6 in [6]). While similar rules existed in earlier standards C89 and C11, C23 introduces new features including `constexpr` and predefined constants.

These rules make clear that an integer constant expression requires more than just having a fixed value; it must also adhere to specific syntactic forms. However, many current study materials still treat expressions with a fixed value as constants, overlooking this syntactic restriction. This long-standing misunderstanding within the programming community has posed substantial challenges to the accurate interpretation of VLAs since their incorporation into the C language standard. A possible reflection is LLMs, since LLMs are typically trained on large-scale internet data. If such data contains inaccuracies and uneven knowledge distributions, the resulting outputs tend to reflect these artifacts [11].

We provide several expressions to some popular LLMs via their web interfaces and ask them to determine whether each expression qualifies as an integer constant or an integer constant expression. Table 1 lists the results, which indicate that all evaluated LLMs fail to conform to the C23 standard when identifying such expressions, suggesting that greater attention

[1]We can use GCC 15.1 to validate all examples in this paper since this version supports C23.

TABLE 1. Examples of Integer Constant Expression Determination and Comparison With LLMs, given `constexpr int a=10`, `int n=10`, `int b[n][10]`, and `const int c=10` (Submitted on 25 of April, 2026)

Expressions	C23 Standard	Deepseek	Qwen	ERNIE	Doubao	GhatGPT	Gemini
`a`	✓	✓	✓	✓	✓	✓	✓
`n`	✗	✗	✗	✗	✗	✗	✗
`(constexpr int){10}`	✓	✓	✓	✗	✓	✓	✗
`'a'`	✓	✓	✓	✓	✓	✓	✓
`+10`	✓	✓	✓	✓	✓	✓	✓
`9+1`	✓	✓	✓	✓	✓	✓	✓
`alignof(int)`	✓	✓	✓	✓	✓	✓	✓
`alignof(int[n])`	✓	✗	✗	✗	✗	✗	✗
`(int)(10.0)`	✓	✓	✓	✓	✓	✓	✓
`(int)(+10.0)`	✗	✓	✓	✓	✓	✓	✓
`sizeof((int)(+10.0))`	✓	✓	✓	✓	✓	✓	✓
`(int)(9.0+1)`	✗	✓	✓	✓	✓	✓	✓
`10>4?1:0`	✓	✓	✓	✓	✓	✓	✓
`10.0>4?1:0`	✗	✓	✓	✓	✓	✓	✓
`sizeof(n)`	✓	✗	✓	✓	✓	✓	✓
`sizeof(int[n])`	✗	✗	✗	✗	✗	✗	✗
`sizeof(b)`	✗	✗	✗	✗	✗	✗	✗
`sizeof(b[0])`	✓	✓	✓	✓	✓	✓	✓
`c`	✗	✗	✗	✗	✗	✓	✗

should be devoted to these two concepts within the community.

For example, both expressions `(int)(10.0)` and `(int)(+10.0)` are cast expressions. According to Rule 6, `(int)(10.0)` qualifies as an integer constant expression, whereas `(int)(+10.0)` does not. This distinction arises because these two cast expressions do not appear as operands of `sizeof`, `alignof`, or `typeof`. Specifically, `10.0` is a floating constant, but `+10.0` is not; instead, it is a unary expression. Consequently, `(int)(+10.0)` is not an integer constant expression. A similar situation occurs with conditional expressions, i.e., `10>4?1:0` and `10.0>4?1:0`. Since `10.0` is a floating constant, `10.0>4?1:0` is not an integer constant expression. All of these expressions have fixed, unchangeable values, which clearly highlights the misunderstanding prevalent within the community.

Meanwhile, LLMs misclassify `alignof(int[n])` as a non-integer constant expression. This reveals a fundamental misunderstanding: `alignof` is always determined at compile time, irrespective of its operand.

VLAs as Operands of `sizeof`

The introduction of VLAs has significantly altered the behavior of `sizeof` operator. The `sizeof(operand)` is a valid unary expression and obtain the size (in bytes) of the operand. The `operand` can be a type name or an expression:

1. **Type names**, which can be further subdivided into Non-VLA type names (e.g., `int`, `char[10]`) and VLA type names (e.g., `int[n]`, given `int n=10`).

2. **Expressions**, which can be further categorized into those with non-VLA type and those with VLA type.

If `operand` is a type name, for VLA types, any subexpressions within the type-name must be evaluated to determine the array length. For example, given `int n=10`, `sizeof(char[++n])` will increase `n` to `11` because `++n` must be evaluated, making it equivalent to `sizeof(char[11])`.

If `operand` is an expression with a variable-length array (VLA) type, it must be evaluated. Otherwise, the `operand` is not evaluated.

In summary, VLA operands (either VLA type-name or VLA expression) will trigger operand evaluation to determine sizes, while Non-VLA operands will not (Section 6.5.3.4 in [6]). The rules are listed in Table 2.

TABLE 2. Operand Evaluation Rules in `sizeof`

Operand	Operand Type	Evaluated?
Type-names	VLA Type	✓
	Non-VLA Type	✗
Expressions	Have VLA type	✓
	Have Non-VLA type	✗

Employing these rules, we revisit the code in Listing 1. For the expression `A[++j][++i]`, which has type `int` (a non-VLA type), the operand of `sizeof` is not evaluated. Consequently, `i` and `j` remain unchanged at 0. In contrast, `A[++j]` has type `int[m]`, which is a VLA type. Therefore, `A[++j]` must be evaluated, incrementing `j` to 1.

We submitted this code to the web-based versions

of several prominent large language models (LLMs), requesting them to provide the value of `j` and explain their reasoning, as shown in Fig. 1. The results, summarized in Table 3, indicate that the majority of LLMs were unable to correctly determine the value of `j` or provide accurate reasoning.

> Suppose you are a C language expert. Given the following code, please give the value of j and provide the reasoning process.
> FIRST: If A[++j][++i] a Variable Length Array or not
> SECOND: If A[++j][++i] should be evaluated as operand of sizeof operator
> THIRD: If A[++j] a Variable Length Array or Not.
> FOURTH: If A[++j] should be evaluated as operand of sizeof operator

FIGURE 1. Prompt for LLMs (Submitted on 25 of April, 2026)

TABLE 3. Reasoning Process Provided by Popular LLMs

LLMs	`j`	1st	2nd	3rd	4th
Groundtruth	1	✗	✗	✓	✓
Deepseek	1	✗	✗	✓	✓
Qwen	0	✗	✓	✗	✓
ERNIE	0	✗	✗	✗	✗
Doubao	1	✗	✗	✓	✓
ChatGPT	0	✗	✗	✗	✗
Gemini	0	✗	✗	✓	✓

Since both `m` and `n` in Listing 1 are of type `int`, LLMs can identify that `A` is a VLA. Upon analyzing their reasoning processes, all LLMs correctly concluded that `A[++j][++j]` is not a VLA and that `j` is not incremented. However, some LLMs appear to incorrectly infer whether the expression `A[++j]` itself constitutes a VLA. While one LLM recognized that `A[++j]` is not a VLA, it still chose to evaluate it as an operand of `sizeof`.

From the above discussion, it is evident that a proper understanding of a VLA as an operand of `sizeof` involves two key points. Firstly, one must correctly identify whether the operand is indeed a VLA. Secondly, it is crucial to understand that when the operand is a VLA, it must be evaluated, meaning any side effects within that operand will occur.

Lesson Learned

In this paper, we firstly introduce the definition of VLAs and their distinctions from ordinary arrays, with a particular emphasis on clarifying the difference between an integer constant and an integer constant expression: a distinction often ambiguously or incorrectly presented in existing educational materials. Some resources conflate these concepts, referring to both `10` and `+10` as integer constants, while others inappropriately use immutability of value as the criterion for classifying an expression as an integer constant. For instance, the variable `m` in `const int m=10` is sometimes mistakenly considered a constant expression. Such conceptual ambiguity poses significant challenges to understanding VLAs, as evidenced by the performance of LLMs, which are trained on publicly available materials. Their responses reflect the current lack of clear and accurate explanations regarding these concepts.

Meanwhile, this paper also discusses the implications of VLAs on the behavior of the `sizeof` operator. Prior to the introduction of VLAs, `sizeof` was evaluated purely at compile time, and its operand was never evaluated. However, with VLAs, operands of VLA type must be evaluated at runtime. This shift can lead to considerable confusion when the operand contains side effects. This scenario deserves explicit clarification in educational resources. We also highlight the importance of cautioning programmers against using expressions with side effects within `sizeof` operands, in order to prevent subtle program errors and ensure robust code.

Conclusion

Variable-length arrays (VLAs) are an important feature in the C language standard, yet properly understanding VLAs remains a challenging task. In this paper, we aim to provide a clearer definition of VLAs by illustrating error-prone concepts such as integer constants and integer constant expressions, explicitly stating that whether a value can change cannot serve as a criterion for distinguishing integer constant expressions. Additionally, in this paper we use practical examples to clarify key considerations when VLAs are used as operands in `sizeof` operator We hope this work will contribute to a clearer understanding of array-related concepts in the C language.

REFERENCES

1. M. Piteira and C. Costa, “Learning computer programming: study of difficulties in learning programming,” in *Proceedings of 2013 International Conference on Information Systems and Design of Communication*, 2013, pp. 75–80.
2. B. D. Boulay, “Some difficulties of learning to program,” *Journal of Educational Computing Research*, vol. 2, no. 1, 1986.
3. ISO, *ISO/IEC 9899:1999 Programming Languages – C*, 2nd ed., 1999, available from American National

Standards Institute (ANSI), and from International Organization for Standardization (ISO).
4. J. Corbet, "Variable-length arrays and the max() mess," https://lwn.net/Articles/749064, 2018, [Online; accessed 2025-08-01].
5. J. Gustedt, "Types and sizes," https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2838.htm, 2021, [Online; accessed 2025-08-01].
6. ISO, *ISO/IEC 9899:2024 Information technology Programming languages C*, 2024.
7. S. Prata, *C Primer Plus*, 6th ed. Pearson Education, 2014, page 64, 147.
8. P. Van der Linden, *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.
9. P. Prinz and T. Crawford, *C in a Nutshell*, 4th ed. O'Reilly Media, Inc., 2005.
10. H. Schildt, *C: The Complete Reference*, 4th ed. McGraw-Hill, 2000, page 37.
11. I. Augenstein, T. Baldwin, M. Cha, T. Chakraborty, G. L. Ciampaglia, D. Corney, R. DiResta, E. Ferrara, S. Hale, A. Halevy *et al.*, "Factuality challenges in the era of large language models and opportunities for fact-checking," *Nature Machine Intelligence*, vol. 6, no. 8, pp. 852–863, 2024.

Zhiwei Sun is currently pursing his PhD in School of Computer Science and Engineering, Beihang University. He received his MSc and BSc from School of Computer Science of Engineering of Beihang University in 2015 and 2012, respectively. His research interests include smart education. Contact him at sunzhiwei@buaa.edu.cn.

Wenge Rong is professor in School of Computer Science and Engineering, Beihang University, China. He received his PhD from University of Reading, UK, in 2010; MSc from Queen Mary College, University of London, UK, in 2003; and BSc from Nanjing University of Science and Technology, China, in 1996. He has many years of working experience as a senior software engineer in numerous research projects and commercial software products. His area of research covers natural language processing and smart education. Contact him at w.rong@buaa.edu.cn.

Nanxiang Jiang is a BSc student in Shen Yuan Honors College, Beihang University. His research interests include computer vision, neural representations, and embodied AI. Contact him at jiangnx@buaa.edu.cn.

Yuxi Bai is a BSc student in Shen Yuan Honors College, Beihang University. His research covers machine learning. Contact him at baiyuxi@buaa.edu.cn.

Jing Li is a research assistant professor at the Engineering Research Center of Integration and Application of Digital Learning Technology, Ministry of Education. She received her Ph.D. from the University of Hong Kong in 2010, and her Mphils and Bachelor's degrees in Public Administration from Peking University in 2003 and 2000 separately. Her main research areas include digital education, lifelong learning policies, and credit banking systems. Contact her at lijing2015@ouchn.edu.cn.

Zhang Xiong is professor in School of Computer Science and Engineering, Beihang University. He has published over 300 referred papers in international journals and conference proceedings and won a National Science and Technology Progress Award. His research interests span from smart education. Contact him at xiongz@buaa.edu.cn.