Learn How to Insert Tab Characters in VBA Using vbTab and Chr(9)

Authored by

Mohammed looti

November 10, 2025

In VBA (Visual Basic for Applications), meticulous text formatting is fundamental for generating professional, highly readable outputs, whether these are internal log files, summarized reports, or interactive user elements like dialog boxes. A frequent requirement for structuring tabular data within a single string is the capability to insert a tab character, which guarantees uniform horizontal spacing and alignment. Although inserting this control character seems like a trivial task, VBA provides developers with two distinct, yet functionally identical, approaches for specifying this critical element within a string expression.

A comprehensive understanding of these two methods--the mnemonic, built-in constant and the numeric character code equivalent--is vital, as it allows developers to select the technique that best aligns with their project's standards for clarity and maintainability. Both options represent foundational techniques utilized extensively across diverse VBA applications, ranging from basic string concatenation for temporary variables to complex, high-volume data parsing operations. Mastering the insertion of control characters is essential for any professional seeking to fully automate data processing and display tasks within the Microsoft Office ecosystem.

The Dual Approach to Tab Insertion: Constants vs. Codes

When executing string manipulation routines in VBA, developers have two proven, reliable mechanisms for inserting the horizontal tab character. These mechanisms function as control characters, directing the runtime environment to inject the necessary spacing equivalent to pressing the physical Tab key. The decision between the two methods generally hinges on factors such as established code readability standards and the developer's familiarity with character set encoding systems, specifically ASCII.

The two primary and interchangeable methods for representing the tab character are:

vbTab: The predefined, self-documenting constant.
Chr(9): The function call utilizing the numeric ASCII code.

The constant **vbTab** is a standard component of the core VBA library and is widely regarded as the preferred approach due to its immediate clarity. When a programmer encounters **vbTab** within the source code, its purpose--inserting a tab character--is unequivocally understood without the need for external reference. This practice significantly enhances code maintenance and facilitates collaborative development efforts, as team members do not need to memorize or look up specific character codes. Essentially, **vbTab** serves as a robust, symbolic representation of the tab control character.

In contrast, the method **Chr(9)** relies directly on the underlying character set architecture. The **Chr()** function is designed to return the character corresponding to a specified numeric ASCII code. Given that the Horizontal Tab control character is mapped to ASCII value 9, the expression

Chr(9) yields an output precisely identical to **vbTab**. While this method is less intuitive for those unfamiliar with ASCII encoding, it offers superior flexibility when dealing with a broader range of control characters that may not have a dedicated, named VBA constant.

Practical Implementation: Formatting Data with vbTab

To practically demonstrate the utility of these constants, we will examine a typical scenario involving data extraction and formatted presentation. A common task in automation is retrieving structured information from a worksheet and consolidating it into a single, organized output element, such as a message box (MsgBox). This technique is frequently employed for rapid data validation checks or summary reporting.

Consider a simple Microsoft Excel spreadsheet containing a list of first and last names spread across rows 2 through 11. The primary objective is to iterate through this dataset, combine the first and last names, and ensure they are distinctly separated by a uniform tab stop, with each complete record appearing on a new, separate line. Achieving this structure necessitates the careful use of both the tab character constant and a new line character constant within the string concatenation process.

The initial dataset configuration is as follows:

	A	B	C	D	E
1	**First Name**	**Last Name**			
2	Andy	Miller			
3	Bob	Johnsone			
4	Chad	Stone			
5	Doug	Reed			
6	Eric	Munsen			
7	Frank	Jimmen			
8	Greg	Rhenster			
9	Henry	Fields			
10	Isaac	McDeen			
11	John	Armleder			
12					
13					
14					
15					
16					
17					
18					
19					

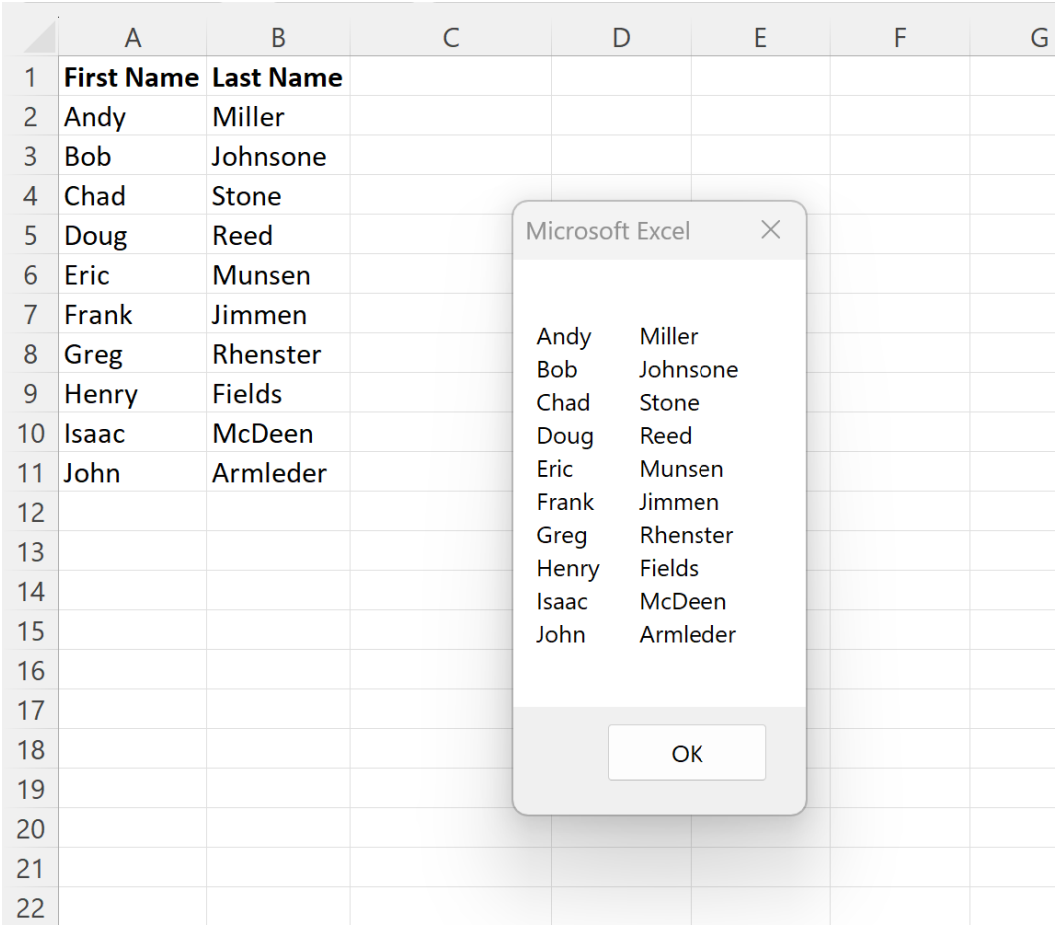
Our macro is designed to loop through the range (A2:B11) and incrementally construct a single string variable named `AllNames`. During each iteration, the first name and last name are read from columns A and B, respectively, and appended to the collective string. Crucially, the **vbTab** constant is embedded directly between the two data fields to introduce the required horizontal separation. Notice how seamlessly the tab constant integrates into the string concatenation operation using the ampersand (&).

The macro below illustrates the use of **vbTab** for precise data formatting:

```
Sub UseTab()

Dim i As Integer
Dim AllNames As String

For i = 2 To 11
AllNames = AllNames & Range("A" & i).Value & vbTab & Range("B" & i).Value & vbNewLine
Next i

MsgBox AllNames

End Sub
```

Upon execution, the MsgBox function presents the resulting accumulated string. The presence of the **vbTab** constant ensures that the last names are aligned neatly, creating a clean, column-like appearance within the dialog box. This alignment is a primary characteristic of the tab character control code and significantly improves the output's readability compared to simply using a standard space character.

The visual outcome clearly demonstrates the effect of the tab character, providing consistent alignment:



It is important to emphasize that **vbTab** manages the horizontal separation of the data elements. The necessary vertical separation--placing each name pair on a distinct line--is managed by the separate constant, **vbNewLine**, which is crucial for multi-line output formatting.

Achieving Equivalence: Utilizing the Chr(9) Function

As previously established, the alternative, yet functionally equivalent, method for inserting a tab character involves employing the **Chr()** function combined with its numeric ASCII code, 9. While **vbTab** is generally preferred for its straightforward readability, using **Chr(9)** demonstrates a deeper comprehension of string encoding mechanisms and produces an identical output result. This approach is often utilized when a developer needs to insert a variety of control characters where dedicated VBA constants might not be readily available or defined.

To confirm this equivalence, we can seamlessly modify the preceding macro by globally replacing all occurrences of **vbTab** with the function call **Chr(9)**. The remaining logic of the macro--including the iterative loop structure, variable initialization, and the use of **vbNewLine**--remains completely unchanged. This intentional substitution highlights the fact that the definition of the tab character is the only element being altered internally.

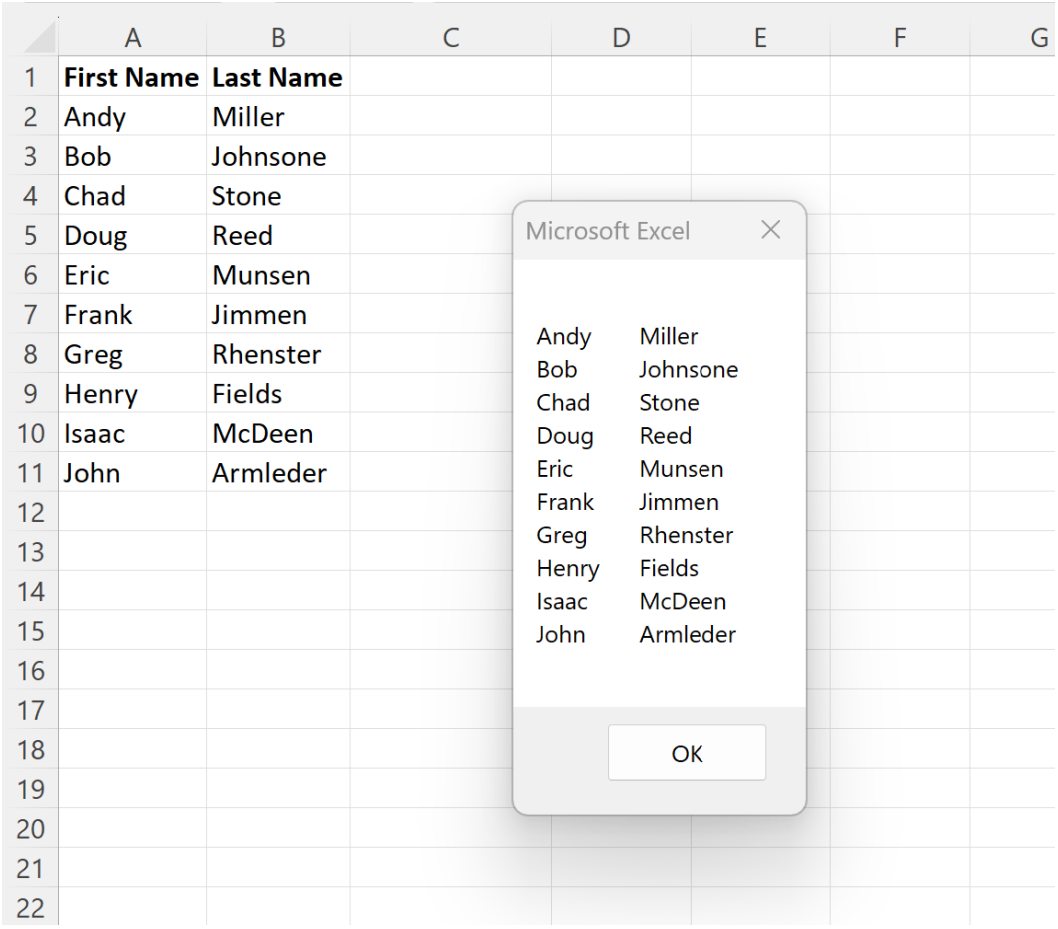
The mechanism of the **Chr()** function is crucial to grasp here. It accepts an integer argument, typically ranging from 0 to 255 for standard ASCII, and returns the corresponding character string defined by that code point. By supplying the value 9, we explicitly instruct the function to retrieve the Horizontal Tab character. This reinforces the technical reality that the constant **vbTab** is essentially a convenient, user-friendly alias for the underlying **Chr(9)** mechanism.

The revised code block below demonstrates the implementation utilizing **Chr(9)**:

```
Sub UseTab()

Dim i As Integer
Dim AllNames As String

For i = 2 To 11
AllNames = AllNames & Range("A" & i).Value & chr(9) & Range("B" & i).Value & vbNewLine
Next i

MsgBox AllNames

End Sub
```

When this modified macro is executed, the resulting output is visually and functionally identical to the one produced using **vbTab**. The alignment, spacing, and overall presentation of the names are perfectly consistent because the underlying character code inserted into the string remains precisely the same. This confirms that, regarding functional behavior and output rendering, **vbTab** and **Chr(9)** are completely interchangeable methods for introducing the tab character into a string variable within VBA projects.

The resulting message box output is displayed one final time to underscore the confirmed equivalence between the two methods:



	A	B	C	D	E	F	G
1	**First Name**	**Last Name**					
2	Andy	Miller					
3	Bob	Johnsone					
4	Chad	Stone					
5	Doug	Reed					
6	Eric	Munsen					
7	Frank	Jimmen					
8	Greg	Rhenster					
9	Henry	Fields					
10	Isaac	McDeen					
11	John	Armleder					
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							

Essential Context: The Role of vbNewLine

While the focus remains on the horizontal tab character, a complete analysis of the demonstrated examples requires a detailed look at the crucial role played by **vbNewLine**. In the context of the data extraction macros, **vbNewLine** is just as indispensable as **vbTab**, as it directly governs the vertical structure and arrangement of the final string output presented in the message box.

The core function of **vbNewLine** is to insert the correct new line character sequence that is appropriate for the current operating system environment. In Windows, this constant resolves dynamically to the carriage return and line feed sequence (which is equivalent to Chr(13) + Chr(10)). Had we carelessly omitted **vbNewLine** from the string concatenation process, all the extracted names would have been concatenated onto a single, excessively long line, separated only by the tab characters, thereby rendering the formatted data virtually unreadable and defeating the purpose of structuring the output.

Reviewing the critical line of concatenation confirms its comprehensive structure: `AllNames = AllNames & Range("A" & i).Value & vbTab & Range("B" & i).Value & vbNewLine`. This precise sequence ensures that during every iteration of the loop, the macro executes four essential

actions sequentially: 1) Appends the first name, 2) Appends the horizontal tab (for column spacing), 3) Appends the last name, and critically, 4) Appends the new line character (to position the cursor for the next record on a fresh line).

It is strongly recommended to use **vbNewLine** over manually inserting the numeric ASCII codes 13 and 10 (e.g., Chr(13) & Chr(10)). This is because **vbNewLine** is a constant specifically engineered to be compatible across potentially varying operating systems or execution environments where the technical definition of a "new line" might differ. This built-in abstraction layer significantly enhances code robustness and portability, which are defining characteristics of clean and maintainable VBA development practices.

Choosing Your Method: Readability vs. Flexibility

Given the confirmed functional identity of **vbTab** and **Chr(9)**, a developer needs to establish clear criteria for selecting the optimal method within a specific project context. This choice usually falls under established code style guidelines rather than performance concerns, as the underlying compiler processes both constants identically during runtime execution, resulting in negligible performance differences.

The consensus among seasoned VBA developers strongly favors defaulting to **vbTab**. This preference is rooted in the fundamental principle of creating self-documenting code. Code is invariably read, reviewed, and debugged far more frequently than it is initially written. Explicit, descriptive constants like **vbTab** immediately inform the reader exactly which character is being inserted, eliminating the need for the reader to possess external specialized knowledge of ASCII codes. This approach dramatically lowers the cognitive burden during debugging, especially in complex macros involving numerous distinct string manipulation operations.

Nevertheless, niche programming scenarios exist where **Chr(9)** might be the more appropriate choice. For instance, if a macro is designed to dynamically determine and handle various control characters based on a user input or an external configuration (e.g., if the character code 9 is stored dynamically in a cell or passed as an argument), leveraging the generic **Chr()** function makes the code inherently more adaptable and flexible. Furthermore, if the existing codebase already extensively utilizes **Chr()** for other non-printing characters (such as Chr(12) for a form feed or Chr(13) for carriage return), maintaining stylistic consistency might dictate using **Chr(9)** for the tab character as well.

In conclusion, for standard, routine string concatenation, data reporting, and internal formatting within the Microsoft Office application suite, **vbTab** represents the standard, modern, and most human-readable choice. Professional developers should establish **vbTab** as their default choice unless a specific, technical requirement explicitly necessitates the use of the numeric ASCII code accessed via the powerful **Chr()** function.

Expanding Your Skills in VBA String Manipulation

Mastering the precise usage of tab and new line characters is merely the initial step toward achieving truly effective VBA string manipulation. For developers committed to expanding their expertise in advanced formatting and control character usage, exploring several related concepts and advanced functions is highly recommended. Subsequent learning steps should focus on understanding how to handle various delimiters, effectively escaping special characters within strings, and efficiently parsing highly complex string datasets.

To continue building on this foundation, consider delving into the following advanced topics related to string formatting in VBA:

A detailed comparison of how and when to use the **vbCrLf** versus **vbNewLine** constants in different environments.
Advanced techniques for utilizing the **Split** and **Join** functions for complex data array processing and reconstruction.
Methods for formatting dates, times, and numerical values within a string variable using the **Format** function.
Working successfully with file Input/Output (I/O) operations, where specific delimiters like tabs are absolutely vital for data integrity and interoperability with external systems.

By establishing a robust foundation in string concatenation and the proper use of character constants, developers gain the ability to create highly robust, professional, and flawlessly formatted applications tailored to sophisticated data reporting and automation requirements within the Excel and broader Office environments.