Converting Strings to Long Integers in VBA: A CLng Tutorial

Authored by

Mohammed looti

November 15, 2025

In complex application development using **VBA**, managing reliable **data type** conversions is essential for computational accuracy. A very common requirement involves transforming a numeric value that is currently stored as a **string**--often imported from external sources or captured via user input--into a usable numeric format. Specifically, developers frequently need to convert these text representations into the **Long** integer type. The dedicated and efficient function for this task is **CLng**, designed to provide accurate conversion for whole numbers within its extensive range.

This authoritative guide offers a detailed breakdown of two critical strategies for deploying the **CLng** function effectively. We will first examine the method of direct, unconditional conversion, which is suitable for validated and guaranteed data sets. Subsequently, we will explore an advanced, conditional conversion technique that incorporates proactive error handling, making it indispensable for processing complex or mixed-data environments. Both approaches are supported by practical **macro** examples, ensuring you can implement precise and stable conversions immediately in your code.

Understanding Data Integrity and Type Conversion in VBA

Data conversion is far more than a technical formality; it is crucial for ensuring computational integrity within any **VBA** application. When data originates from external sources--such as text files, web queries, or cells in **Excel** formatted as text--the underlying value is typically captured and stored as a **text string**. Although these strings may appear visually identical to numbers, the program treats them strictly as sequences of characters, not as mathematical quantities.

Attempting to perform arithmetic operations directly on unconverted **strings** poses a serious risk. While **VBA** sometimes attempts implicit conversion, this behavior is often unreliable, leading to inconsistent results or, more frequently, triggering immediate runtime exceptions like "Type Mismatch." By explicitly converting the **string** representation to a numeric **data type**, developers eliminate this ambiguity and guarantee predictable program execution.

The **Long** **data type** stands as the preferred choice for most whole-number operations in modern **VBA** environments. Its 4-byte structure provides a substantial capacity, allowing it to store integers ranging from approximately -2.1 billion to +2.1 billion. This range is significantly greater than the older 2-byte **Integer** type, dramatically reducing the potential for **Overflow** errors when managing medium to large datasets. Utilizing the **Long** type ensures numeric processing is both efficient and robust, safeguarding the overall stability and performance of your code.

Method 1: Performing Unconditional Conversion with CLng

The direct conversion approach is ideal when you operate under the strict assumption that the source data is clean, standardized, and contains only valid numerical values. In such scenarios--for instance, when importing standardized data from a known source--the fastest and simplest method

is to employ the **CLng** (Convert to Long) function without any preliminary checks.

The syntax is straightforward: `CLng(expression)`. The `expression` argument can be diverse, accepting anything from a variable holding a **string**, a specific cell value, or the result of another numeric calculation. The function attempts to convert the input into a **Long** **data type**, applying banker's rounding (rounding to the nearest even number) for any fractional components. It is crucial to understand the risk here: if the input **string** contains non-numeric characters or if the numerical value exceeds the maximum limit for a **Long**, the code will invariably trigger an immediate runtime error.

The following code snippet illustrates a basic **macro** designed for direct conversion. It loops through a defined **range** of cells, applies the direct conversion, and outputs the resulting **Long** value into the adjacent column. This methodology should only be employed when you have absolute confidence in the integrity and format of the source data.

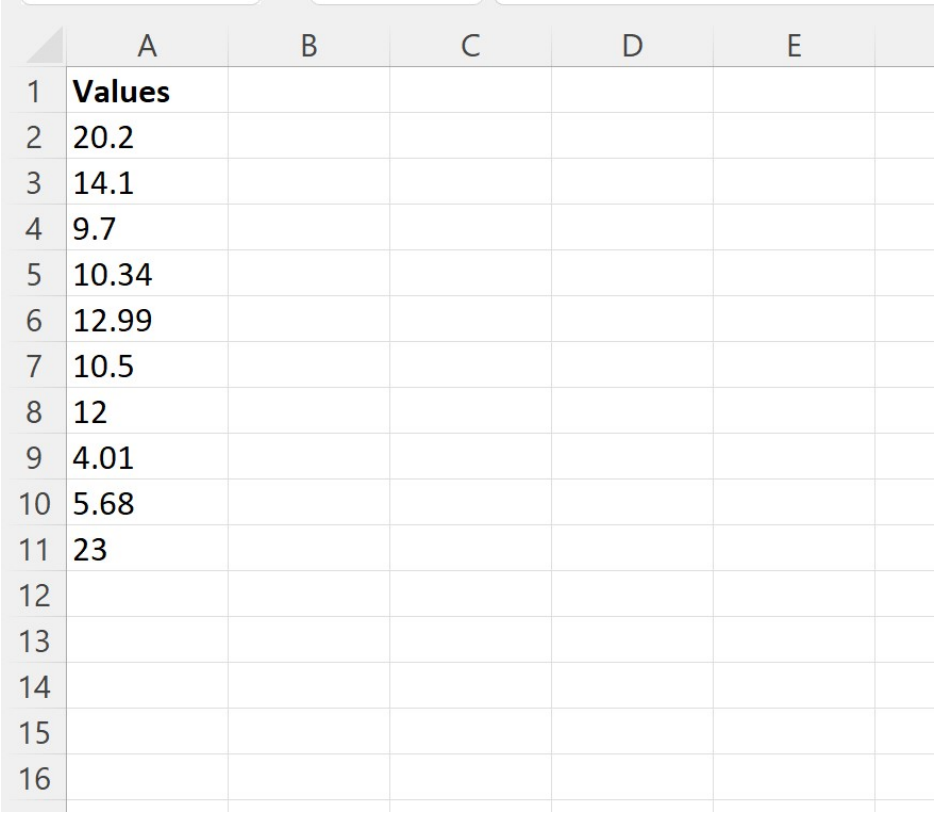
```
Sub ConvertStringToLong()

Dim i As Integer

For i = 2 To 11
Range("B" & i) = CLng(Range("A" & i))
Next i

End Sub
```

In this example, the code iterates through values located in the cell **range** A2:A11. For each cell, it reads the value (stored as a **text string**), utilizes **CLng** to convert it, and finally assigns the resulting numerical **Long** value to the corresponding row in **column B**.

Demonstration 1: Implementing Guaranteed Direct Conversion

Consider a common scenario in data manipulation: you have successfully imported raw data into an **Excel** worksheet, but due to import settings or previous formatting errors, an entire column containing only numbers has been incorrectly stored as text strings. Your immediate operational necessity is to efficiently and quickly transform these textual numbers into a true numeric format suitable for complex mathematical calculations or database integration.

Examine the dataset presented below in column A. Although the values look undeniably numerical, they are currently recognized by VBA as simple text strings:



	A	B	C	D	E
1	**Values**				
2	20.2				
3	14.1				
4	9.7				
5	10.34				
6	12.99				
7	10.5				
8	12				
9	4.01				
10	5.68				
11	23				
12					
13					
14					
15					
16					

Given our confidence that every entry within the source range A2:A11 represents a valid number that falls within the Long integer limits, we can confidently apply the unconditional conversion method detailed previously. We will use the identical **CLng** **macro** to execute the transformation:

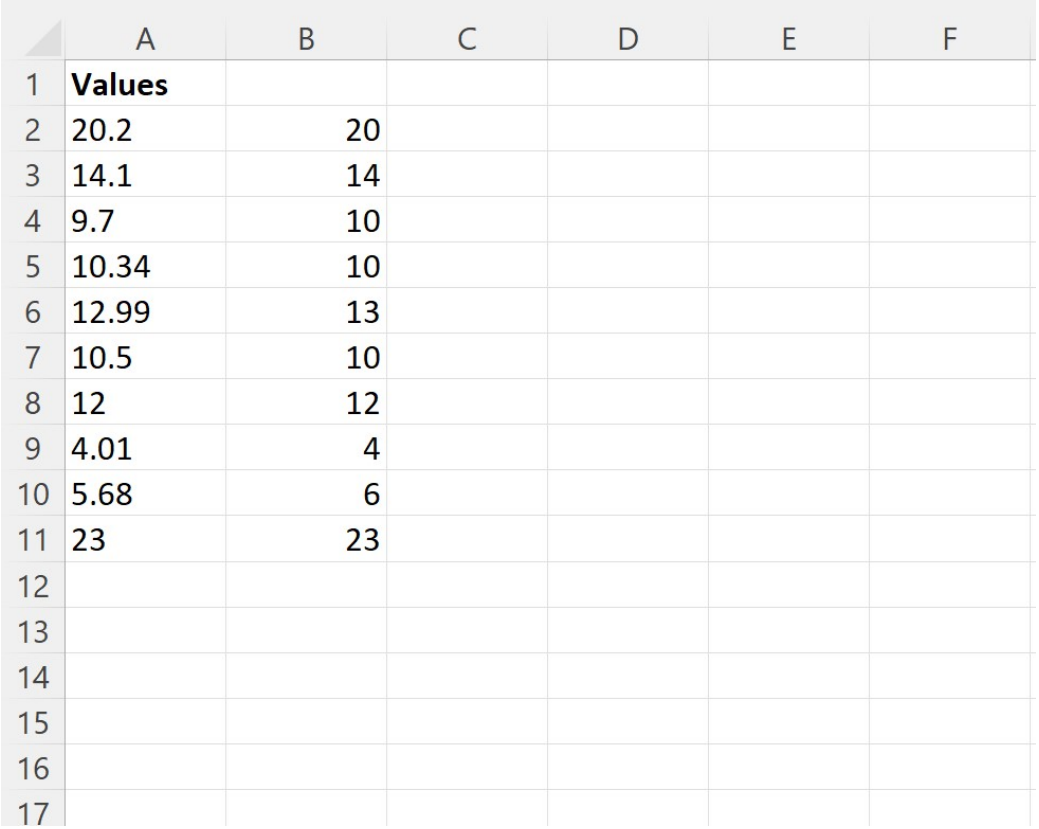
Sub ConvertStringToLong()

Dim i As Integer

For i = 2 To 11
Range("B" & i) = CLng(Range("A" & i))
Next i

End Sub

Upon successful execution of this code, the data appearing in **column B** will consist of successfully converted numeric Long values. These values are now fully accessible for use in any required mathematical formula, pivot table calculation, or integration process without further data type conflicts.



	A	B	C	D	E	F
1	**Values**					
2	20.2	20				
3	14.1	14				
4	9.7	10				
5	10.34	10				
6	12.99	13				
7	10.5	10				
8	12	12				
9	4.01	4				
10	5.68	6				
11	23	23				
12						
13						
14						
15						
16						
17						

Method 2: Conditional Conversion for Maximum Code Robustness

In practical data processing environments, relying solely on Method 1's direct conversion is inherently risky. Real-world datasets are rarely perfectly clean, often containing anomalies such as missing entries, erroneous text inputs, or non-numeric placeholders like "N/A" or "TBD," interspersed among legitimate numerical strings. Applying the **CLng** function directly to these non-numeric items will result in an immediate runtime error, causing the entire **macro** execution to fail prematurely.

To construct truly resilient **VBA** code, we must integrate a proactive pre-check mechanism. The **IsNumeric** function provides this necessary validation layer. **IsNumeric** examines an expression and returns `True` if the input can be successfully interpreted as a number, and `False` if it contains non-numeric characters or formatting issues.

By embedding the **CLng** conversion within an `If...Then...Else` conditional structure controlled by **IsNumeric**, we guarantee that conversion only proceeds when it is safe. If the value is identified as non-numeric, the `Else` block executes, allowing the developer to assign a predetermined default value, such as `0` or `Null`. This strategy ensures data integrity is maintained and prevents disruptive runtime errors, thereby processing the entire dataset gracefully regardless of input quality.

Demonstration 2: Implementing Conditional Conversion on Mixed Data

Imagine working with a data column in **Excel** that presents a typical mix of inputs: legitimate numerical entries stored as text strings, interspersed with clear text entries or error markers. Without conditional checks, this data is certain to cause a conversion failure:

	A	B	C	D	E	F
1	**Values**					
2	20.2					
3	14.1					
4	9.7					
5	10.34					
6	12.99					
7	10.5					
8	Twelve					
9	4.01					
10	5 Dollars					
11	Three					
12						
13						
14						
15						
16						
17						
18						

To reliably process this complex data set, we must utilize the conditional conversion technique. The primary objective is to convert any numerical text strings to the target Long data type, while simultaneously assigning a default value (in this case, `0`) to any entry that fails the necessary numeric validation check. This ensures that the macro completes successfully and provides a clean, homogeneous output.

The following **VBA macro** demonstrates the correct implementation of the IsNumeric function to control the conversion flow:

Sub ConvertStringToLong()

Dim i As Integer

For i = 2 To 11

```
If IsNumeric(Range("A" & i)) Then
Range("B" & i) = CLng(Range("A" & i))
Else
Range("B" & i) = 0
End If
Next i

End Sub
```

When executed, this macro first verifies the numeric status of the cell content in column A. If the check passes, the string is safely converted to a Long integer. For non-numeric entries like "N/A" or "Text," the `Else` block triggers, ensuring a reliable assignment of `0`. The outcome is a clean, error-free result set in column B, as depicted below:

	A	B	C	D	E	F
1	**Values**					
2	20.2	20				
3	14.1	14				
4	9.7	10				
5	10.34	10				
6	12.99	13				
7	10.5	10				
8	Twelve	0				
9	4.01	4				
10	5 Dollars	0				
11	Three	0				
12						
13						
14						
15						
16						
17						

Implementing this conditional logic represents the gold standard for processing external or user-generated data. It completely avoids runtime errors associated with invalid inputs and guarantees a predictable, stable outcome for every operation.

Error Prevention: Best Practices for Robust Conversion

Even when using explicit conversion functions, it is vital to anticipate potential failure points. Functions like CLng can lead to two primary runtime errors if the input data has not been thoroughly validated:

Type Mismatch (Error 13): This is the most frequently encountered conversion error. It occurs when the input string contains characters that VBA cannot interpret as part of a numerical value (e.g., text descriptions, unrecognized currency symbols, or improper formatting).
Overflow (Error 6): This critical error arises when the numerical value represented by the input string exceeds the maximum allowable limit for the target Long data type, which is 2,147,483,647. If your application handles numbers potentially larger than this ceiling, you must target a wider data type instead.

To develop professional and resilient VBA code that withstands unpredictable data inputs, adhere rigorously to these best practices:

Prioritize Validation: As demonstrated in Method 2, always use the **IsNumeric** function immediately before attempting conversion. This proactive check is the single most effective way to prevent Type Mismatch errors and ensures that only data guaranteed to be numerical proceeds to conversion.
Address Range Limitations: If there is any probability that input numbers might exceed the Long data type's capacity, developers should immediately consider using the **CDbl** function (Convert to Double) or the **CCur** function (Convert to Currency) instead of CLng.
Implement Advanced Trapping: For mission-critical applications where data integrity is paramount, integrate sophisticated error handling using **On Error statements**. This mechanism allows your code to specifically intercept runtime errors (like Error 6 or Error 13), log the failure, and continue processing or prompt the user for data correction without abrupt program termination.

Note: For precise details regarding the conversion function's behavior, particularly concerning rounding rules and specific error triggers, always consult the official documentation for the **VBA CLng function**.

Expanding Your VBA Data Manipulation Expertise

Achieving proficiency in Visual Basic for Applications requires a complete command of all standard data types and their related conversion functions. To further enhance your skill set and explore other essential data manipulation tasks critical for professional development, we recommend reviewing the following related tutorials:

How to Convert String to Date in VBA

How to Convert String to Double in VBA
How to Convert String to Boolean in VBA