Data Type Summary (Visual Basic)

Visual Studio 2015

The following table shows the Visual Basic data types, their supporting common language runtime types, their nominal storage allocation, and their value ranges.

Visual Basic type	Common language runtime type structure	Nominal storage allocation	Value range
Boolean	Boolean	Depends on implementing platform	**True** or **False**
Byte	Byte	1 byte	0 through 255 (unsigned)
Char (single character)	Char	2 bytes	0 through 65535 (unsigned)
Date	DateTime	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	Decimal	16 bytes	0 through +/-79,228,162,514,264,337,593,543,950,335 (+/-7.9...E+28) † with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/-0.0000000000000000000000000001 (+/-1E-28) †
Double (double-precision floating-point)	Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 † for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 † for positive values
Integer	Int32	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)
Long (long integer)	Int64	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18 †) (signed)
Object	Object (class)	4 bytes on	Any type can be stored in a variable of type **Object**

		32-bit platform 8 bytes on 64-bit platform	
SByte	SByte	1 byte	-128 through 127 (signed)
Short (short integer)	Int16	2 bytes	-32,768 through 32,767 (signed)
Single (single-precision floating-point)	Single	4 bytes	-3.4028235E+38 through -1.401298E-45 † for negative values; 1.401298E-45 through 3.4028235E+38 † for positive values
String (variable-length)	String (class)	Depends on implementing platform	0 to approximately 2 billion Unicode characters
UInteger	UInt32	4 bytes	0 through 4,294,967,295 (unsigned)
ULong	UInt64	8 bytes	0 through 18,446,744,073,709,551,615 (1.8...E+19 †) (unsigned)
User-Defined (structure)	(inherits from ValueType)	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members
UShort	UInt16	2 bytes	0 through 65,535 (unsigned)

† In *scientific notation*, "E" refers to a power of 10. So 3.56E+2 signifies 3.56×10^2 or 356, and 3.56E-2 signifies $3.56 / 10^2$ or 0.0356.

Note
For strings containing text, use the StrConv function to convert from one text format to another.

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements by using a type character. See Type Characters (Visual Basic).

Memory Consumption

When you declare an elementary data type, it is not safe to assume that its memory consumption is the same as its nominal storage allocation. This is due to the following considerations:

- **Storage Assignment.** The common language runtime can assign storage based on the current characteristics of

the platform on which your application is executing. If memory is nearly full, it might pack your declared elements as closely together as possible. In other cases it might align their memory addresses to natural hardware boundaries to optimize performance.

- **Platform Width.** Storage assignment on a 64-bit platform is different from assignment on a 32-bit platform.

Composite Data Types

The same considerations apply to each member of a composite data type, such as a structure or an array. You cannot rely on simply adding together the nominal storage allocations of the type's members. Furthermore, there are other considerations, such as the following:

- **Overhead.** Some composite types have additional memory requirements. For example, an array uses extra memory for the array itself and also for each dimension. On a 32-bit platform, this overhead is currently 12 bytes plus 8 bytes for each dimension. On a 64-bit platform this requirement is doubled.
- **Storage Layout.** You cannot safely assume that the order of storage in memory is the same as your order of declaration. You cannot even make assumptions about byte alignment, such as a 2-byte or 4-byte boundary. If you are defining a class or structure and you need to control the storage layout of its members, you can apply the StructLayoutAttribute attribute to the class or structure.

Object Overhead

An **Object** referring to any elementary or composite data type uses 4 bytes in addition to the data contained in the data type.

See Also

StrConv
StructLayoutAttribute
Type Conversion Functions (Visual Basic)
Conversion Summary (Visual Basic)
Type Characters (Visual Basic)
Efficient Use of Data Types (Visual Basic)

Data Types of Operator Results (Visual Basic)

Visual Studio 2015

Visual Basic determines the result data type of an operation based on the data types of the operands. In some cases this might be a data type with a greater range than that of either operand.

Data Type Ranges

The ranges of the relevant data types, in order from smallest to largest, are as follows:

- Boolean — two possible values
- SByte, Byte — 256 possible integral values
- Short, UShort — 65,536 (6.5...E+4) possible integral values
- Integer, UInteger — 4,294,967,296 (4.2...E+9) possible integral values
- Long, ULong — 18,446,744,073,709,551,615 (1.8...E+19) possible integral values
- Decimal — 1.5...E+29 possible integral values, maximum range 7.9...E+28 (absolute value)
- Single — maximum range 3.4...E+38 (absolute value)
- Double — maximum range 1.7...E+308 (absolute value)

For more information on Visual Basic data types, see Data Type Summary (Visual Basic).

If an operand evaluates to Nothing, the Visual Basic arithmetic operators treat it as zero.

Decimal Arithmetic

Note that the Decimal data type is neither floating-point nor integer.

If either operand of a **+**, **–**, *****, **/**, or **Mod** operation is **Decimal** and the other is not **Single** or **Double**, Visual Basic widens the other operand to **Decimal**. It performs the operation in **Decimal**, and the result data type is **Decimal**.

Floating-Point Arithmetic

Visual Basic performs most floating-point arithmetic in Double, which is the most efficient data type for such operations. However, if one operand is Single and the other is not **Double**, Visual Basic performs the operation in **Single**. It widens each operand as necessary to the appropriate data type before the operation, and the result has that data type.

/ and ^ Operators

The **/** operator is defined only for the Decimal, Single, and Double data types. Visual Basic widens each operand as necessary to the appropriate data type before the operation, and the result has that data type.

The following table shows the result data types for the **/** operator. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Decimal	**Single**	**Double**	Any integer type
Decimal	Decimal	Single	Double	Decimal
Single	Single	Single	Double	Single
Double	Double	Double	Double	Double
Any integer type	Decimal	Single	Double	Double

The **^** operator is defined only for the **Double** data type. Visual Basic widens each operand as necessary to **Double** before the operation, and the result data type is always **Double**.

Integer Arithmetic

The result data type of an integer operation depends on the data types of the operands. In general, Visual Basic uses the following policies for determining the result data type:

- If both operands of a binary operator have the same data type, the result has that data type. An exception is **Boolean**, which is forced to **Short**.
- If an unsigned operand participates with a signed operand, the result has a signed type with at least as large a range as either operand.
- Otherwise, the result usually has the larger of the two operand data types.

Note that the result data type might not be the same as either operand data type.

Note
The result data type is not always large enough to hold all possible values resulting from the operation. An OverflowException exception can occur if the value is too large for the result data type.

Unary + and – Operators

The following table shows the result data types for the two unary operators, **+** and **–**.

	Boolean	**SByte**	**Byte**	**Short**	**UShort**	**Integer**	**UInteger**	**Long**	**ULong**
Unary **+**	Short	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Unary **–**	Short	SByte	Short	Short	Integer	Integer	Long	Long	Decimal

<< and >> Operators

The following table shows the result data types for the two bit-shift operators, **<<** and **>>**. Visual Basic treats each bit-shift operator as a unary operator on its left operand (the bit pattern to be shifted).

	Boolean	**SByte**	**Byte**	**Short**	**UShort**	**Integer**	**UInteger**	**Long**	**ULong**
<<, **>>**	Short	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong

If the left operand is **Decimal**, **Single**, **Double**, or **String**, Visual Basic attempts to convert it to **Long** before the operation, and the result data type is **Long**. The right operand (the number of bit positions to shift) must be **Integer** or a type that widens to **Integer**.

Binary +, –, *, and Mod Operators

The following table shows the result data types for the binary **+** and **–** operators and the ***** and **Mod** operators. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Boolean	**SByte**	**Byte**	**Short**	**UShort**	**Integer**	**UInteger**	**Long**	**ULong**
Boolean	Short	SByte	Short	Short	Integer	Integer	Long	Long	Decimal
SByte	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Decimal
Byte	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
Short	Short	Short	Short	Short	Integer	Integer	Long	Long	Decimal
UShort	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
Integer	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Decimal

UInteger	Long	Long	UInteger	Long	UInteger	Long	UInteger	Long	ULong
Long	Long	Long	Long	Long	Long	Long	Long	Long	Decimal
ULong	Decimal	Decimal	ULong	Decimal	ULong	Decimal	ULong	Decimal	ULong

\ Operator

The following table shows the result data types for the **** operator. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Boolean	**SByte**	**Byte**	**Short**	**UShort**	**Integer**	**UInteger**	**Long**	**ULong**
Boolean	Short	SByte	Short	Short	Integer	Integer	Long	Long	Long
SByte	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Long
Byte	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
Short	Short	Short	Short	Short	Integer	Integer	Long	Long	Long
UShort	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
Integer	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Long
UInteger	Long	Long	UInteger	Long	UInteger	Long	UInteger	Long	ULong
Long	Long	Long	Long	Long	Long	Long	Long	Long	Long
ULong	Long	Long	ULong	Long	ULong	Long	ULong	Long	ULong

If either operand of the **** operator is Decimal, Single, or Double, Visual Basic attempts to convert it to Long before the operation, and the result data type is **Long**.

Relational and Bitwise Comparisons

The result data type of a relational operation (=, <>, <, >, <=, >=) is always **Boolean**Boolean Data Type (Visual Basic). The same is true for logical operations (**And**, **AndAlso**, **Not**, **Or**, **OrElse**, **Xor**) on **Boolean** operands.

The result data type of a bitwise logical operation depends on the data types of the operands. Note that **AndAlso** and **OrElse** are defined only for **Boolean**, and Visual Basic converts each operand as necessary to **Boolean** before performing the operation.

=, <>, <, >, <=, and >= Operators

If both operands are **Boolean**, Visual Basic considers **True** to be less than **False**. If a numeric type is compared with a **String**, Visual Basic attempts to convert the **String** to **Double** before the operation. A **Char** or **Date** operand can be compared only with another operand of the same data type. The result data type is always **Boolean**.

Bitwise Not Operator

The following table shows the result data types for the bitwise **Not** operator.

	Boolean	**SByte**	**Byte**	**Short**	**UShort**	**Integer**	**UInteger**	**Long**	**ULong**
Not	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong

If the operand is **Decimal**, **Single**, **Double**, or **String**, Visual Basic attempts to convert it to **Long** before the operation, and the result data type is **Long**.

Bitwise And, Or, and Xor Operators

The following table shows the result data types for the bitwise **And**, **Or**, and **Xor** operators. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Boolean	**SByte**	**Byte**	**Short**	**UShort**	**Integer**	**UInteger**	**Long**	**ULong**
Boolean	Boolean	SByte	Short	Short	Integer	Integer	Long	Long	Long
SByte	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Long
Byte	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
Short	Short	Short	Short	Short	Integer	Integer	Long	Long	Long
UShort	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
Integer	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Long
UInteger	Long	Long	UInteger	Long	UInteger	Long	UInteger	Long	ULong
Long	Long	Long	Long	Long	Long	Long	Long	Long	Long
ULong	Long	Long	ULong	Long	ULong	Long	ULong	Long	ULong

If an operand is **Decimal**, **Single**, **Double**, or **String**, Visual Basic attempts to convert it to **Long** before the operation, and the result data type is the same as if that operand had already been **Long**.

Miscellaneous Operators

The **&** operator is defined only for concatenation of **String** operands. Visual Basic converts each operand as necessary to **String** before the operation, and the result data type is always **String**. For the purposes of the **&** operator, all conversions to **String** are considered to be widening, even if **Option Strict** is **On**.

The **Is** and **IsNot** operators require both operands to be of a reference type. The **TypeOf**...**Is** expression requires the first operand to be of a reference type and the second operand to be the name of a data type. In all these cases the result data type is **Boolean**.

The **Like** operator is defined only for pattern matching of **String** operands. Visual Basic attempts to convert each operand as necessary to **String** before the operation. The result data type is always **Boolean**.

See Also

Data Type Summary (Visual Basic)
Operators and Expressions in Visual Basic
Arithmetic Operators in Visual Basic
Comparison Operators in Visual Basic
Operators (Visual Basic)
Operator Precedence in Visual Basic
Operators Listed by Functionality (Visual Basic)
Arithmetic Operators (Visual Basic)
Comparison Operators (Visual Basic)
Option Strict Statement

Conversion Functions (Visual Basic)

Visual Studio 2015

Str

Val

See Also

Type Conversion Functions (Visual Basic)
Converting Data Types

Type Conversion Functions (Visual Basic)

Visual Studio 2015

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Sometimes there is no call to a procedure to accomplish the conversion, which improves performance. Each function coerces an expression to a specific data type.

Syntax

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CSByte(expression)
CShort(expression)
CSng(expression)
CStr(expression)
CUInt(expression)
CULng(expression)
CUShort(expression)
```

Part

expression
Required. Any expression of the source data type.

Return Value Data Type

The function name determines the data type of the value it returns, as shown in the following table.

Function name	Return data type	Range for *expression* argument

CBool	Boolean Data Type (Visual Basic)	Any valid **Char** or **String** or numeric expression.
CByte	Byte Data Type (Visual Basic)	0 through 255 (unsigned); fractional parts are rounded.[1]
CChar	Char Data Type (Visual Basic)	Any valid **Char** or **String** expression; only first character of a **String** is converted; value can be 0 through 65535 (unsigned).
CDate	Date Data Type (Visual Basic)	Any valid representation of a date and time.
CDbl	Double Data Type (Visual Basic)	-1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values.
CDec	Decimal Data Type (Visual Basic)	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001 (+/-1E-28).
CInt	Integer Data Type (Visual Basic)	-2,147,483,648 through 2,147,483,647; fractional parts are rounded.[1]
CLng	Long Data Type (Visual Basic)	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractional parts are rounded.[1]
CObj	Object Data Type	Any valid expression.
CSByte	SByte Data Type (Visual Basic)	-128 through 127; fractional parts are rounded.[1]
CShort	Short Data Type (Visual Basic)	-32,768 through 32,767; fractional parts are rounded.[1]
CSng	Single Data Type (Visual Basic)	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	String Data Type (Visual Basic)	Returns for **CStr** depend on the *expression* argument. See Return Values for the CStr Function (Visual Basic).
CUInt	UInteger Data Type	0 through 4,294,967,295 (unsigned); fractional parts are rounded.[1]

CULng	ULong Data Type (Visual Basic)	0 through 18,446,744,073,709,551,615 (unsigned); fractional parts are rounded.[1]
CUShort	UShort Data Type (Visual Basic)	0 through 65,535 (unsigned); fractional parts are rounded.[1]

[1] Fractional parts can be subject to a special type of rounding called *banker's rounding*. See "Remarks" for more information.

Remarks

As a rule, you should use the Visual Basic type conversion functions in preference to the .NET Framework methods such as `ToString()`, either on the Convert class or on an individual type structure or class. The Visual Basic functions are designed for optimal interaction with Visual Basic code, and they also make your source code shorter and easier to read. In addition, the .NET Framework conversion methods do not always produce the same results as the Visual Basic functions, for example when converting **Boolean** to **Integer**. For more information, see Troubleshooting Data Types (Visual Basic).

Behavior

- **Coercion.** In general, you can use the data type conversion functions to coerce the result of an operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic would normally take place.

- **Failed Conversions.** If the *expression* passed to the function is outside the range of the data type to which it is to be converted, an OverflowException occurs.

- **Fractional Parts.** When you convert a nonintegral value to an integral type, the integer conversion functions (**CByte**, **CInt**, **CLng**, **CSByte**, **CShort**, **CUInt**, **CULng**, and **CUShort**) remove the fractional part and round the value to the closest integer.

  If the fractional part is exactly 0.5, the integer conversion functions round it to the nearest even integer. For example, 0.5 rounds to 0, and 1.5 and 2.5 both round to 2. This is sometimes called *banker's rounding*, and its purpose is to compensate for a bias that could accumulate when adding many such numbers together.

  CInt and **CLng** differ from the Int and Fix functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same data type as you pass in.

- **Date/Time Conversions.** Use the IsDate function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals but not numeric values. To convert a Visual Basic 6.0 **Date** value to a **Date** value in Visual Basic 2005 or later versions, you can use the DateTime.FromOADate method.

- **Neutral Date/Time Values.** The Date Data Type (Visual Basic) always contains both date and time information. For purposes of type conversion, Visual Basic considers 1/1/0001 (January 1 of the year 1) to be a *neutral value* for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date

information is still present in the original **Date** value and can be recovered with functions such as DatePart function.

- **Culture Sensitivity.** The type conversion functions involving strings perform conversions based on the current culture settings for the application. For example, **CDate** recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date might not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

  If you need to convert to or from a string representation of a value in a format other than the one specified by your locale, you cannot use the Visual Basic type conversion functions. To do this, use the `ToString(IFormatProvider)` and `Parse(String, IFormatProvider)` methods of that value's type. For example, use Double.Parse when converting a string to a **Double**, and use Double.ToString when converting a value of type **Double** to a string.

CType Function

The CType Function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface to which there exists a valid conversion.

For a comparison of **CType** with the other type conversion keywords, see DirectCast Operator (Visual Basic) and TryCast Operator (Visual Basic).

CBool Example

The following example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

VB

```
Dim a, b, c As Integer
Dim check As Boolean
a = 5
b = 5
' The following line of code sets check to True.
check = CBool(a = b)
c = 0
' The following line of code sets check to False.
check = CBool(c)
```

CByte Example

The following example uses the **CByte** function to convert an expression to a **Byte**.

VB

```
Dim aDouble As Double
```

```
Dim aByte As Byte
aDouble = 125.5678
' The following line of code sets aByte to 126.
aByte = CByte(aDouble)
```

CChar Example

The following example uses the **CChar** function to convert the first character of a **String** expression to a **Char** type.

VB

```
Dim aString As String
Dim aChar As Char
' CChar converts only the first character of the string.
aString = "BCD"
' The following line of code sets aChar to "B".
aChar = CChar(aString)
```

The input argument to **CChar** must be of data type **Char** or **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. The following example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses the InputBox function to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

VB

```
Dim someDigits As String
Dim codePoint As Integer
Dim thisChar As Char
someDigits = InputBox("Enter code point of character:")
codePoint = CInt(someDigits)
' The following line of code sets thisChar to the Char value of codePoint.
thisChar = ChrW(codePoint)
```

CDate Example

The following example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

VB

```
Dim aDateString, aTimeString As String
Dim aDate, aTime As Date
aDateString = "February 12, 1969"
aTimeString = "4:35:47 PM"
' The following line of code sets aDate to a Date value.
aDate = CDate(aDateString)
' The following line of code sets aTime to Date value.
```

```
aTime = CDate(aTimeString)
```

CDbl Example

VB

```
Dim aDec As Decimal
Dim aDbl As Double
' The following line of code uses the literal type character D to make aDec a Decimal.
aDec = 234.456784D
' The following line of code sets aDbl to 1.9225456288E+1.
aDbl = CDbl(aDec * 8.2D * 0.01D)
```

CDec Example

The following example uses the **CDec** function to convert a numeric value to **Decimal**.

VB

```
Dim aDouble As Double
Dim aDecimal As Decimal
aDouble = 10000000.0587
' The following line of code sets aDecimal to 10000000.0587.
aDecimal = CDec(aDouble)
```

CInt Example

The following example uses the **CInt** function to convert a value to **Integer**.

VB

```
Dim aDbl As Double
Dim anInt As Integer
aDbl = 2345.5678
' The following line of code sets anInt to 2346.
anInt = CInt(aDbl)
```

CLng Example

The following example uses the **CLng** function to convert values to **Long**.

VB

```
Dim aDbl1, aDbl2 As Double
```

```
Dim aLng1, aLng2 As Long
aDbl1 = 25427.45
aDbl2 = 25427.55
' The following line of code sets aLng1 to 25427.
aLng1 = CLng(aDbl1)
' The following line of code sets aLng2 to 25428.
aLng2 = CLng(aDbl2)
```

CObj Example

The following example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

VB

```
Dim aDouble As Double
Dim anObject As Object
aDouble = 2.7182818284
' The following line of code sets anObject to a pointer to aDouble.
anObject = CObj(aDouble)
```

CSByte Example

The following example uses the **CSByte** function to convert a numeric value to **SByte**.

VB

```
Dim aDouble As Double
Dim anSByte As SByte
aDouble = 39.501
' The following line of code sets anSByte to 40.
anSByte = CSByte(aDouble)
```

CShort Example

The following example uses the **CShort** function to convert a numeric value to **Short**.

VB

```
Dim aByte As Byte
Dim aShort As Short
aByte = 100
' The following line of code sets aShort to 100.
aShort = CShort(aByte)
```

CSng Example

The following example uses the **CSng** function to convert values to **Single**.

VB

```
Dim aDouble1, aDouble2 As Double
Dim aSingle1, aSingle2 As Single
aDouble1 = 75.3421105
aDouble2 = 75.3421567
' The following line of code sets aSingle1 to 75.34211.
aSingle1 = CSng(aDouble1)
' The following line of code sets aSingle2 to 75.34216.
aSingle2 = CSng(aDouble2)
```

CStr Example

The following example uses the **CStr** function to convert a numeric value to **String**.

VB

```
Dim aDouble As Double
Dim aString As String
aDouble = 437.324
' The following line of code sets aString to "437.324".
aString = CStr(aDouble)
```

The following example uses the **CStr** function to convert **Date** values to **String** values.

VB

```
Dim aDate As Date
Dim aString As String
' The following line of code generates a COMPILER ERROR because of invalid format.
' aDate = #February 12, 1969 00:00:00#
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' The following line of code sets the time component of aDate to midnight.
aDate = #2/12/1969#
' The following conversion suppresses the neutral time value of 00:00:00.
' The following line of code sets aString to "2/12/1969".
aString = CStr(aDate)
' The following line of code sets the time component of aDate to one second past
midnight.
aDate = #2/12/1969 12:00:01 AM#
' The time component becomes part of the converted value.
' The following line of code sets aString to "2/12/1969 12:00:01 AM".
aString = CStr(aDate)
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47

PM". However, **CStr** suppresses the *neutral values* of 1/1/0001 for the date and 00:00:00 for the time.

For more detail on the values returned by **CStr**, see Return Values for the CStr Function (Visual Basic).

CUInt Example

The following example uses the **CUInt** function to convert a numeric value to **UInteger**.

VB

```
Dim aDouble As Double
Dim aUInteger As UInteger
aDouble = 39.501
' The following line of code sets aUInteger to 40.
aUInteger = CUInt(aDouble)
```

CULng Example

The following example uses the **CULng** function to convert a numeric value to **ULong**.

VB

```
Dim aDouble As Double
Dim aULong As ULong
aDouble = 39.501
' The following line of code sets aULong to 40.
aULong = CULng(aDouble)
```

CUShort Example

The following example uses the **CUShort** function to convert a numeric value to **UShort**.

VB

```
Dim aDouble As Double
Dim aUShort As UShort
aDouble = 39.501
' The following line of code sets aUShort to 40.
aUShort = CUShort(aDouble)
```

See Also

Asc

AscW
Chr
ChrW
Int
Fix
Format
Hex
Oct
Str
Val
Conversion Functions (Visual Basic)
Type Conversions in Visual Basic