Visual Basic for Applications Reference

Visual Studio 6.0

+ Operator

See Also Example Specifics

Used to sum two numbers.

Syntax

result = *expression1*+*expression2*

The + operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

When you use the + operator, you may not be able to determine whether addition or string concatenation will occur. Use the **&** operator for concatenation to eliminate ambiguity and provide self-documenting code.

If at least one expression is not a Variant, the following rules apply:

If	Then
Both expressions are numeric data types (Byte, Boolean, Integer, Long, Single, Double, Date, Currency, or Decimal)	Add.
Both expressions are String	Concatenate.
One expression is a numeric data type and the other is any **Variant** except Null	Add.
One expression is a **String** and the other is any **Variant** except **Null**	Concatenate.
One expression is an Empty **Variant**	Return the remaining expression unchanged as *result*.

One expression is a numeric data type and the other is a **String**	A `Type mismatch` error occurs.
Either expression is **Null**	*result* is **Null**.

If both expressions are **Variant** expressions, the following rules apply:

If	**Then**
Both **Variant** expressions are numeric	Add.
Both **Variant** expressions are strings	Concatenate.
One **Variant** expression is numeric and the other is a string	Add.

For simple arithmetic addition involving only expressions of numeric data types, the data type of *result* is usually the same as that of the most precise expression. The order of precision, from least to most precise, is **Byte**, **Integer**, **Long**, **Single**, **Double**, **Currency**, and **Decimal**. The following are exceptions to this order:

If	**Then *result* is**
A **Single** and a **Long** are added,	a **Double**.
The data type of *result* is a **Long**, **Single**, or **Date** variant that overflows its legal range,	converted to a **Double** variant.
The data type of *result* is a **Byte** variant that overflows its legal range,	converted to an **Integer** variant.
The data type of *result* is an **Integer** variant that overflows its legal range,	converted to a **Long** variant.
A **Date** is added to any data type,	a **Date**.

If one or both expressions are **Null** expressions, *result* is **Null**. If both expressions are **Empty**, *result* is an **Integer**. However, if only one expression is **Empty**, the other expression is returned unchanged as *result*.

Note The order of precision used by addition and subtraction is not the same as the order of precision used by multiplication.

Visual Basic for Applications Reference

+ Operator Example

This example uses the + operator to sum numbers. The + operator can also be used to concatenate strings. However, to eliminate ambiguity, you should use the **&** operator instead. If the components of an expression created with the + operator include both strings and numerics, the arithmetic result is assigned. If the components are exclusively strings, the strings are concatenated.

```
Dim MyNumber, Var1, Var2
MyNumber = 2 + 2   ' Returns 4.
MyNumber = 4257.04 + 98112   ' Returns 102369.04.

Var1 = "34": Var2 = 6   ' Initialize mixed variables.
MyNumber = Var1 + Var2   ' Returns 40.

Var1 = "34": Var2 = "6"   ' Initialize variables with strings.
MyNumber = Var1 + Var2   ' Returns "346" (string concatenation).
```

Visual Basic for Applications Reference

Visual Studio 6.0

- Operator

See Also Example Specifics

Used to find the difference between two numbers or to indicate the negative value of a numeric expression.

Syntax 1

result = *number1number2*

Syntax 2

number

The operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number	Required; any numeric expression.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

In Syntax 1, the operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the operator is used as the unary negation operator to indicate the negative value of an expression.

The data type of *result* is usually the same as that of the most precise expression. The order of precision, from least to most precise, is Byte, Integer, Long, Single, Double, Currency, and Decimal. The following are exceptions to this order:

If	Then *result* is
Subtraction involves a **Single** and a **Long**,	converted to a **Double**.
The data type of *result* is a **Long**, **Single**, or Date variant that overflows its legal range,	converted to a Variant containing a **Double**.
The data type of *result* is a **Byte** variant that overflows its legal range,	converted to an **Integer** variant.

The data type of *result* is an **Integer** variant that overflows its legal range,	converted to a **Long** variant.
Subtraction involves a **Date** and any other data type,	a **Date**.
Subtraction involves two **Date** expressions,	a **Double**.

One or both expressions are Null expressions, *result* is **Null**. If an expression is Empty, it is treated as 0.

Note The order of precision used by addition and subtraction is not the same as the order of precision used by multiplication.

Visual Basic for Applications Reference

- Operator Example

This example uses the - operator to calculate the difference between two numbers.

```
Dim MyResult
MyResult = 4 - 2   ' Returns 2.
MyResult = 459.35 - 334.90   ' Returns 124.45.
```

Visual Basic for Applications Reference

Visual Studio 6.0

* Operator

See Also Example Specifics

Used to multiply two numbers.

Syntax

result = *number1**number2*

The * operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

The data type of *result* is usually the same as that of the most precise expression. The order of precision, from least to most precise, is Byte, Integer, Long, Single, Currency, Double, and Decimal. The following are exceptions to this order:

If	Then *result* is
Multiplication involves a **Single** and a **Long**,	converted to a **Double**.
The data type of *result* is a **Long**, **Single**, or **Date** variant that overflows its legal range,	converted to a **Variant** containing a **Double**.
The data type of *result* is a **Byte** variant that overflows its legal range,	converted to an **Integer** variant.
the data type of *result* is an **Integer** variant that overflows its legal range,	converted to a **Long** variant.

If one or both expressions are Null expressions, *result* is **Null**. If an expression is Empty, it is treated as 0.

Note The order of precision used by multiplication is not the same as the order of precision used by addition and subtraction.

Visual Basic for Applications Reference

* Operator Example

This example uses the * operator to multiply two numbers.

```
Dim MyValue
MyValue = 2 * 2   ' Returns 4.
MyValue = 459.35 * 334.90    ' Returns 153836.315.
```

Visual Basic for Applications Reference

Visual Studio 6.0

/ Operator

See Also Example Specifics

Used to divide two numbers and return a floating-point result.

Syntax

result = *number1*/*number2*

The **/** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

The data type of *result* is usually a Double or a **Double** variant. The following are exceptions to this rule:

If	Then *result* is
Both expressions are Byte, Integer, or Single expressions,	a **Single** unless it overflows its legal range; in which case, an error occurs.
Both expressions are **Byte**, **Integer**, or **Single** variants,	a **Single** variant unless it overflows its legal range; in which case, *result* is a Variant containing a **Double**.
Division involves a Decimal and any other data type,	a **Decimal** data type.

One or both expressions are Null expressions, *result* is **Null**. Any expression that is Empty is treated as 0.

Visual Basic for Applications Reference

/ Operator Example

This example uses the **/** operator to perform floating-point division.

```
Dim MyValue
MyValue = 10 / 4   ' Returns 2.5.
MyValue = 10 / 3   ' Returns 3.333333.
```

Visual Basic for Applications Reference

Visual Studio 6.0

\ Operator

See Also Example Specifics

Used to divide two numbers and return an integer result.

Syntax

result = *number1**number2*

The **** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

Before division is performed, the numeric expressions are rounded to Byte, Integer, or Long expressions.

Usually, the data type of *result* is a **Byte**, **Byte** variant, **Integer**, **Integer** variant, **Long**, or **Long** variant, regardless of whether *result* is a whole number. Any fractional portion is truncated. However, if any expression is Null, *result* is **Null**. Any expression that is Empty is treated as 0.

Visual Basic for Applications Reference

\ Operator Example

This example uses the **** operator to perform integer division.

```
Dim MyValue
MyValue = 11 \ 4   ' Returns 2.
MyValue = 9 \ 3   ' Returns 3.
MyValue = 100 \ 3   ' Returns 33.
```

Visual Basic for Applications Reference

Visual Studio 6.0

^Operator

See Also Example Specifics

Used to raise a number to the power of an exponent.

Syntax

result = *number*^*exponent*

The ^ operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number	Required; any numeric expression.
exponent	Required; any numeric expression.

Remarks

A *number* can be negative only if *exponent* is an integer value. When more than one exponentiation is performed in a single expression, the ^ operator is evaluated as it is encountered from left to right.

Usually, the data type of *result* is a Double or a Variant containing a **Double**. However, if either *number* or *exponent* is a Null expression, *result* is **Null**.

Visual Basic for Applications Reference

^ Operator Example

This example uses the ^ operator to raise a number to the power of an exponent.

```
Dim MyValue
MyValue = 2 ^ 2   ' Returns 4.
MyValue = 3 ^ 3 ^ 3   ' Returns 19683.
MyValue = (-5) ^ 3   ' Returns -125.
```

Visual Basic for Applications Reference

Visual Studio 6.0

& Operator

See Also Example Specifics

Used to force string concatenation of two expressions.

Syntax

result = *expression1* **&** *expression2*

The **&** operator syntax has these parts:

Part	Description
result	Required; any String or Variant variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If an *expression* is not a string, it is converted to a **String** variant. The data type of *result* is **String** if both expressions are string expressions; otherwise, *result* is a **String** variant. If both expressions are Null, *result* is **Null**. However, if only one *expression* is **Null**, that expression is treated as a zero-length string ("") when concatenated with the other expression. Any expression that is Empty is also treated as a zero-length string.

Visual Basic for Applications Reference

& Operator Example

This example uses the **&** operator to force string concatenation.

```
Dim MyStr
MyStr = "Hello" & " World"    ' Returns "Hello World".
MyStr = "Check " & 123 & " Check"    ' Returns "Check 123 Check".
```

Visual Basic for Applications Reference

Visual Studio 6.0

= Operator

See Also Example Specifics

Description

Used to assign a value to a variable or property.

Syntax

variable = *value*

The = operator syntax has these parts:

Part	Description
variable	Any variable or any writable property.
value	Any numeric or string literal, constant, or expression.

Remarks

The name on the left side of the equal sign can be a simple scalar variable or an element of an array. Properties on the left side of the equal sign can only be those properties that are writable at run time.

Visual Basic for Applications Reference

Visual Studio 6.0

Not Operator

See Also Example Specifics

Used to perform logical negation on an expression.

Syntax

result = **Not** *expression*

The **Not** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression	Required; any expression.

Remarks

The following table illustrates how *result* is determined:

If *expression* is	Then *result* is
True	**False**
False	**True**
Null	**Null**

In addition, the **Not** operator inverts the bit values of any variable and sets the corresponding bit in *result* according to the following table:

If bit in *expression* is	Then bit in *result* is
0	1
1	0

Visual Basic for Applications Reference

Not Operator Example

This example uses the **Not** operator to perform logical negation on an expression.

```
Dim A, B, C, D, MyCheck
A = 10: B = 8: C = 6: D = Null   ' Initialize variables.
MyCheck = Not(A > B)   ' Returns False.
MyCheck = Not(B > A)   ' Returns True.
MyCheck = Not(C > D)   ' Returns Null.
MyCheck = Not A   ' Returns -11 (bitwise comparison).
```

Visual Basic for Applications Reference

Visual Studio 6.0

And Operator

See Also Example Specifics

Used to perform a logical conjunction on two expressions.

Syntax

result = *expression1* **And** *expression2*

The **And** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If both expressions evaluate to **True**, *result* is **True**. If either expression evaluates to **False**, *result* is **False.** The following table illustrates how *result* is determined:

If *expression1* is	And *expression2* is	The *result* is
True	**True**	**True**
True	**False**	**False**
True	Null	**Null**
False	**True**	**False**
False	**False**	**False**
False	**Null**	**False**
Null	**True**	**Null**

Null	**False**	**False**
Null	**Null**	**Null**

The **And** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	0
0	1	0
1	0	0
1	1	1

Visual Basic for Applications Reference

And Operator Example

This example uses the **And** operator to perform a logical conjunction on two expressions.

```
Dim A, B, C, D, MyCheck
A = 10: B = 8: C = 6: D = Null   ' Initialize variables.
MyCheck = A > B And B > C   ' Returns True.
MyCheck = B > A And B > C   ' Returns False.
MyCheck = A > B And B > D   ' Returns Null.
MyCheck = A And B   ' Returns 8 (bitwise comparison).
```

Visual Basic for Applications Reference

Visual Studio 6.0

Or Operator

See Also Example Specifics

Used to perform a logical disjunction on two expressions.

Syntax

result = *expression1* **Or** *expression2*

The **Or** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If either or both expressions evaluate to **True**, *result* is **True**. The following table illustrates how *result* is determined:

If ***expression1*** is	And ***expression2*** is	Then ***result*** is
True	**True**	**True**
True	**False**	**True**
True	Null	**True**
False	**True**	**True**
False	**False**	**False**
False	**Null**	**Null**
Null	**True**	**True**
Null	**False**	**Null**

Null	**Null**	**Null**

The **Or** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	**And bit in *expression2* is**	**Then *result* is**
0	0	0
0	1	1
1	0	1
1	1	1

Visual Basic for Applications Reference

Or Operator Example

This example uses the **Or** operator to perform logical disjunction on two expressions.

```
Dim A, B, C, D, MyCheck
A = 10: B = 8: C = 6: D = Null   ' Initialize variables.
MyCheck = A > B Or B > C   ' Returns True.
MyCheck = B > A Or B > C   ' Returns True.
MyCheck = A > B Or B > D   ' Returns True.
MyCheck = B > D Or B > A   ' Returns Null.
MyCheck = A Or B   ' Returns 10 (bitwise comparison).
```

Visual Basic for Applications Reference

Visual Studio 6.0

Xor Operator

See Also Example Specifics

Used to perform a logical exclusion on two expressions.

Syntax

[*result* =] *expression1* **Xor** *expression2*

The **Xor** operator syntax has these parts:

Part	Description
result	Optional; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If one, and only one, of the expressions evaluates to **True**, *result* is **True**. However, if either expression is Null, *result* is also **Null**. When neither expression is **Null**, *result* is determined according to the following table:

If *expression1* is	And *expression2* is	Then *result* is
True	**True**	**False**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

The **Xor** operator performs as both a logical and bitwise operator. A bit-wise comparison of two expressions using exclusive-or logic to form the result, as shown in the following table:

If bit in *expression1* is	And bit in *expression2* is	Then *result* is
0	0	0
0	1	1
1	0	1
1	1	0

Visual Basic for Applications Reference

Xor Operator Example

This example uses the **Xor** operator to perform logical exclusion on two expressions.

```
Dim A, B, C, D, MyCheck
A = 10: B = 8: C = 6: D = Null   ' Initialize variables.
MyCheck = A > B Xor B > C   ' Returns False.
MyCheck = B > A Xor B > C   ' Returns True.
MyCheck = B > A Xor C > B   ' Returns False.
MyCheck = B > D Xor A > B   ' Returns Null.
MyCheck = A Xor B   ' Returns 2 (bitwise comparison).
```

Visual Basic for Applications Reference

Visual Studio 6.0

Eqv Operator

See Also Example Specifics

Used to perform a logical equivalence on two expressions.

Syntax

result = *expression1* **Eqv** *expression2*

The **Eqv** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If either expression is Null, *result* is also **Null**. When neither expression is **Null**, *result* is determined according to the following table:

If *expression1* is	And *expression2* is	The *result* is
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**True**

The **Eqv** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	1
0	1	0
1	0	0
1	1	1

Visual Basic for Applications Reference

Eqv Operator Example

This example uses the **Eqv** operator to perform logical equivalence on two expressions.

```
Dim A, B, C, D, MyCheck
A = 10: B = 8: C = 6: D = Null   ' Initialize variables.
MyCheck = A > B Eqv B > C   ' Returns True.
MyCheck = B > A Eqv B > C   ' Returns False.
MyCheck = A > B Eqv B > D   ' Returns Null.
MyCheck = A Eqv B   ' Returns -3 (bitwise comparison).
```

Visual Basic for Applications Reference

Visual Studio 6.0

Mod Operator

See Also Example Specifics

Used to divide two numbers and return only the remainder.

Syntax

result = *number1* **Mod** *number2*

The **Mod** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

The modulus, or remainder, operator divides *number1* by *number2* (rounding floating-point numbers to integers) and returns only the remainder as *result*. For example, in the following expression, A (*result*) equals 5.

```
A = 19 Mod 6.7
```

Usually, the data type of *result* is a Byte, **Byte** variant, Integer, **Integer** variant, Long, or Variant containing a **Long**, regardless of whether or not *result* is a whole number. Any fractional portion is truncated. However, if any expression is Null, *result* is **Null**. Any expression that is Empty is treated as 0.

Mod Operator Example

This example sets the column width of every other column on Sheet1 to 4 points.

```
For Each col In Worksheets("Sheet1").Columns
    If col.Column Mod 2 = 0 Then
        col.ColumnWidth = 4
    End If
Next col
```

This example sets the row height of every other row on Sheet1 to 4 points.

```
For Each rw In Worksheets("Sheet1").Rows
    If rw.Row Mod 2 = 0 Then
        rw.RowHeight = 4
    End If
Next rw
```

This example selects every other item in list box one on Sheet1.

```
Dim items() As Boolean
Set lbox = Worksheets("Sheet1").ListBoxes(1)
ReDim items(1 To lbox.ListCount)
For i = 1 To lbox.ListCount
    If i Mod 2 = 1 Then
        items(i) = True
    Else
        items(i) = False
    End If
Next
lbox.MultiSelect = xlExtended
lbox.Selected = items
```

Visual Basic for Applications Reference

Visual Studio 6.0

Is Operator

See Also Example Specifics

Used to compare two object reference variables.

Syntax

result = *object1* **Is** *object2*

The **Is** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
object1	Required; any object name.
object2	Required; any object name.

Remarks

If *object1* and *object2* both refer to the same object, *result* is **True**; if they do not, *result* is **False**. Two variables can be made to refer to the same object in several ways.

In the following example, A has been set to refer to the same object as B:

```
Set A = B
```

The following example makes A and B refer to the same object as C:

```
Set A = C
Set B = C
```

Is Operator Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example selects the intersection of two named ranges ("rg1" and "rg2") on Sheet1. If the ranges dont intersect, the example displays a message.

```
Worksheets("Sheet1").Activate
Set isect = Application.Intersect(Range("rg1"), Range("rg2"))
If isect Is Nothing Then
    MsgBox "Ranges do not intersect"
Else
    isect.Select
End If
```

This example finds the first occurrence of the word Phoenix in column B on Sheet1 and then displays the address of the cell that contains this word. If the word isnt found, the example diplays a message.

```
Set foundCell = Worksheets("Sheet1").Columns("B").Find("Phoenix")
If foundCell Is Nothing Then
    MsgBox "The word was not found"
Else
    MsgBox "The word was found in cell " & foundCell.Address
End If
```

Visual Basic for Applications Reference

Visual Studio 6.0

Imp Operator

See Also Example Specifics

Used to perform a logical implication on two expressions.

Syntax

result = *expression1* **Imp** *expression2*

The **Imp** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

The following table illustrates how *result* is determined:

If *expression1* is	And *expression2* is	The *result* is
True	**True**	**True**
True	**False**	**False**
True	Null	**Null**
False	**True**	**True**
False	**False**	**True**
False	**Null**	**True**
Null	**True**	**True**
Null	**False**	**Null**

Null	**Null**	**Null**

The **Imp** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	1
0	1	1
1	0	0
1	1	1

Visual Basic for Applications Reference

Imp Operator Example

This example uses the **Imp** operator to perform logical implication on two expressions.

```
Dim A, B, C, D, MyCheck
A = 10: B = 8: C = 6: D = Null   ' Initialize variables.
MyCheck = A > B Imp B > C   ' Returns True.
MyCheck = A > B Imp C > B   ' Returns False.
MyCheck = B > A Imp C > B   ' Returns True.
MyCheck = B > A Imp C > D   ' Returns True.
MyCheck = C > D Imp B > A   ' Returns Null.
MyCheck = B Imp A   ' Returns -1 (bitwise comparison).
```

Visual Basic for Applications Reference

Visual Studio 6.0

Like Operator

See Also Example Specifics

Used to compare two strings.

Syntax

result = *string* **Like** *pattern*

The **Like** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
string	Required; any string expression.
pattern	Required; any string expression conforming to the pattern-matching conventions described in Remarks.

Remarks

If *string* matches *pattern*, *result* is **True**; if there is no match, *result* is **False**. If either *string* or *pattern* is Null, *result* is **Null**.

The behavior of the **Like** operator depends on the **Option Compare** statement. The default string-comparison method for each module is **Option Compare Binary**.

Option Compare Binary results in string comparisons based on a sort order derived from the internal binary representations of the characters. Sort order is determined by the code page. In the following example, a typical binary sort order is shown:

A < B < E < Z < a < b < e < z < < < < < <

Option Compare Text results in string comparisons based on a case-insensitive, textual sort order determined by your system's locale. When you sort The same characters using **Option Compare Text**, the following text sort order is produced:

(A=a) < (=) < (B=b) < (E=e) < (=) < (Z=z) < (=)

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features allow you to use wildcard characters, character lists, or character ranges, in any combination, to match strings. The following table shows the characters allowed in *pattern* and what they match:

Characters in *pattern*	Matches in *string*

?	Any single character.
*****	Zero or more characters.
#	Any single digit (09).
[*charlist*]	Any single character in *charlist*.
[**!***charlist*]	Any single character not in *charlist*.

A group of one or more characters (*charlist*) enclosed in brackets (**[ ]**) can be used to match any single character in *string* and can include almost any character code, including digits.

Note To match the special characters left bracket (**[**), question mark (**?**), number sign (**#**), and asterisk (*****), enclose them in brackets. The right bracket (**]**) can't be used within a group to match itself, but it can be used outside a group as an individual character.

By using a hyphen () to separate the upper and lower bounds of the range, *charlist* can specify a range of characters. For example, `[A-Z]` results in a match if the corresponding character position in *string* contains any uppercase letters in the range AZ. Multiple ranges are included within the brackets without delimiters.

The meaning of a specified range depends on the character ordering valid at run time (as determined by **Option Compare** and the locale setting of the system the code is running on). Using the **Option Compare Binary** example, the range `[AE]` matches A, B and E. With **Option Compare Text**, `[AE]` matches A, a, , , B, b, E, e. The range does not match or because accented characters fall after unaccented characters in the sort order.

Other important rules for pattern matching include the following:

- An exclamation point (**!**) at the beginning of *charlist* means that a match is made if any character except the characters in *charlist* is found in *string*. When used outside brackets, the exclamation point matches itself.
- A hyphen () can appear either at the beginning (after an exclamation point if one is used) or at the end of *charlist* to match itself. In any other location, the hyphen is used to identify a range of characters.
- When a range of characters is specified, they must appear in ascending sort order (from lowest to highest). `[A-Z]` is a valid pattern, but `[Z-A]` is not.
- The character sequence `[]` is considered a zero-length string ("").

In some languages, there are special characters in the alphabet that represent two separate characters. For example, several languages use the character "" to represent the characters "a" and "e" when they appear together. The **Like** operator recognizes that the single special character and the two individual characters are equivalent.

When a language that uses a special character is specified in the system locale settings, an occurrence of the single special character in either *pattern* or *string* matches the equivalent 2-character sequence in the other string. Similarly, a single special character in *pattern* enclosed in brackets (by itself, in a list, or in a range) matches the equivalent 2-character sequence in *string*.

Like Operator Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example deletes every defined name that contains "temp". The `Option Compare Text` statement must be included at the top of any module that contains this example.

```
For Each nm In ActiveWorkbook.Names
    If nm.Name Like "*temp*" Then
        nm.Delete
    End If
Next nm
```

This example adds an arrowhead to every shape on Sheet1 that has the word Line in its name.

```
For Each d In Worksheets("Sheet1").DrawingObjects
    If d.Name Like "*Line*" Then
        d.ArrowHeadLength = xlLong
        d.ArrowHeadStyle = xlOpen
        d.ArrowHeadWidth = xlNarrow
    End If
Next
```

Visual Basic for Applications Reference

Visual Studio 6.0

Comparison Operators

See Also Example Specifics

Used to compare expressions.

Syntax

result = *expression1 comparisonoperator expression2*

result = *object1* **Is** *object2*

result = *string* **Like** *pattern*

Comparison operators have these parts:

Part	Description
result	Required; any numeric variable.
expression	Required; any expression.
comparisonoperator	Required; any comparison operator.
object	Required; any object name.
string	Required; any string expression.
pattern	Required; any string expression or range of characters.

Remarks

The following table contains a list of the comparison operators and the conditions that determine whether *result* is **True**, **False**, or Null:

Operator	True if	False if	Null if
< (Less than)	*expression1* < *expression2*	*expression1* >= *expression2*	*expression1* or *expression2* = **Null**
<= (Less than or equal to)	*expression1* <= *expression2*	*expression1* > *expression2*	*expression1* or *expression2* = **Null**

> (Greater than)	*expression1* > *expression2*	*expression1* <= *expression2*	*expression1* or *expression2* = **Null**
>= (Greater than or equal to)	*expression1* >= *expression2*	*expression1* < *expression2*	*expression1* or *expression2* = **Null**
= (Equal to)	*expression1* = *expression2*	*expression1* <> *expression2*	*expression1* or *expression2* = **Null**
<> (Not equal to)	*expression1* <> *expression2*	*expression1* = *expression2*	*expression1* or *expression2* = **Null**

Note The **Is** and **Like** operators have specific comparison functionality that differs from the operators in the table.

When comparing two expressions, you may not be able to easily determine whether the expressions are being compared as numbers or as strings. The following table shows how the expressions are compared or the result when either expression is not a Variant:

If	**Then**
Both expressions are numeric data types (Byte, Boolean, Integer, Long, Single, Double, Date, Currency, or Decimal)	Perform a numeric comparison.
Both expressions are String	Perform a string comparison.
One expression is a numeric data type and the other is a **Variant** that is, or can be, a number	Perform a numeric comparison.
One expression is a numeric data type and the other is a string **Variant** that can't be converted to a number	A `Type Mismatch` error occurs.
One expression is a **String** and the other is any **Variant** except a **Null**	Perform a string comparison.
One expression is Empty and the other is a numeric data type	Perform a numeric comparison, using 0 as the **Empty** expression.
One expression is **Empty** and the other is a **String**	Perform a string comparison, using a zero-length string ("") as the **Empty** expression.

If *expression1* and *expression2* are both **Variant** expressions, their underlying type determines how they are compared. The following table shows how the expressions are compared or the result from the comparison, depending on the underlying type of the **Variant**:

If	**Then**
Both **Variant** expressions are numeric	Perform a numeric comparison.
Both **Variant** expressions are strings	Perform a string comparison.

One **Variant** expression is numeric and the other is a string	The numeric expression is less than the string expression.
One **Variant** expression is **Empty** and the other is numeric	Perform a numeric comparison, using 0 as the **Empty** expression.
One **Variant** expression is **Empty** and the other is a string	Perform a string comparison, using a zero-length string ("") as the **Empty** expression.
Both **Variant** expressions are **Empty**	The expressions are equal.

When a **Single** is compared to a **Double**, the **Double** is rounded to the precision of the **Single**.

If a **Currency** is compared with a **Single** or **Double**, the **Single** or **Double** is converted to a **Currency**. Similarly, when a **Decimal** is compared with a **Single** or **Double**, the **Single** or **Double** is converted to a **Decimal**. For **Currency**, any fractional value less than .0001 may be lost; for **Decimal**, any fractional value less than 1E-28 may be lost, or an overflow error can occur. Such fractional value loss may cause two values to compare as equal when they are not.

Visual Basic for Applications Reference

Comparison Operators Example

This example shows various uses of comparison operators, which you use to compare expressions.

```
Dim MyResult, Var1, Var2
MyResult = (45 < 35)   ' Returns False.
MyResult = (45 = 45)   ' Returns True.
MyResult = (4 <> 3)   ' Returns True.
MyResult = ("5" > "4")   ' Returns True.

Var1 = "5": Var2 = 4   ' Initialize variables.
MyResult = (Var1 > Var2)   ' Returns True.

Var1 = 5: Var2 = Empty
MyResult = (Var1 > Var2)   ' Returns True.

Var1 = 0: Var2 = Empty
MyResult = (Var1 = Var2)   ' Returns True.
```

Visual Basic for Applications Reference

Visual Studio 6.0

AddressOf Operator

See Also Example Specifics

A unary operator that causes the address of the procedure it precedes to be passed to an API procedure that expects a function pointer at that position in the argument list.

Syntax

AddressOf *procedurename*

The required *procedurename* specifies the procedure whose address is to be passed. It must represent a procedure in a standard module module in the project in which the call is made.

Remarks

When a procedure name appears in an argument list, usually the procedure is evaluated, and the address of the procedures return value is passed. **AddressOf** permits the address of the procedure to be passed to a Windows API function in a dynamic-link library (DLL), rather passing the procedure's return value. The API function can then use the address to call the Basic procedure, a process known as a callback. The **AddressOf** operator appears only in the call to the API procedure.

Although you can use **AddressOf** to pass procedure pointers among Basic procedures, you can't call a function through such a pointer from within Basic. This means, for example, that a class written in Basic can't make a callback to its controller using such a pointer. When using **AddressOf** to pass a procedure pointer among procedures within Basic, the parameter of the called procedure must be typed **As Long**.

Warning Using **AddressOf** may cause unpredictable results if you don't completely understand the concept of function callbacks. You must understand how the Basic portion of the callback works, and also the code of the DLL into which you are passing your function address. Debugging such interactions is difficult since the program runs in the same process as the development environment. In some cases, systematic debugging may not be possible.

Note You can create your own call-back function prototypes in DLLs compiled with Microsoft Visual C++ (or similar tools). To work with **AddressOf**, your prototype must use the __stdcall calling convention. The default calling convention (__cdecl) will not work with **AddressOf**.

Since the caller of a callback is not within your program, it is important that an error in the callback procedure not be propagated back to the caller. You can accomplish this by placing the **On Error Resume Next** statement at the beginning of the callback procedure.

Visual Basic for Applications Reference

AddressOf Operator Example

The following example creates a form with a list box containing an alphabetically sorted list of the fonts in your system.

To run this example, create a form with a list box on it. The code for the form is as follows:

```
Option Explicit

Private Sub Form_Load()
    Module1.FillListWithFonts List1
End Sub
```

Place the following code in a module. The third argument in the definition of the EnumFontFamilies function is a **Long** that represents a procedure. The argument must contain the address of the procedure, rather than the value that the procedure returns. In the call to EnumFontFamilies, the third argument requires the **AddressOf** operator to return the address of the EnumFontFamProc procedure, which is the name of the callback procedure you supply when calling the Windows API function, **EnumFontFamilies**. Windows calls EnumFontFamProc once for each of the font families on the system when you pass **AddressOf** EnumFontFamProc to **EnumFontFamilies**. The last argument passed to **EnumFontFamilies** specifies the list box in which the information is displayed.

```
'Font enumeration types
Public Const LF_FACESIZE = 32
Public Const LF_FULLFACESIZE = 64

Type LOGFONT
        lfHeight As Long
        lfWidth As Long
        lfEscapement As Long
        lfOrientation As Long
        lfWeight As Long
        lfItalic As Byte
        lfUnderline As Byte
        lfStrikeOut As Byte
        lfCharSet As Byte
        lfOutPrecision As Byte
        lfClipPrecision As Byte
        lfQuality As Byte
        lfPitchAndFamily As Byte
        lfFaceName(LF_FACESIZE) As Byte
End Type

Type NEWTEXTMETRIC
        tmHeight As Long
        tmAscent As Long
        tmDescent As Long
        tmInternalLeading As Long
        tmExternalLeading As Long
        tmAveCharWidth As Long
        tmMaxCharWidth As Long
        tmWeight As Long
        tmOverhang As Long
        tmDigitizedAspectX As Long
        tmDigitizedAspectY As Long
        tmFirstChar As Byte
```

```
        tmLastChar As Byte
        tmDefaultChar As Byte
        tmBreakChar As Byte
        tmItalic As Byte
        tmUnderlined As Byte
        tmStruckOut As Byte
        tmPitchAndFamily As Byte
        tmCharSet As Byte
        ntmFlags As Long
        ntmSizeEM As Long
        ntmCellHeight As Long
        ntmAveWidth As Long
End Type

' ntmFlags field flags
Public Const NTM_REGULAR = &H40&
Public Const NTM_BOLD = &H20&
Public Const NTM_ITALIC = &H1&

'  tmPitchAndFamily flags
Public Const TMPF_FIXED_PITCH = &H1
Public Const TMPF_VECTOR = &H2
Public Const TMPF_DEVICE = &H8
Public Const TMPF_TRUETYPE = &H4

Public Const ELF_VERSION = 0
Public Const ELF_CULTURE_LATIN = 0

'  EnumFonts Masks
Public Const RASTER_FONTTYPE = &H1
Public Const DEVICE_FONTTYPE = &H2
Public Const TRUETYPE_FONTTYPE = &H4

Declare Function EnumFontFamilies Lib "gdi32" Alias _
    "EnumFontFamiliesA" _
    (ByVal hDC As Long, ByVal lpszFamily As String, _
    ByVal lpEnumFontFamProc As Long, LParam As Any) As Long
Declare Function GetDC Lib "user32" (ByVal hWnd As Long) As Long
Declare Function ReleaseDC Lib "user32" (ByVal hWnd As Long, _
    ByVal hDC As Long) As Long

Function EnumFontFamProc(lpNLF As LOGFONT, lpNTM As NEWTEXTMETRIC, _
    ByVal FontType As Long, LParam As ListBox) As Long
Dim FaceName As String
Dim FullName As String
    FaceName = StrConv(lpNLF.lfFaceName, vbUnicode)
    LParam.AddItem Left$(FaceName, InStr(FaceName, vbNullChar) - 1)
    EnumFontFamProc = 1
End Function

Sub FillListWithFonts(LB As ListBox)
Dim hDC As Long
    LB.Clear
    hDC = GetDC(LB.hWnd)
    EnumFontFamilies hDC, vbNullString, AddressOf EnumFontFamProc, LB
    ReleaseDC LB.hWnd, hDC
End Sub
```