Arrays in Visual Basic

Visual Studio 2015

An array is a set of values that are logically related to each other, such as the number of students in each grade in a grammar school. If you are looking for help on arrays in Visual Basic for Applications (VBA), see the language reference.

By using an array, you can refer to these related values by the same name, and use a number that's called an index or subscript to tell them apart. The individual values are called the elements of the array. They're contiguous from index 0 through the highest index value.

In contrast to an array, a variable that contain a single value is called a *scalar* variable.

Some quick examples before explanation:

VB

```
'Declare a single-dimension array of 5 values
Dim numbers(4) As Integer

‘Declare a single-dimension array and set array element values
Dim numbers = New Integer() {1, 2, 4, 8}

 ‘Redefine the size of an existing array retaining the current values
ReDim Preserve numbers(15)

 ‘Redefine the size of an existing array, resetting the values
ReDim numbers(15)

‘Declare a multi-dimensional array
Dim matrix(5, 5) As Double

‘Declare a multi-dimensional array and set array element values
Dim matrix = New Integer(4, 4) {{1, 2}, {3, 4}, {5, 6}, {7, 8}}

 ‘Declare a jagged array
Dim sales()() As Double = New Double(11)() {}
```

In this topic

Array Elements in a Simple Array

The following example declares an array variable to hold the number of students in each grade in a grammar school.

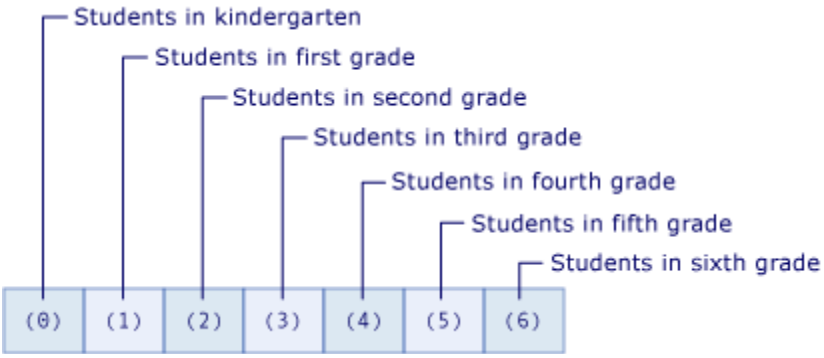
VB

```
Dim students(6) As Integer
```

The array `students` in the preceding example contains seven elements. The indexes of the elements range from 0 through 6. Having this array is simpler than declaring seven variables.

The following illustration shows the array `students`. For each element of the array:

- The index of the element represents the grade (index 0 represents kindergarten).
- The value that's contained in the element represents the number of students in that grade.



Elements of the "students" array

The following example shows how to refer to the first, second, and last element of the array `students`.

VB

```
Dim kindergarten As Integer = students(0)
Dim firstGrade As Integer = students(1)
```

```
Dim sixthGrade As Integer = students(6)
MsgBox("Students in kindergarten = " & CStr(kindergarten))
MsgBox("Students in first grade = " & CStr(firstGrade))
MsgBox("Students in sixth grade = " & CStr(sixthGrade))
```

You can refer to the array as a whole by using just the array variable name without indexes.

The array `students` in the preceding example uses one index and is said to be one-dimensional. An array that uses more than one index or subscript is called multidimensional. For more information, see the rest of this topic and Array Dimensions in Visual Basic.

Creating an Array

You can define the size of an array several ways. You can supply the size when the array is declared, as the following example shows.

VB

```
Dim cargoWeights(10) As Double
Dim atmospherePressures(2, 2, 4, 10) As Short
Dim inquiriesByYearMonthDay(20)()() As Byte
```

You can also use a **New** clause to supply the size of an array when it's created, as the following example shows.

VB

```
cargoWeights = New Double(10) {}
atmospherePressures = New Short(2, 2, 4, 10) {}
inquiriesByYearMonthDay = New Byte(20)()() {}
```

If you have an existing array, you can redefine its size by using the **Redim** statement. You can specify that the **Redim** statement should keep the values that are in the array, or you can specify that it create an empty array. The following example shows different uses of the **Redim** statement to modify the size of an existing array.

VB

```
' Assign a new array size and retain the current element values.
ReDim Preserve cargoWeights(20)
' Assign a new array size and retain only the first five element values.
ReDim Preserve cargoWeights(4)
' Assign a new array size and discard all current element values.
ReDim cargoWeights(15)
```

For more information, see ReDim Statement (Visual Basic).

Storing Values in an Array

You can access each location in an array by using an index of type **Integer**. You can store and retrieve values in an array by referencing each array location by using its index enclosed in parentheses. Indexes for multi-dimensional arrays are separated by commas (,). You need one index for each array dimension. The following example shows some statements that store values in arrays.

VB

```
Dim i = 4
Dim j = 2

Dim numbers(10) As Integer
Dim matrix(5, 5) As Double

numbers(i + 1) = 0
matrix(3, j * 2) = j
```

The following example shows some statements that get values from arrays.

VB

```
Dim v = 2
Dim i = 1
Dim j = 1
Dim k = 1
Dim wTotal As Double = 0.0
Dim sortedValues(5), rawValues(5), estimates(2, 2, 2) As Double
Dim lowestValue = sortedValues(0)
wTotal += (rawValues(v) ^ 2)
Dim firstGuess = estimates(i, j, k)
```

Populating an Array with Initial Values

By using an array literal, you can create an array that contains an initial set of values. An array literal consists of a list of comma-separated values that are enclosed in braces (**{}**).

When you create an array by using an array literal, you can either supply the array type or use type inference to determine the array type. The following code shows both options.

VB

```
Dim numbers = New Integer() {1, 2, 4, 8}
Dim doubles = {1.5, 2, 9.9, 18}
```

When you use type inference, the type of the array is determined by the dominant type in the list of values that's supplied for the array literal. The dominant type is a unique type to which all other types in the array literal can widen. If this unique type can't be determined, the dominant type is the unique type to which all other types in the array can narrow. If neither of these unique types can be determined, the dominant type is **Object**. For example, if the list of values that's supplied to the array literal contains values of type **Integer**, **Long**, and **Double**, the resulting array is of type **Double**. Both **Integer** and **Long** widen only to **Double**. Therefore, **Double** is the dominant type. For more information, see Widening and

Narrowing Conversions (Visual Basic). These inference rules apply to types that are inferred for arrays that are local variables that are defined in a class member. Although you can use array literals when you create class-level variables, you can't use type inference at the class level. As a result, array literals that are specified at the class level infer the values that are supplied for the array literal as type **Object**.

You can explicitly specify the type of the elements in an array that's created by using an array literal. In this case, the values in the array literal must widen to the type of the elements of the array. The following code example creates an array of type **Double** from a list of integers.

VB

```
Dim values As Double() = {1, 2, 3, 4, 5, 6}
```

Nested Array Literals

You can create a multidimensional array by using nested array literals. Nested array literals must have a dimension and number of dimensions, or rank, that's consistent with the resulting array. The following code example creates a two-dimensional array of integers by using an array literal.

VB

```
Dim grid = {{1, 2}, {3, 4}}
```

In the previous example, an error would occur if the number of elements in the nested array literals didn't match. An error would also occur if you explicitly declared the array variable to be other than two-dimensional.

Note

You can avoid an error when you supply nested array literals of different dimensions by enclosing the inner array literals in parentheses. The parentheses force the array literal expression to be evaluated, and the resulting values are used with the outer array literal, as the following code shows.

VB

```
Dim values = {({1, 2}), ({3, 4, 5})}
```

When you create a multidimensional array by using nested array literals, you can use type inference. When you use type inference, the inferred type is the dominant type for all the values in all the array literals for a nesting level. The following code example creates a two-dimensional array of type **Double** from values that are of type **Integer** and **Double**.

VB

```
Dim a = {{1, 2.0}, {3, 4}, {5, 6}, {7, 8}}
```

For additional examples, see How to: Initialize an Array Variable in Visual Basic.

Iterating Through an Array

When you iterate through an array, you access each element in the array from the lowest index to the highest index.

The following example iterates through a one-dimensional array by using the For...Next Statement (Visual Basic). The GetUpperBound method returns the highest value that the index can have. The lowest index value is always 0.

VB

```
Dim numbers = {10, 20, 30}

For index = 0 To numbers.GetUpperBound(0)
    Debug.WriteLine(numbers(index))
Next
' Output:
'   10
'   20
'   30
```

The following example iterates through a multidimensional array by using a **For...Next** statement. The GetUpperBound method has a parameter that specifies the dimension. `GetUpperBound(0)` returns the high index value for the first dimension, and `GetUpperBound(1)` returns the high index value for the second dimension.

VB

```
Dim numbers = {{1, 2}, {3, 4}, {5, 6}}

For index0 = 0 To numbers.GetUpperBound(0)
    For index1 = 0 To numbers.GetUpperBound(1)
        Debug.Write(numbers(index0, index1).ToString & " ")
    Next
    Debug.WriteLine("")
Next
' Output
'   1 2
'   3 4
'   5 6
```

The following example iterates through a one-dimensional array by using a For Each...Next Statement (Visual Basic).

VB

```
Dim numbers = {10, 20, 30}

For Each number In numbers
    Debug.WriteLine(number)
Next
' Output:
'   10
'   20
'   30
```

The following example iterates through a multidimensional array by using a **For Each...Next** statement. However, you have more control over the elements of a multidimensional array if you use a nested **For...Next** statement, as in a previous example, instead of a **For Each...Next** statement.

VB

```
Dim numbers = {{1, 2}, {3, 4}, {5, 6}}

For Each number In numbers
    Debug.WriteLine(number)
Next
' Output:
'   1
'   2
'   3
'   4
'   5
'   6
```

Arrays as Return Values and Parameters

To return an array from a **Function** procedure, specify the array data type and the number of dimensions as the return type of the Function Statement (Visual Basic). Within the function, declare a local array variable with same data type and number of dimensions. In the Return Statement (Visual Basic), include the local array variable without parentheses.

To specify an array as a parameter to a **Sub** or **Function** procedure, define the parameter as an array with a specified data type and number of dimensions. In the call to the procedure, send an array variable with the same data type and number of dimensions.

In the following example, the `GetNumbers` function returns an `Integer()`. This array type is a one dimensional array of type **Integer**. The `ShowNumbers` procedure accepts an `Integer()` argument.

VB

```
Public Sub Process()
    Dim numbers As Integer() = GetNumbers()
    ShowNumbers(numbers)
End Sub
```

```
Private Function GetNumbers() As Integer()
    Dim numbers As Integer() = {10, 20, 30}
    Return numbers
End Function

Private Sub ShowNumbers(numbers As Integer())
    For index = 0 To numbers.GetUpperBound(0)
        Debug.WriteLine(numbers(index) & " ")
    Next
End Sub

' Output:
'   10
'   20
'   30
```

In the following example, the `GetNumbersMultiDim` function returns an `Integer(,)`. This array type is a two dimensional array of type **Integer**. The `ShowNumbersMultiDim` procedure accepts an `Integer(,)` argument.

VB

```
Public Sub ProcessMultidim()
    Dim numbers As Integer(,) = GetNumbersMultidim()
    ShowNumbersMultidim(numbers)
End Sub

Private Function GetNumbersMultidim() As Integer(,)
    Dim numbers As Integer(,) = {{1, 2}, {3, 4}, {5, 6}}
    Return numbers
End Function

Private Sub ShowNumbersMultidim(numbers As Integer(,))
    For index0 = 0 To numbers.GetUpperBound(0)
        For index1 = 0 To numbers.GetUpperBound(1)
            Debug.Write(numbers(index0, index1).ToString & " ")
        Next
        Debug.WriteLine("")
    Next
End Sub

' Output
'  1 2
'  3 4
'  5 6
```

Jagged Arrays

An array that holds other arrays as elements is known as an array of arrays or a jagged array. A jagged array and each element in a jagged array can have one or more dimensions. Sometimes the data structure in your application is

two-dimensional but not rectangular.

The following example has an array of months, each element of which is an array of days. Because different months have different numbers of days, the elements don't form a rectangular two-dimensional array. Therefore, a jagged array is used instead of a multidimensional array.

VB

```
' Declare the jagged array.
' The New clause sets the array variable to a 12-element
' array. Each element is an array of Double elements.
Dim sales()() As Double = New Double(11)() {}

' Set each element of the sales array to a Double
' array of the appropriate size.
For month As Integer = 0 To 11
    Dim days As Integer =
        DateTime.DaysInMonth(Year(Now), month + 1)
    sales(month) = New Double(days - 1) {}
Next month

' Store values in each element.
For month As Integer = 0 To 11
    Dim upper = sales(month).GetUpperBound(0)
    For day = 0 To upper
        sales(month)(day) = (month * 100) + day
    Next
Next
```

Zero-Length Arrays

An array that contains no elements is also called a zero-length array. A variable that holds a zero-length array doesn't have the value **Nothing**. To create an array that has no elements, declare one of the array's dimensions to be -1, as the following example shows.

VB

```
Dim twoDimensionalStrings(-1, 3) As String
```

You might need to create a zero-length array under the following circumstances:

- Without risking a NullReferenceException exception, your code must access members of the Array class, such as Length or Rank, or call a Visual Basic function such as UBound.
- You want to keep the consuming code simpler by not having to check for **Nothing** as a special case.
- Your code interacts with an application programming interface (API) that either requires you to pass a zero-length array to one or more procedures or returns a zero-length array from one or more procedures.

Array Size

The size of an array is the product of the lengths of all its dimensions. It represents the total number of elements currently contained in the array.

The following example declares a three-dimensional array.

```
Dim prices(3, 4, 5) As Long
```

The overall size of the array in variable `prices` is *(3 + 1) x (4 + 1) x (5 + 1) = 120*.

You can find the size of an array by using the Length property. You can find the length of each dimension of a multi-dimensional array by using the GetLength method.

You can resize an array variable by assigning a new array object to it or by using the **ReDim** statement.

There are several things to keep in mind when dealing with the size of an array.

Dimension Length	The index of each dimension is 0-based, which means it ranges from 0 through its upper bound. Therefore, the length of a given dimension is greater by 1 than the declared upper bound for that dimension.
Length Limits	The length of every dimension of an array is limited to the maximum value of the **Integer** data type, which is *(2 ^ 31) - 1*. However, the total size of an array is also limited by the memory available on your system. If you attempt to initialize an array that exceeds the amount of available RAM, the common language runtime throws an OutOfMemoryException exception.

Size and Element Size	An array's size is independent of the data type of its elements. The size always represents the total number of elements, not the number of bytes that they consume in storage.
Memory Consumption	It is not safe to make any assumptions regarding how an array is stored in memory. Storage varies on platforms of different data widths, so the same array can consume more memory on a 64-bit system than on a 32-bit system. Depending on system configuration when you initialize an array, the common language runtime (CLR) can assign storage either to pack elements as close together as possible, or to align them all on natural hardware boundaries. Also, an array requires a storage overhead for its control information, and this overhead increases with each added dimension.

Array Types and Other Types

Every array has a data type, but it differs from the data type of its elements. There is no single data type for all arrays. Instead, the data type of an array is determined by the number of dimensions, or *rank*, of the array, and the data type of the elements in the array. Two array variables are considered to be of the same data type only when they have the same rank and their elements have the same data type. The lengths of the dimensions in an array do not influence the array data type.

Every array inherits from the System.Array class, and you can declare a variable to be of type **Array**, but you cannot create an array of type **Array**. Also, the ReDim Statement (Visual Basic) cannot operate on a variable declared as type **Array**. For these reasons, and for type safety, it is advisable to declare every array as a specific type, such as **Integer** in the preceding example.

You can find out the data type of either an array or its elements in several ways.

- You can call the Object.GetType method on the variable to receive a Type object for the run-time type of the variable. The Type object holds extensive information in its properties and methods.
- You can pass the variable to the TypeName function to receive a **String** containing the name of run-time type.
- You can pass the variable to the VarType function to receive a **VariantType** value representing the type classification of the variable.

The following example calls the **TypeName** function to determine the type of the array and the type of the elements in the array. The array type is **Integer(,)** and the elements in the array are of type **Integer**.

VB

```
Dim thisTwoDimArray(,) As Integer = New Integer(9, 9) {}
MsgBox("Type of thisTwoDimArray is " & TypeName(thisTwoDimArray))
MsgBox("Type of thisTwoDimArray(0, 0) is " & TypeName(thisTwoDimArray(0, 0)))
```

Collections as an Alternative to Arrays

Arrays are most useful for creating and working with a fixed number of strongly typed objects. Collections provide a more

flexible way to work with groups of objects. Unlike arrays, the group of objects that you work with can grow and shrink dynamically as the needs of the application change.

If you need to change the size of an array, you must use the ReDim Statement (Visual Basic). When you do this, Visual Basic creates a new array and releases the previous array for disposal. This takes execution time. Therefore, if the number of items you are working with changes frequently, or you cannot predict the maximum number of items you need, you might obtain better performance using a collection.

For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

For more information about collections, see Collections (C# and Visual Basic).

Example

The following example uses the .NET Framework generic class System.Collections.Generic.List(Of T) to create a list collection of `Customer` objects.

VB

```
' Define the class for a customer.
Public Class Customer
    Public Property Name As String
    ' Insert code for other members of customer structure.
End Class

' Create a module-level collection that can hold 200 elements.
Public CustomerList As New List(Of Customer)(200)

' Add a specified customer to the collection.
Private Sub AddNewCustomer(ByVal newCust As Customer)
    ' Insert code to perform validity check on newCust.
    CustomerList.Add(newCust)
End Sub

' Display the list of customers in the Debug window.
Private Sub PrintCustomers()
    For Each cust As Customer In CustomerList
        Debug.WriteLine(cust)
    Next cust
End Sub
```

The declaration of the `CustomerFile` collection specifies that it can contain elements only of type `Customer`. It also provides for an initial capacity of 200 elements. The procedure `AddNewCustomer` checks the new element for validity and then adds it to the collection. The procedure `PrintCustomers` uses a **For Each** loop to traverse the collection and display its elements.

Related Topics

Term	Definition
Array Dimensions in Visual Basic	Explains rank and dimensions in arrays.
How to: Initialize an Array Variable in Visual Basic	Describes how to populate arrays with initial values.
How to: Sort An Array in Visual Basic	Shows how to sort the elements of an array alphabetically.
How to: Assign One Array to Another Array (Visual Basic)	Describes the rules and steps for assigning an array to another array variable.
Troubleshooting Arrays (Visual Basic)	Discusses some common problems that arise when working with arrays.

See Also

Array
Dim Statement (Visual Basic)
ReDim Statement (Visual Basic)

Array Dimensions in Visual Basic

Visual Studio 2015

A *dimension* is a direction in which you can vary the specification of an array's elements. An array that holds the sales total for each day of the month has one dimension (the day of the month). An array that holds the sales total by department for each day of the month has two dimensions (the department number and the day of the month). The number of dimensions an array has is called its *rank*.

Note
You can use the Rank property to determine the how many dimensions an array has.

Working with Dimensions

You specify an element of an array by supplying an *index* or *subscript* for each of its dimensions. The elements are contiguous along each dimension from index 0 through the highest index for that dimension.

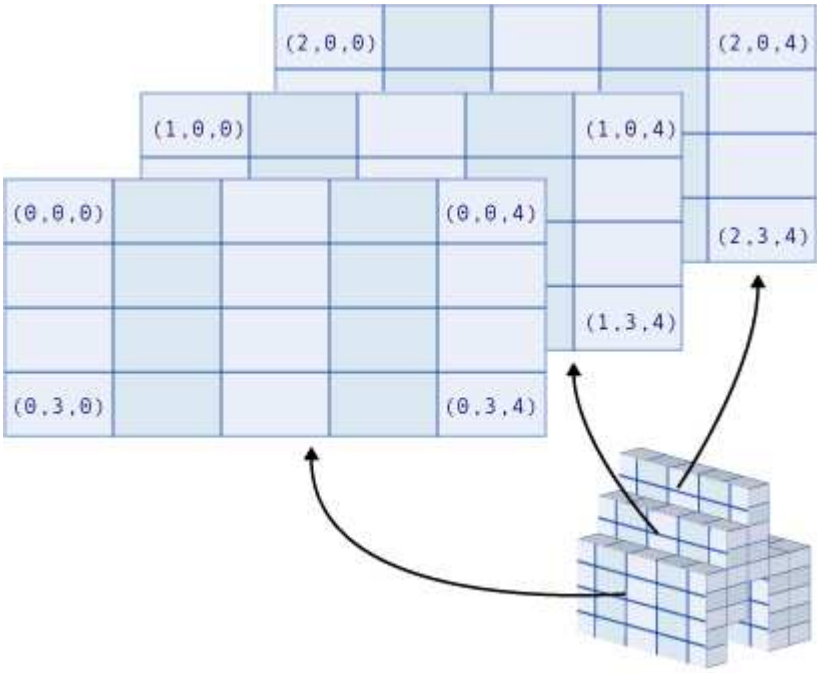
The following illustrations show the conceptual structure of arrays with different ranks. Each element in the illustrations shows the index values that access it. For example, you can access the first element of the second row of the two-dimensional array by specifying indexes `(1, 0)`.

(0)	(1)	(2)	(3)	(4)

One-dimensional array

(0,0)				(0,4)
(1,0)				
(3,0)				(3,4)

Two-dimensional array



Three-dimensional array

One Dimension

Many arrays have only one dimension, such as the number of people of each age. The only requirement to specify an element is the age for which that element holds the count. Therefore, such an array uses only one index. The following example declares a variable to hold a *one-dimensional array* of age counts for ages 0 through 120.

```
Dim ageCounts(120) As UInteger
```

Two Dimensions

Some arrays have two dimensions, such as the number of offices on each floor of each building on a campus. The specification of an element requires both the building number and the floor, and each element holds the count for that combination of building and floor. Therefore, such an array uses two indexes. The following example declares a variable to hold a *two-dimensional array* of office counts, for buildings 0 through 40 and floors 0 through 5.

```
Dim officeCounts(40, 5) As Byte
```

A two-dimensional array is also called a *rectangular array*.

Three Dimensions

A few arrays have three dimensions, such as values in three-dimensional space. Such an array uses three indexes, which in this case represent the *x*, *y*, and *z* coordinates of physical space. The following example declares a variable to hold a

three-dimensional array of air temperatures at various points in a three-dimensional volume.

```
Dim airTemperatures(99, 99, 24) As Single
```

More than Three Dimensions

Although an array can have as many as 32 dimensions, it is rare to have more than three.

Note

When you add dimensions to an array, the total storage needed by the array increases considerably, so use multidimensional arrays with care.

Using Different Dimensions

Suppose you want to track sales amounts for every day of the present month. You might declare a one-dimensional array with 31 elements, one for each day of the month, as the following example shows.

```
Dim salesAmounts(30) As Double
```

Now suppose you want to track the same information not only for every day of a month but also for every month of the year. You might declare a two-dimensional array with 12 rows (for the months) and 31 columns (for the days), as the following example shows.

```
Dim salesAmounts(11, 30) As Double
```

Now suppose you decide to have your array hold information for more than one year. If you want to track sales amounts for 5 years, you could declare a three-dimensional array with 5 layers, 12 rows, and 31 columns, as the following example shows.

```
Dim salesAmounts(4, 11, 30) As Double
```

Note that, because each index varies from 0 to its maximum, each dimension of `salesAmounts` is declared as one less than the required length for that dimension. Note also that the size of the array increases with each new dimension. The three sizes in the preceding examples are 31, 372, and 1,860 elements respectively.

Note

You can create an array without using the **Dim** statement or the **New** clause. For example, you can call the CreateInstance method, or another component can pass your code an array created in this manner. Such an array can have a lower bound other than 0. You can always test for the lower bound of a dimension by using the GetLowerBound method or the **LBound** function.

See Also

Arrays in Visual Basic
Troubleshooting Arrays (Visual Basic)

How to: Initialize an Array Variable in Visual Basic

Visual Studio 2015

You initialize an array variable by including an array literal in a **New** clause and specifying the initial values of the array. You can either specify the type or allow it to be inferred from the values in the array literal. For more information about how the type is inferred, see "Populating an Array with Initial Values" in Arrays in Visual Basic.

To initialize an array variable by using an array literal

- Either in the **New** clause, or when you assign the array value, supply the element values inside braces (**{}**). The following example shows several ways to declare, create, and initialize a variable to contain an array that has elements of type **Char**.

VB

```
' The following five lines of code create the same array.
' Preferred syntaxes are on the lines with chars1 and chars2.
Dim chars1 = {"%"c, "&"c, "@"c}
Dim chars2 As Char() = {"%"c, "&"c, "@"c}

Dim chars3() As Char = {"%"c, "&"c, "@"c}
Dim chars4 As Char() = New Char(2) {"%"c, "&"c, "@"c}
Dim chars5() As Char = New Char(2) {"%"c, "&"c, "@"c}
```

After each statement executes, the array that's created has a length of 3, with elements at index 0 through index 2 containing the initial values. If you supply both the upper bound and the values, you must include a value for every element from index 0 through the upper bound.

Notice that you do not have to specify the index upper bound if you supply element values in an array literal. If no upper bound is specified, the size of the array is inferred based on the number of values in the array literal.

To initialize a multidimensional array variable by using array literals

- Nest values inside braces (**{}**) within braces. Ensure that the nested array literals all infer as arrays of the same type and length. The following code example shows several examples of multidimensional array initialization.

VB

```
Dim numbers = {{1, 2}, {3, 4}, {5, 6}}
Dim customerData = {{"City Power & Light", "http://www.cpandl.com/"},
                    {"Wide World Importers", "http://wideworldimporters.com"},
                    {"Lucerne Publishing", "http://www.lucernepublishing.com"}}
```

```
' You can nest array literals to create arrays that have more than two
' dimensions.
Dim twoSidedCube = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}
```

- You can explicitly specify the array bounds, or leave them out and have the compiler infer the array bounds based on the values in the array literal. If you supply both the upper bounds and the values, you must include a value for every element from index 0 through the upper bound in every dimension. The following example shows several ways to declare, create, and initialize a variable to contain a two-dimensional array that has elements of type **Short**

VB

```
' The following five lines of code create the same array.
' Preferred syntaxes are on the lines with scores1 and scores2.
Dim scores1 = {{10S, 10S, 10S}, {10S, 10S, 10S}}
Dim scores2 As Short(,) = {{10, 10, 10}, {10, 10, 10}}

Dim scores3(,) As Short = {{10, 10, 10}, {10, 10, 10}}
Dim scores4 As Short(,) = New Short(1, 2) {{10, 10, 10}, {10, 10, 10}}
Dim scores5(,) As Short = New Short(1, 2) {{10, 10, 10}, {10, 10, 10}}
```

After each statement executes, the created array contains six initialized elements that have indexes `(0,0)`, `(0,1)`, `(0,2)`, `(1,0)`, `(1,1)`, and `(1,2)`. Each array location contains the value `10`.

- The following example iterates through a multidimensional array. In a Windows console application that is written in Visual Basic, paste the code inside the `Sub Main()` method. The last comments show the output.

VB

```
Dim numbers = {{1, 2}, {3, 4}, {5, 6}}

' Iterate through the array.
For index0 = 0 To numbers.GetUpperBound(0)
    For index1 = 0 To numbers.GetUpperBound(1)
        Debug.Write(numbers(index0, index1).ToString & " ")
    Next
    Debug.WriteLine("")
Next
' Output
'  1 2
'  3 4
'  5 6
```

To initialize a jagged array variable by using array literals

- Nest object values inside braces (**{}**). Although you can also nest array literals that specify arrays of different lengths, in the case of a jagged array, make sure that that the nested array literals are enclosed in parentheses (**()**). The parentheses force the evaluation of the nested array literals, and the resulting arrays are used as the initial values of the jagged array. The following code example shows two examples of jagged array initialization.

VB

```
' Create a jagged array of arrays that have different lengths.
Dim jaggedNumbers = {({1, 2, 3}), ({4, 5}), ({6}), ({7})}

' Create a jagged array of Byte arrays.
Dim images = {New Byte() {}, New Byte() {}, New Byte() {}}
```

- The following example iterates through a jagged array. In a Windows console application that is written in Visual Basic, paste the code inside the `Sub Main()` method. The last comments in the code show the output.

VB

```
' Create a jagged array of arrays that have different lengths.
Dim jaggedNumbers = {({1, 2, 3}), ({4, 5}), ({6}), ({7})}

For indexA = 0 To jaggedNumbers.Length - 1
    For indexB = 0 To jaggedNumbers(indexA).Length - 1
        Debug.Write(jaggedNumbers(indexA)(indexB) & " ")
    Next
    Debug.WriteLine("")
Next

' Output:
'  1 2 3
'  4 5
'  6
'  7
```

See Also

Arrays in Visual Basic
Troubleshooting Arrays (Visual Basic)

How to: Sort An Array in Visual Basic

Visual Studio 2015

This example declares an array of **String** objects named `zooAnimals`, populates it, and then sorts it alphabetically.

Example

```
Private Sub sortAnimals()
    Dim zooAnimals(2) As String
    zooAnimals(0) = "lion"
    zooAnimals(1) = "turtle"
    zooAnimals(2) = "ostrich"
    Array.Sort(zooAnimals)
End Sub
```

Compiling the Code

This example requires:

- Access to Mscorlib.dll and the System namespace.

Robust Programming

The following conditions may cause an exception:

- Array is empty (ArgumentNullException class)
- Array is multidimensional (RankException class)
- One or more elements of the array do not implement the IComparable interface (InvalidOperationException class)

See Also

Array.Sort
Arrays in Visual Basic
Troubleshooting Arrays (Visual Basic)
Collections (C# and Visual Basic)
For Each...Next Statement (Visual Basic)

Troubleshooting Arrays (Visual Basic)

Visual Studio 2015

This page lists some common problems that can occur when working with arrays.

Compilation Errors Declaring and Initializing an Array

Compilation errors can arise from misunderstanding of the rules for declaring, creating, and initializing arrays. The most common causes of errors are the following:

- Supplying a New Operator (Visual Basic) clause after specifying dimension lengths in the array variable declaration. The following code lines show invalid declarations of this type.

```
Dim INVALIDsingleDimByteArray(2) As Byte = New Byte()

Dim INVALIDtwoDimShortArray(1, 1) As Short = New Short(,)

Dim INVALIDjaggedByteArray(1)() As Byte = New Byte()()
```

- Specifying dimension lengths for more than the top-level array of a jagged array. The following code line shows an invalid declaration of this type.

```
Dim INVALIDjaggedByteArray(1)(1) As Byte
```

- Omitting the **New** keyword when specifying the element values. The following code line shows an invalid declaration of this type.

```
Dim INVALIDoneDimShortArray() As Short = Short() {0, 1, 2, 3}
```

- Supplying a **New** clause without braces (**{}**). The following code lines show invalid declarations of this type.

```
Dim INVALIDsingleDimByteArray() As Byte = New Byte()

Dim INVALIDsingleDimByteArray() As Byte = New Byte(2)

Dim INVALIDtwoDimShortArray(,) As Short = New Short(,)

Dim INVALIDtwoDimShortArray(,) As Short = New Short(1, 1)
```

Accessing an Array Out of Bounds

The process of initializing an array assigns an upper bound and a lower bound to each dimension. Every access to an element of the array must specify a valid index, or subscript, for every dimension. If any index is below its lower bound or above its upper bound, an IndexOutOfRangeException exception results. The compiler cannot detect such an error, so an

error occurs at run time.

Determining Bounds

If another component passes an array to your code, for example as a procedure argument, you do not know the size of that array or the lengths of its dimensions. You should always determine the upper bound for every dimension of an array before you attempt to access any elements. If the array has been created by some means other than a Visual Basic **New** clause, the lower bound might be something other than 0, and it is safest to determine that lower bound as well.

Specifying the Dimension

When determining the bounds of a multidimensional array, take care how you specify the dimension. The *dimension* parameters of the GetLowerBound and GetUpperBound methods are 0-based, while the *Rank* parameters of the Visual Basic LBound and UBound functions are 1-based.

See Also

Arrays in Visual Basic
How to: Initialize an Array Variable in Visual Basic

ReDim Statement (Visual Basic)

Visual Studio 2015

Reallocates storage space for an array variable.

Syntax

```
ReDim [ Preserve ] name(boundlist) [ ,  name(boundlist) [, ... ] ]
```

Parts

Term	Definition
Preserve	Optional. Modifier used to preserve the data in the existing array when you change the size of only the last dimension.
name	Required. Name of the array variable. See Declared Element Names (Visual Basic).
boundlist	Required. List of bounds of each dimension of the redefined array.

Remarks

You can use the **ReDim** statement to change the size of one or more dimensions of an array that has already been declared. If you have a large array and you no longer need some of its elements, **ReDim** can free up memory by reducing the array size. On the other hand, if your array needs more elements, **ReDim** can add them.

The **ReDim** statement is intended only for arrays. It's not valid on scalars (variables that contain only a single value), collections, or structures. Note that if you declare a variable to be of type **Array**, the **ReDim** statement doesn't have sufficient type information to create the new array.

You can use **ReDim** only at procedure level. Therefore, the declaration context for the variable must be a procedure; it can't be a source file, a namespace, an interface, a class, a structure, a module, or a block. For more information, see Declaration Contexts and Default Access Levels (Visual Basic).

Rules

- **Multiple Variables.** You can resize several array variables in the same declaration statement and specify the *name* and *boundlist* parts for each variable. Multiple variables are separated by commas.
- **Array Bounds.** Each entry in *boundlist* can specify the lower and upper bounds of that dimension. The lower bound is always 0 (zero). The upper bound is the highest possible index value for that dimension, not the length of the dimension (which is the upper bound plus one). The index for each dimension can vary from 0 through its upper bound value.

  The number of dimensions in *boundlist* must match the original number of dimensions (rank) of the array.
- **Data Types.** The **ReDim** statement cannot change the data type of an array variable or its elements.
- **Initialization.** The **ReDim** statement cannot provide new initialization values for the array elements.
- **Rank.** The **ReDim** statement cannot change the rank (the number of dimensions) of the array.
- **Resizing with Preserve.** If you use **Preserve**, you can resize only the last dimension of the array. For every other dimension, you must specify the bound of the existing array.

  For example, if your array has only one dimension, you can resize that dimension and still preserve all the contents of the array, because you are changing the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension if you use **Preserve**.
- **Properties.** You can use **ReDim** on a property that holds an array of values.

Behavior

- **Array Replacement. ReDim** releases the existing array and creates a new array with the same rank. The new array replaces the released array in the array variable.
- **Initialization without Preserve.** If you do not specify **Preserve**, **ReDim** initializes the elements of the new array by using the default value for their data type.
- **Initialization with Preserve.** If you specify **Preserve**, Visual Basic copies the elements from the existing array to the new array.

Example

The following example increases the size of the last dimension of a dynamic array without losing any existing data in the array, and then decreases the size with partial data loss. Finally, it decreases the size back to its original value and reinitializes all the array elements.

VB

```
Dim intArray(10, 10, 10) As Integer
ReDim Preserve intArray(10, 10, 20)
ReDim Preserve intArray(10, 10, 15)
ReDim intArray(10, 10, 10)
```

The **Dim** statement creates a new array with three dimensions. Each dimension is declared with a bound of 10, so the array index for each dimension can range from 0 through 10. In the following discussion, the three dimensions are referred to as layer, row, and column.

The first **ReDim** creates a new array which replaces the existing array in variable `intArray`. **ReDim** copies all the elements from the existing array into the new array. It also adds 10 more columns to the end of every row in every layer and initializes the elements in these new columns to 0 (the default value of **Integer**, which is the element type of the array).

The second **ReDim** creates another new array and copies all the elements that fit. However, five columns are lost from the end of every row in every layer. This is not a problem if you have finished using these columns. Reducing the size of a large array can free up memory that you no longer need.

The third **ReDim** creates another new array and removes another five columns from the end of every row in every layer. This time it does not copy any existing elements. This statement reverts the array to its original size. Because the statement doesn't include the **Preserve** modifier, it sets all array elements to their original default values.

For additional examples, see Arrays in Visual Basic.

See Also

IndexOutOfRangeException
Const Statement (Visual Basic)
Dim Statement (Visual Basic)
Erase Statement (Visual Basic)
Nothing (Visual Basic)
Arrays in Visual Basic