Arrays and ArrayLists

Dr.Haitham A. El-Ghareeb

Information Systems Department
Faculty of Computers and Information Sciences
Mansoura University

helghareeb@gmail.com

October 14, 2012

Array vs. ArrayList

Array vs. ArrayList

- Using an array in C# involves creating an array object of System.Array type, the abstract base type for all arrays.

Array vs. ArrayList

- Using an array in C# involves creating an array object of System.Array type, the abstract base type for all arrays.
- Array class provides a set of methods for performing tasks such as sorting and searching

Array vs. ArrayList

- Using an array in C# involves creating an array object of System.Array type, the abstract base type for all arrays.
- Array class provides a set of methods for performing tasks such as sorting and searching
- An interesting alternative to using arrays in C# is the ArrayList class.

Array vs. ArrayList

- Using an array in C# involves creating an array object of System.Array type, the abstract base type for all arrays.
- Array class provides a set of methods for performing tasks such as sorting and searching
- An interesting alternative to using arrays in C# is the ArrayList class.
- An arraylist is an array that grows dynamically as more space is needed.

Array Basics

- Arrays are indexed collections of data.
- The data can be of either a built-in type or a user-defined type.
- Arrays in C# are actually objects themselves because they derive from the System.Array class.
- Since an array is a declared instance of the System.Array class, you have the use of all the methods and properties of this class when using arrays.

Declaring and Initializing Arrays

Arrays are declared using the following syntax:

```
type[ ] array-name;
```

where type is the data type of the array elements. Here is an example:

```
string[ ] names;
```

Declaring and Initializing Arrays (Cont.)

A second line is necessary to instantiate the array (since it is an object of System.Array type) and to determine the size of the array. The following line instantiates the names array just declared:

```
names = new string[10];
```

and reserves memory for ten strings. You can combine these two statements into one line when necessary to do so:

```
string[ ] names = new string[10];
```

Declaring and Initializing Arrays (Cont.)

- There are times when you will want to declare, instantiate, and assign data to an array in one statement.
- You can do this in C# using an initialization list:

```
int[ ] numbers = new int[ ] {1,2,3,4,5};
```

- The list of numbers, called the initialization list, is delimited with curly braces, and each element is delimited with a comma.
- When you declare an array using this technique, you dont have to specify the number of elements.
- The compiler infers this data from the number of items in the initialization list.

Setting and Accessing Array Elements

- Elements are stored in an array either by direct access or by calling the Array class method SetValue.
- Direct access involves referencing an array position by index on the left-hand side of an assignment statement:

```
Names[2] = "Haitham";
professors[19] = 23123;
```

- The SetValue method provides a more object-oriented way to set the value of an array element.
- The method takes two arguments, an index number and the value of the element.

```
names.SetValue(2, "Haitham");
professors.SetValue(19, 23123);
```

Setting and Accessing Array Elements

- Array elements are accessed either by direct access or by calling the GetValue method.
- The GetValue method takes a single argumentan index.

```
myName = names[2];
monthSalary = professors.GetValue(19);
```

- It is common to loop through an array in order to access every array element using a For loop.
- A frequent mistake programmers make when coding the loop is to either hard-code the upper value of the loop (which is a mistake because the upper bound may change if the array is dynamic)

```
for (int i = 0; i <= professors.GetUpperBound(0); i++) {
totalSalary = totalSalary + professors[i];
}
```

Methods and Properties for Retrieving Array Metadata

- The Array class provides several properties for retrieving metadata about an array:
  - Length: Returns the total number of elements in all dimensions of an array.
  - GetLength: Returns the number of elements in specified dimension of an array.
  - Rank: Returns the number of dimensions of an array.
  - GetType: Returns the Type of the current array instance.

Methods and Properties for Retrieving Array Metadata

- The GetType method is used for determining the data type of an array
- such as when the array is passed as an argument to a method.
- In the following code fragment, we create a variable of type Type, which allows us to use call a class method, IsArray, to determine if an object is an array. If the object is an array, then the code returns the data type of the array.

```
int[ ] numbers;
numbers = new int[] {0,1,2,3,4};
Type arrayType = numbers.GetType();
if (arrayType.IsArray) {
Console.WriteLine("The array type is: {0}", arrayType);
} else {
    Console.WriteLine("Not an array");
}
   Console.Read();
}
```

Methods and Properties for Retrieving Array Metadata (Cont.)

- The GetType method returns not only the type of the array, but also lets us know that the object is indeed an array.
- Here is the output from the code: `The array type is: System.Int32[]`
- The brackets indicate the object is an array.
- Also notice that we use a format when displaying the data type.
- We have to do this because we cant convert the Type data to string in order to concatenate it with the rest of the displayed string.

Multidimensional Arrays

- In C#, an array can have up to 32 dimensions, though arrays with more than three dimensions are very rare and very confusing.
- Multidimensional arrays are declared by providing the upper bound of each of the dimensions of the array.
- The two-dimensional declaration: `int[,] grades = new int[4,5];` declares an array that consists of 4 rows and 5 columns.
- Two-dimensional arrays are often used to model matrices.

Multidimensional Arrays (Cont.)

- You can also declare a multidimensional array without specifing the dimension bounds. To do this, you use commas to specify the number of dimensions. For example,
  ```
  double[,] Sales;
  ```
  declares a two-dimensional array, whereas
  ```
  double[,,] sales;
  ```
  declares a three-dimensional array.
- When you declare arrays without providing the upper bounds of the dimensions, you have to later redimension the array with those bounds:
  ```
  sales = new double[4,5];
  ```

Multidimensional Arrays (Cont.)

- Multidimensional arrays can be initialized with an initialization list. Look at the following statement:

```
Int[,] grades = new int[,] {{1, 82, 74, 89, 100},
{2, 93, 96, 85, 86},
{3, 83, 72, 95, 89},
{4, 91, 98, 79, 88}}
```

- Upper bounds of the array are not specified.
- When you initialize an array with an initialization list, you cant specify the bounds of the array.
- The compiler computes the upper bounds of each dimension from the data in the initialization list.
- The initialization list itself is demarked with curly braces, as is each row of the array.
- Each element in the row is delimited with a comma.

Multidimensional Arrays (Cont.)

- Accessing the elements of a multidimensional array is similar to accessing the elements of a one-dimensional array. You can use the traditional array access technique,
  `grade = Grades[2,2];`
  `Grades(2,2) = 99` or you can use the methods of the Array class:
  `grade = Grades.GetValue[0,2]`

Multidimensional Arrays (Cont.)

- You cant use the SetValue method with a multidimensional array because the method only accepts two arguments: a value and a single index.
- It is a common operation to perform calculations on all the elements of a multidimensional array.
- Using the Grades array, if each row of the array is a student record, we can calculate the grade average for each student as follows:

Multidimensional Arrays (Cont.)

```
int[,] grades = new int[,] {{1, 82, 74, 89, 100},
{2, 93, 96, 85, 86},
{3, 83, 72, 95, 89},
{4, 91, 98, 79, 88}};
int last_grade = grades.GetUpperBound(1);
double average = 0.0;
int total;
int last_student = grades.GetUpperBound(0);
for(int row = 0; row <= last_student; row++) {
    total = 0;
    for (int col = 0; col <= last_grade; col++)
      total += grades[row, col];
      average = total / last_grade;
      Console.WriteLine("Average: " + average);
}
```

Parameter Arrays

- Most method definitions require that a set number of parameters be provided to the method, but there are times when you want to write a method definition that allows an optional number of parameters.
- You can do this using a construct called a parameter array.
- A parameter array is specified in the parameter list of a method definition by using the keyword ParamArray.
- The following method definition allows any amount of numbers to be supplied as parameters, with the total of the numbers returned from the method:

Parameter Arrays (Cont.)

```
static int sumNums(params int[] nums) {
int sum = 0;
for(int i = 0; i <= nums.GetUpperBound(0); i++)
      sum += nums[i];
    return sum;
}
```

- This method will work with the either of the following calls:
  `total = sumNums(1, 2, 3);`
  `total = sumNums(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);`
- When you define a method using a parameter array, the parameter array arguments have to be supplied last in the parameter list in order for the compiler to be able to process the list of parameters correctly.
- Otherwise, the compiler wouldnt know the ending point of the parameter array elements and the beginning of other parameters of the method.

Jagged Arrays

- A jagged array is an array of arrays where each row of an array is made up of an array.
- Each dimension of a jagged array is a one-dimensional array.
- We call it a jagged array because the number of elements in each row may be different.
- A picture of a jagged array would not be square or rectangular, but would have uneven or jagged edges.
- A jagged array is declared by putting two sets of parentheses after the array variable name.
  - The first set of parentheses indicates the number of rows in the array.
  - The second set of parentheses is left blank. This marks the place for the one-dimensional array that is stored in each row.

Jagged Arrays (Cont.)

- Normally, the number of rows is set in an initialization list in the declaration statement, like this:
  ```
  int[][] jagged = new int[12][];
  ```
- jagged is an Integer array of 12 elements, where each of the elements is also an Integer array.
- The initialization list is actually just the initialization for the rows of the array, indicating that each row element is an array of 12 elements, with each element initialized to the default value.
- Once the jagged array is declared, the elements of the individual row arrays can be assigned values.
  ```
  jagged[0][0] = 23;
  jagged[0][1] = 13;
  ...
  jagged[7][5] = 45;
  ```
- The first set of parentheses indicates the row number and the second set indicates the element of the row array.

The ArrayList Class

- Static arrays are not very useful when the size of an array is unknown in advance or is likely to change during the lifetime of a program.
- One solution to this problem is to use a type of array that automatically resizes itself when the array is out of storage space.
- This array is called an ArrayList and it is part of the System.Collections namespace in the .NET Framework library.
- An ArrayList object has a Capacity property that stores its size.
- The initial value of the property is 16.

The ArrayList Class (Cont.)

- When the number of elements in an ArrayList reaches this limit, the Capacity property adds another 16 elements to the storage space of the ArrayList.
- Using an ArrayList in a situation where the number of elements in an array can grow larger, or smaller, can be more efficient than using Array.Resize Preserver with a standard array.
- ArrayList stores objects using the Object type.
- If you need a strongly typed array, you should use a standard array or some other data structure.

Example of Using ArrayList Class

```
using System;
using System.Collections;
public class SamplesArrayList  {

   public static void Main()  {

      // Creates and initializes a new ArrayList.
      ArrayList myAL = new ArrayList();
      myAL.Add("Hello");
      myAL.Add("World");
      myAL.Add("!");

      // Displays the properties and values of the ArrayList.
      Console.WriteLine( "myAL" );
      Console.WriteLine( "\tCount:    {0}", myAL.Count );
      Console.WriteLine( "\tCapacity: {0}", myAL.Capacity );
      Console.Write( "\tValues:" );
      PrintValues( myAL );
   }
}
```

ArrayList Class Propoerties

Here is a list of some of the most commonly used methods and properties:

- Capacity: Gets or sets the number of elements that the ArrayList can contain.
- Count: Gets the number of elements actually contained in the ArrayList.
- IsFixedSize: Gets a value indicating whether the ArrayList has a fixed size.
- IsReadOnly: Gets a value indicating whether the ArrayList is read-only.
- IsSynchronized: Gets a value indicating whether access to the ArrayList is synchronized (thread-safe).
- Item: Gets or sets the element at the specified index. In C#, this property is the indexer for the ArrayList class.
- SyncRoot: Gets an object that can be used to synchronize access to the ArrayList.

ArrayList Class Methods

- Adapter: Creates an ArrayList wrapper for a specific IList.
- Add: Adds an object to the end of the ArrayList.
- AddRange: Adds the elements of an ICollection to the end of the ArrayList.
- BinarySearch: Overloaded. Uses a binary search algorithm to locate a specific element in the sorted ArrayList or a portion of it.
- Clear: Removes all elements from the ArrayList.
- Clone: Creates a shallow copy of the ArrayList.
- Contains: Determines whether an element is in the ArrayList.
- CopyTo: Overloaded. Copies the ArrayList or a portion of it to a one-dimensional array.
- Equals (inherited from Object): Overloaded. Determines whether two Object instances are equal.
- FixedSize: Overloaded. Returns a list wrapper with a fixed size, where elements are allowed to be modified, but not added or removed.

ArrayList Class Methods

- GetEnumerator: Overloaded. Returns an enumerator that can iterate through the ArrayList.
- GetHashCode (inherited from Object): Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
- GetRange: Returns an ArrayList which represents a subset of the elements in the source ArrayList.
- GetType (inherited from Object): Gets the Type of the current instance.
- IndexOf: Overloaded. Returns the zero-based index of the first occurrence of a value in the ArrayList or in a portion of it.
- Insert: Inserts an element into the ArrayList at the specified index.
- InsertRange: Inserts the elements of a collection into the ArrayList at the specified index.
- LastIndexOf: Overloaded. Returns the zero-based index of the last occurrence of a value in the ArrayList or in a portion of it.

ArrayList Class Methods

- ReadOnly: Overloaded. Returns a list wrapper that is read-only.
- Remove: Removes the first occurrence of a specific object from the ArrayList.
- RemoveAt: Removes the element at the specified index of the ArrayList.
- RemoveRange: Removes a range of elements from the ArrayList.
- Repeat: Returns an ArrayList whose elements are copies of the specified value.
- Reverse: Overloaded. Reverses the order of the elements in the ArrayList or a portion of it.
- SetRange: Copies the elements of a collection over a range of elements in the ArrayList.

ArrayList Class Methods

- Sort: Overloaded. Sorts the elements in the ArrayList or a portion of it.
- Synchronized: Overloaded. Returns a list wrapper that is synchronized (thread-safe).
- ToArray: Overloaded. Copies the elements of the ArrayList to a new array.
- ToString (inherited from Object): Returns a String that represents the current Object.
- TrimToSize: Sets the capacity to the actual number of elements in the ArrayList.
- Finalize (inherited from Object): Overridden. Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. In C# and C++, finalizers are expressed using destructor syntax.
- MemberwiseClone (inherited from Object): Supported by the .NET Compact Framework. Creates a shallow copy of the current Object.

Memory Management of Arrays

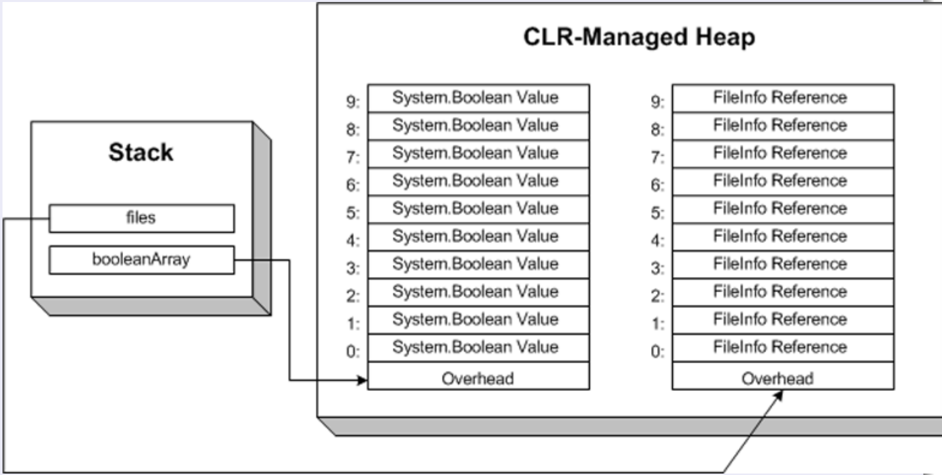

Figure: The contents of an array are laid out contiguously in the managed heap.

Memory Management of Arrays (Cont.)

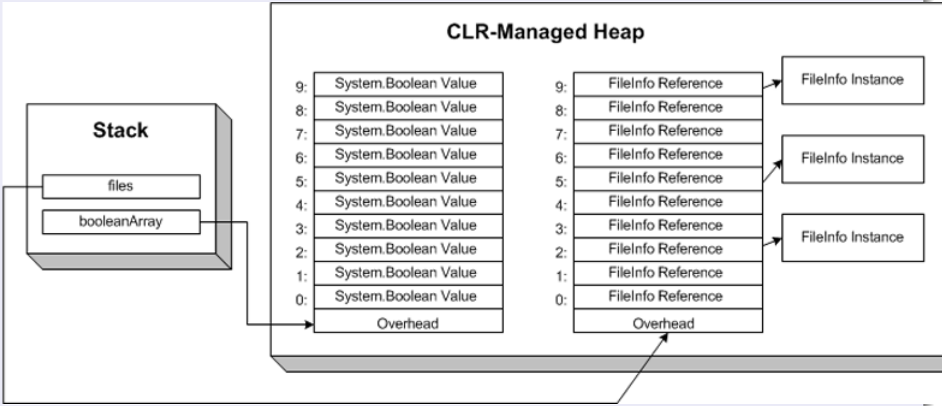

Figure: The contents of an array are laid out contiguously in the managed heap.

Memory Management of ArrayList

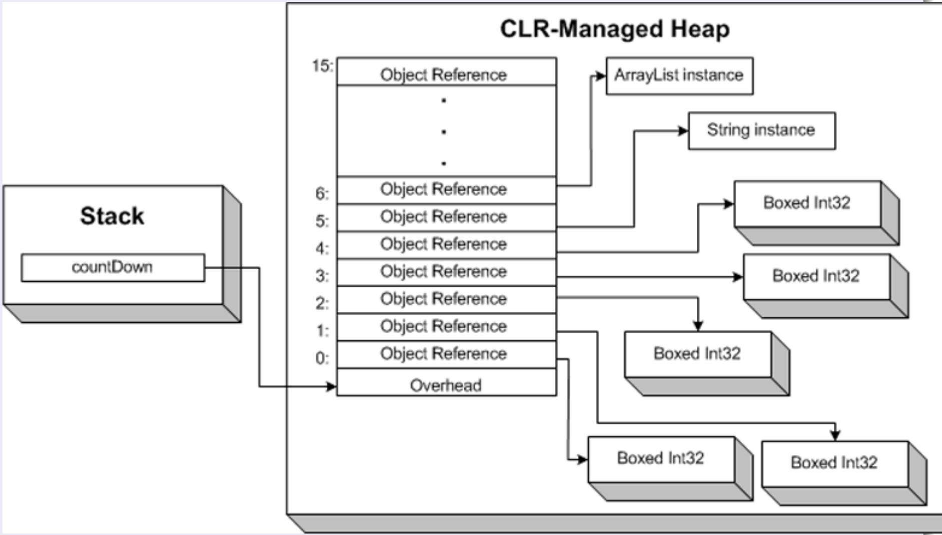

Figure: The ArrayList contains a contiguous block of object references.