An Extensive Examination of Data Structures Using C# 2.0

Visual Studio 2005

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This article kicks off a six-part article series that focuses on important data structures and their use in application development. We'll examine both built-in data structures present in the .NET Framework, as well as essential data structures we'll build ourselves. This first part focuses on an introduction to data structures, defining what data structures are, how the efficiency of data structures are analyzed, and why this analysis is important. In this article, we'll also examine two of the most commonly used data structures present in the .NET Framework: the Array and List. (14 printed pages)

> **Editor's note** This six-part article series originally appeared on MSDN Online starting in November 2003. In January 2005 it was updated to take advantage of the new data structures and features available with the .NET Framework version 2.0, and C# 2.0. The original articles are still available at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp.

> **Note** This article assumes the reader is familiar with C#.

Contents

Introduction

Welcome to the first in a six-part series on using data structures in .NET 2.0. This article series originally appeared on MSDN Online in October 2003, focusing on the .NET Framework version 1.x, and can be accessed at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp. Version 2.0 of the .NET Framework adds new data structures to the Base Class Library, along with new features, such as Generics, that make creating type-safe data structures much easier than with version 1.x. This revised article series introduces these new .NET Framework data structures and examines using these new language features.

Throughout this article series, we will examine a variety of data structures, some of which are included in the .NET Framework's Base Class Library and others that we'll build ourselves. If you're unfamiliar with the term, *data structures* are classes that are used to organize data and provide various operations upon their data. Probably the most common and well-known data structure is the array, which contains a contiguous collection of data items that can be accessed by an ordinal index.

Before jumping into the content for this article, let's first take a quick peek at the roadmap for this six-part article series, so that you can see what lies ahead.

In this first part of the six-part series, we'll look at why data structures are important, and their effect on the performance of an algorithm. To determine a data structure's effect on performance, we'll need to examine how the various operations performed by a data structure can be rigorously analyzed. Finally, we'll turn our attention to two similar data structures present in the .NET Framework: the Array and the List. Chances are you've used these data structures in past projects. In this article, we'll examine what operations they provide and the efficiency of these operations.

In the Part 2, we'll explore the List's "cousins," the Queue and Stack. Like the List, both the Queue and Stack store a collection of data and are data structures available in the .NET Framework Base Class Library. Unlike a List, from which you can retrieve its elements in any order, Queues and Stacks only allow data to be accessed in a predetermined order. We'll examine some applications of Queues and Stacks, and see how these classes are implemented in the .NET Framework. After examining Queues and Stacks, we'll look at hashtables, which allow for direct access like an ArrayList, but store data indexed by a string key.

While arrays and Lists are ideal for directly accessing and storing contents, when working with large amounts of data, these data structures are often sub-optimal candidates when the data needs to be searched. In Part 3, we'll examine the binary search tree data structure, which is designed to improve the time needed to search a collection of items. Despite the improvement in search time with the binary tree, there are some shortcomings. In Part 4, we'll look at SkipLists, which are a mix between binary trees and linked lists, and address some of the issues inherent in binary trees.

In Part 5, we'll turn our attention to data structures that can be used to represent graphs. A graph is a collection of nodes, with a set of edges connecting the various nodes. For example, a map can be visualized as a graph, with cities as nodes and the highways between them as edged between the nodes. Many real-world problems can be abstractly defined in terms of graphs, thereby making graphs an often-used data structure.

Finally, in Part 6 we'll look at data structures to represent sets and disjoint sets. A set is an unordered collection of items. Disjoint sets are a collection of sets that have no elements in common with one another. Both sets and disjoint sets have many uses in everyday programs, which we'll examine in detail in this final part.

Analyzing the Performance of Data Structures

When thinking about a particular application or programming problem, many developers (myself included) find themselves most interested about writing the algorithm to tackle the problem at hand or adding cool features to the application to enhance the user's experience. Rarely, if ever, will you hear someone excited about what type of data structure they are using. However, the data structures used for a particular algorithm can greatly impact its performance. A very common example is finding an element in a data structure. With an unsorted array, this process takes time proportional to the number of elements in the array. With binary search trees or SkipLists, the time required is logarithmically proportional to the number of elements. When searching sufficiently large amounts of data, the data structure chosen can make a difference in the application's performance that can be visibly measured in seconds or even minutes.

Since the data structure used by an algorithm can greatly affect the algorithm's performance, it is important that there exists a rigorous method by which to compare the efficiency of various data structures. What we, as developers utilizing a data structure, are primarily interested in is how the data structures performance changes as the amount of data stored increases. That is, for each new element stored by the data structure, how are the running times of the data structure's operations effected?

Consider the following scenario: imagine that you are tasked with writing a program that will receive as input an array of strings that contain filenames. Your program's job is to determine whether that array of strings contains any filenames with a specific file extension. One approach to do this would be to scan through the array and set some flag once an XML file was encountered. The code might look like so:

```
public bool DoesExtensionExist(string [] fileNames, string extension)
{
      int i = 0;
      for (i = 0; i < fileNames.Length; i++)
         if (String.Compare(Path.GetExtension(fileNames[i]), extension, true) == 0)
            return true;

      return false;   // If we reach here, we didn't find the extension
   }
}
```

Here we see that, in the worst-case—when there is no file with a specified extension, or when there is such a file but it is the last file in the list—we have to search through each element of the array exactly once. To analyze the array's efficiency at sorting, we must ask ourselves the following: "Assume that I have an array with *n* elements. If I add another element, so the array has *n* + 1 elements, what is the new running time?" (The term "running time," despite its name, does not measure the absolute time it takes the program to run, but rather refers to the number of steps the program must perform to complete the given task at hand. When working with arrays, typically the steps considered are how many array accesses one needs to perform.) Since to search for a value in an array we need to visit, potentially, every array value, if we have *n* + 1 array elements, we might have to perform *n* + 1 checks. That is, the time it takes to search an array is linearly proportional to the number of elements in the array.

This sort of analysis described here is called *asymptotic analysis*, as it examines how the efficiency of a data structure changes as the data structure's size approaches infinity. The notation commonly used in asymptotic analysis is called *big-Oh notation*. The big-Oh notation to describe the performance of searching an unsorted array would be denoted as $O(n)$. The large script O is where the terminology big-Oh notation comes from, and the n indicates that the number of steps required to search an array grows linearly as the size of the array grows.

A more methodical way of computing the asymptotic running time of a block of code is to follow these simple steps:

1. Determine the steps that constitute the algorithm's running time. As aforementioned, with arrays, typically the steps considered are the read and write accesses to the array. For other data structures, the steps might differ. Typically, you want to concern yourself with steps that involve the data structure itself, and not simple, atomic operations performed by the computer. That is, with the block of code above, I analyzed its running time by only bothering to count how many times the array needs to be accessed, and did not bother worrying about the time for creating and initializing variables or the check to see if the two strings were equal.
2. Find the line(s) of code that perform the steps you are interested in counting. Put a 1 next to each of those lines.

3. For each line with a 1 next to it, see if it is in a loop. If so, change the 1 to 1 times the maximum number of repetitions the loop may perform. If you have two or more nested loops, continue the multiplication for each loop.
4. Find the largest single term you have written down. This is the running time.

Let's apply these steps to the block of code above. We've already identified that the steps we're interested in are the number of array accesses. Moving onto step 2 note that there is only one line on which the array, `fileNames`, is being accessed: as a parameter in the `String.Compare()` method, so mark a 1 next to that line. Now, applying step 3 notice that the access to `fileNames` in the `String.Compare()` method occurs within a loop that runs at most *n* times (where *n* is the size of the array). So, scratch out the 1 in the loop and replace it with *n*. This is the largest value of *n*, so the running time is denoted as $O(n)$.

$O(n)$, or linear-time, represents just one of a myriad of possible asymptotic running times. Others include $O(\log_2 n)$, $O(n \log_2 n)$, $O(n^2)$, $O(2^n)$, and so on. Without getting into the gory mathematical details of big-Oh, the lower the term inside the parenthesis for large values of *n*, the better the data structure's operation's performance. For example, an operation that runs in $O(\log n)$ is more efficient than one that runs in $O(n)$ since $\log n < n$.

> **Note** In case you need a quick mathematics refresher, $\log_a b = y$ is just another way to write $a^y = b$. So, $\log_2 4 = 2$, since $2^2 = 4$. Similarly, $\log_2 8 = 3$, since $2^3 = 8$. Clearly, $\log_2 n$ grows much slower than n alone, because when $n = 8$, $\log_2 n = 3$. In Part 3 we'll examine binary search trees whose search operation provides an $O(\log_2 n)$ running time.

Throughout this article series, each time we examine a new data structure and its operations, we'll be certain to compute its asymptotic running time and compare it to the running time for similar operations on other data structures.

Asymptotic Running Time and Real-World Algorithms

The asymptotic running time of an algorithm measures how the performance of the algorithm fares as the number of steps that the algorithm must perform approaches infinity. When the running time for one algorithm is said to be greater than another's, what this means mathematically is that there exists some number of steps such that once this number of steps is exceeded the algorithm with the greater running time will always take longer to execute than the one with the shorter running time. However, for instances with fewer steps, the algorithm with the asymptotically-greater running time may run faster than the one with the shorter running time.

For example, there are a myriad of algorithms for sorting an array that have differing running times. One of the simplest and most naïve sorting algorithms is bubble sort, which uses a pair of nested `for` loops to sort the elements of an array. Bubble sort exhibits a running time of $O(n^2)$ due to the two `for` loops. An alternative sorting algorithm is merge sort, which divides the array into halves and recursively sorts each half. The running time for merge sort is $O(n \log_2 n)$. Asymptotically, merge sort is much more efficient than bubble sort, but for small arrays, bubble sort may be more efficient. Merge sort must not only incur the expense of recursive function calls, but also of recombining the sorted array halves, whereas bubble sort simply loops through the array quadratically, swapping pairs of array values as needed. Overall, merge sort must perform fewer steps, but the steps merge sort has to perform are more expensive than the steps involved in bubble sort. For large arrays, this extra expense per step is negligible, but for smaller arrays, bubble sort may actually be more efficient.

Asymptotic analysis definitely has its place, as the asymptotic running time of two algorithms can show how one algorithm will outperform another when the algorithms are operating on sufficiently sized data. Using only asymptotic analysis to judge the performance of an algorithm, though, is foolhardy, as the actual execution times of

different algorithms depends upon specific implementation factors, such as the amount of data being plugged into the algorithm. When deciding what data structure to employ in a real-world project, consider the asymptotic running time, but also carefully profile your application to ascertain the actual impact on performance your data structure choice bears.

Everyone's Favorite Linear, Direct Access, Homogeneous Data Structure: The Array

Arrays are one of the simplest and most widely used data structures in computer programs. Arrays in any programming language all share a few common properties:

- The contents of an array are stored in contiguous memory.
- All of the elements of an array must be of the same type or of a derived type; hence arrays are referred to as homogeneous data structures.
- Array elements can be directly accessed. With arrays if you know you want to access the i^{th} element, you can simply use one line of code: *arrayName*[*i*].

The common operations performed on arrays are:

- Allocation
- Accessing

In C#, when an array (or any reference type variable) is initially declared, it has a `null` value. That is, the following line of code simply creates a variable named `booleanArray` that equals `null`:

```
bool [] booleanArray;
```

Before we can begin to work with the array, we must create an array instance that can store a specific number of elements. This is accomplished using the following syntax:

```
booleanArray = new bool[10];
```

Or more generically:

```
arrayName = new arrayType[allocationSize];
```

This allocates a contiguous block of memory in the CLR-managed heap large enough to hold the *allocationSize* number of *arrayType*s. If *arrayType* is a value type, then *allocationSize* number of unboxed *arrayType* values are created. If *arrayType* is a reference type, then *allocationSize* number of *arrayType* references are created. (If you are unfamiliar with the difference between reference and value types and the managed heap versus the stack, check out Understanding .NET's Common Type System.)

To help hammer home how the .NET Framework stores the internals of an array, consider the following example:

```
bool [] booleanArray;
FileInfo [] files;
```

```
booleanArray = new bool[10];
files = new FileInfo[10];
```

Here, the `booleanArray` is an array of the value type `System.Boolean`, while the `files` array is an array of a reference type, `System.IO.FileInfo`. Figure 1 shows a depiction of the CLR-managed heap after these four lines of code have executed.

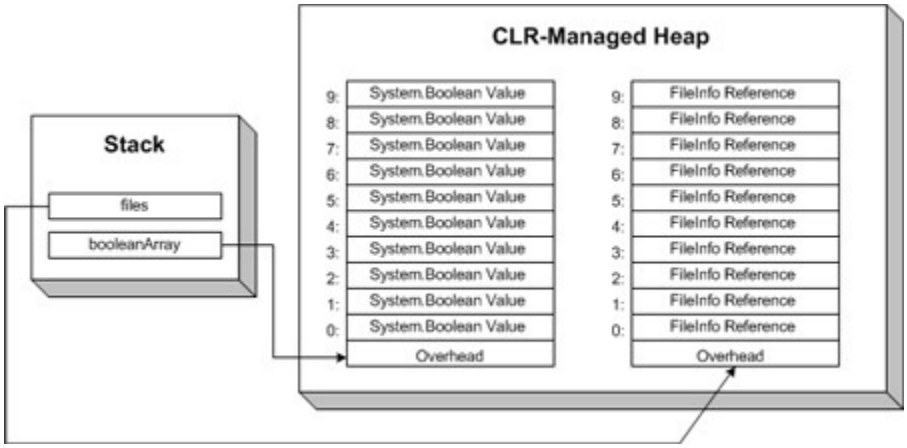

Figure 1. The contents of an array are laid out contiguously in the managed heap.

The thing to keep in mind is that the ten elements in the files array are *references* to `FileInfo` instances. Figure 2 hammers home this point, showing the memory layout if we assign some of the values in the `files` array to `FileInfo` instances.

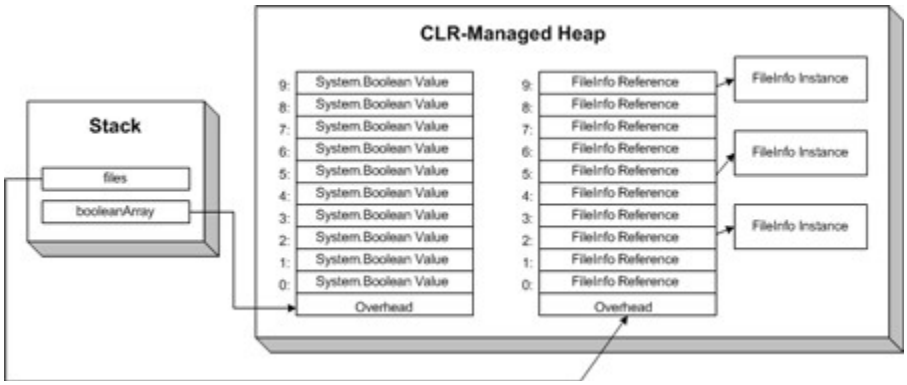

Figure 2. The contents of an array are laid out contiguously in the managed heap.

All arrays in .NET allow their elements to both be read and written to. The syntax for accessing an array element is:

```
// Read an array element
bool b = booleanArray[7];

// Write to an array element
booleanArray[0] = false;
```

The running time of an array access is denoted *O*(1) because it is constant. That is, regardless of how many elements are stored in the array, it takes the same amount of time to lookup an element. This constant running time is possible solely because an array's elements are stored contiguously, hence a lookup only requires knowledge of the array's starting location in memory, the size of each array element, and the element to be indexed.

Realize that in managed code, array lookups are a slight bit more involved than this because with each array access the CLR checks to ensure that the index being requested is within the array's bounds. If the array index specified is out of bounds, an `IndexOutOfRangeException` is thrown. This check help ensures that when stepping through an array we do not accidentally step past the last array index and into some other memory. This check, though, does not affect the asymptotic running time of an array access because the time to perform such checks does not increase as the size of the array increases.

> **Note** This index-bounds check comes at a slight cost of performance for applications that make a large number of array accesses. With a bit of unmanaged code, though, this index out of bounds check can be bypassed. For more information, refer to Chapter 14 of Applied Microsoft .NET Framework Programming by Jeffrey Richter.

When working with an array, you might need to change the number of elements it holds. To do so, you'll need to create a new array instance of the specified size and copy the contents of the old array into the new, resized array. This process can be accomplished with the following code:

```
// Create an integer array with three elements
int [] fib = new int[3];
fib[0] = 1;
fib[1] = 1;
fib[2] = 2;

// Redimension message to a 10 element array
int [] temp = new int[10];

// Copy the fib array to temp
fib.CopyTo(temp, 0);

// Assign temp to fib
fib = temp;
```

After the last line of code, `fib` references a ten-element `Int32` array. The elements 3 through 9 in the `fib` array will have the default `Int32` value—0.

Arrays are excellent data structures to use when storing a collection of homogeneous types that you only need to access directly. Searching an unsorted array has linear running time. While this is acceptable when working with small arrays, or when performing very few searches, if your application is storing large arrays that are searched frequently, there are a number of other data structures better suited for the job. We'll look at some such data structures in upcoming pieces of this article series. Realize that if you are searching an array on some property and the array is *sorted* by that property, you can use an algorithm called binary search to search the array in $O(\log n)$ running time, which is on par with the search times for binary search trees. In fact, the `Array` class contains a static, `BinarySearch()` method. For more information on this method, check out an earlier article on mine, Efficiently Searching a Sorted Array.

> **Note** The .NET Framework allows for multi-dimensional arrays as well. Multi-dimensional arrays, like single-dimensional arrays, offer a constant running time for accessing elements. Recall that the running time to search through a n-element single dimensional array was denoted O(n). For an nxn two-dimensional array, the running time is denoted $O(n^2)$ because the search must check n^2 elements.

More generally, a k-dimensional array has a search running time of $O(n^k)$. Keep in mind here than n is the number of elements in each dimension, not the total number of elements in the multi-dimensional array.

Creating Type-Safe, Performant, Reusable Data Structures

When creating a data structure for a particular problem, oftentimes the data structure's internals can be customized to the specifics of the problem. For example, imagine that you were working on a payroll application. One of the entities of this system would be an employee, so you might create an `Employee` class with applicable properties and methods. To represent a set of employees, you could use an array of type `Employee`, but perhaps you need some extra functionality not present in the array, or you simply don't want to have to concern yourself with writing code to watch the capacity of the array and resize it when necessary. One option would be to create a custom data structure that uses an internal array of `Employee` instances, and offered methods to extend the base functionality of an array, such as automatic resizing, searching of the array for a particular `Employee` object, and so on.

This data structure would likely prove very helpful in your application, so much so that you might want to reuse it in other applications. However, this data structure is not open to reuse because it is tightly-coupled to the payroll application, only being able to store elements of type `Employee` (or types derived from `Employee`). One option to make a more flexible data structure is to have the data structure maintain an internal array of `object` instances, as opposed to `Employee` instances. Because all types in the .NET Framework are derived from the `object` type, the data structure could store any type. This would make your collection data structure usable in other applications and scenarios.

Not surprisingly, the .NET Framework already contains a data structure that provides this functionality—the `System.Collections.ArrayList` class. The ArrayList maintains an internal `object` array and provides automatic resizing of the array as the number of elements added to the ArrayList grows. Because the ArrayList uses an object array, developers can add any type—strings, integers, `FileInfo` objects, `Form` instances, anything.

While the ArrayList provides added flexibility over the standard array, this flexibility comes at the cost of performance. Because the ArrayList stores an array of `objects`, when reading the value from an ArrayList you need to explicitly cast it to the data type being stored in the specified location. Recall that an array of a value type—such as a `System.Int32`, `System.Double`, `System.Boolean`, and so on—is stored contiguously in the managed heap in its unboxed form. The ArrayList's internal array, however, is an array of `object` references. Therefore, even if you have an ArrayList that stores nothing but value types, each ArrayList element is a reference to a boxed value type, as shown in Figure 3.

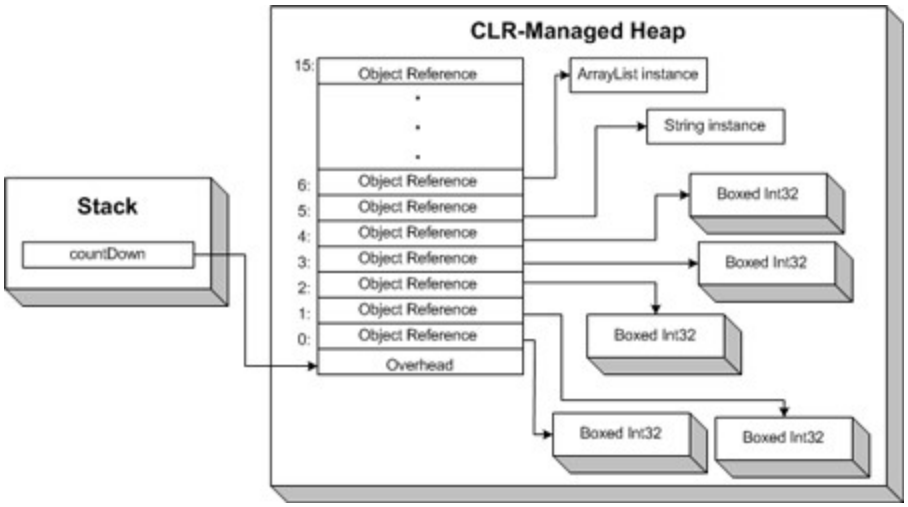

Figure 3. The ArrayList contains a contiguous block of object references

The boxing and unboxing, along with the extra level of indirection, that comes with using value types in an ArrayList can hamper the performance of your application when using large ArrayLists with many reads and writes. As Figure 3 illustrates, the same memory layout occurs for reference types in both ArrayLists and arrays.

Having an `object` array also introduces potential bugs that won't be noticed until run-time. A developer may intend to only add elements of a particular type to an ArrayList, but since the ArrayList allows any type to be added, adding an incorrect type won't be caught during compilation. Instead, such a mistake would not be apparent until run-time, meaning the bug would not be found until testing or, in the worse case, during actual use.

Generics to the Rescue

Fortunately, the typing and performance issues associated with the ArrayList have been remedied in the .NET Framework 2.0, thanks to *Generics*. Generics allow for a developer creating a data structure to defer type selection. The types associated with a data structure can, instead, be chosen by the developer utilizing the data structure. To better understand Generics, let's look at an example of creating a type-safe collection. Specifically, we'll create a class that maintains an internal array of a to-be specified type, with methods to read and add items from the internal array.

```
public class TypeSafeList<T>
{
    T[] innerArray = new T[0];
    int currentSize = 0;
    int capacity = 0;

    public void Add(T item)
    {
        // see if array needs to be resized
        if (currentSize == capacity)
        {
            // resize array
            capacity = capacity == 0 ? 4 : capacity * 2;  // double capacity
            T[] copy = new T[capacity];    // create newly sized array
            Array.Copy(innerArray, copy, currentSize);  // copy over the array
            innerArray = copy;    // assign innerArray to the new, larger array
        }
```

```
        innerArray[currentSize] = item;
        currentSize++;
    }

    public T this[int index]
    {
        get
        {
            if (index < 0 || index >= currentSize)
                throw new IndexOutOfRangeException();
            return innerArray[index];
        }
        set
        {
            if (index < 0 || index >= currentSize)
                throw new IndexOutOfRangeException();
            innerArray[index] = value;
        }
    }

    public override string ToString()
    {
        string output = string.Empty;
        for (int i = 0; i < currentSize - 1; i++)
            output += innerArray[i] + ", ";

        return output + innerArray[currentSize - 1];
    }
}
```

Notice that in the first line of code, in the class definition, a type identifier, `T`, is defined. What this syntax indicates is that the class will require the developer using it to specify a single type. This developer-specified type is aliased as `T`, although any other valid variable name could have been used. The type identifier is used within the class's properties and methods. For example, the inner array is of type `T`, and the `Add()` method accepts an input parameter of type `T`, which is then added to the array.

To declare a variable of this class, a developer would need to specify the type `T`, like so:

```
TypeSafeList<type> variableName;
```

The following code snippet demonstrates creating an instance of TypeSafeList that stores integers, and populating the list with the first 25 Fibonacci numbers.

```
TypeSafeList<int> fib = new TypeSafeList<int>();
fib.Add(1);
fib.Add(1);

for (int i = 2; i < 25; i++)
    fib.Add(fib[i - 2] + fib[i - 1]);
```

```
Console.WriteLine(fib.ToString());
```

The main advantages of Generics include:

- **Type-safety:** a developer using the TypeSafeList class can only add elements that are of the type or are derived from the type specified. For example, trying to add a string to the `fib` TypeSafeList in the example above would result in a compile-time error.
- **Performance:** Generics remove the need to type check at run-time, and eliminate the cost associated with boxing and unboxing.
- **Reusability:** Generics break the tight-coupling between a data structure and the application for which it was created. This provides a higher degree of reuse for data structures.

Many of the data structures we'll be examining throughout this series are data structures that utilize Generics, and when creating data structures—such as the binary tree data structure we'll build in Part 3—we'll be utilizing Generics ourselves.

The List: a Homogeneous, Self-Redimensioning Array

An array, as we saw, is designed to store a specific number of items of the same type in a contiguous fashion. Arrays, while simple to use, can quickly become a nuisance if you find yourself needing to regularly resize the array, or don't know how many elements you'll need when initializing the array. One option to avoid having to manually resize an array is to create a data structure that serves as a wrapper for an array, providing read/write access to the array and automatically resizing the array as needed. We started creating our own such data structure in the previous section—the TypeSafeList, but there's no need to implement this yourself as the .NET Framework provides such a class for you. This class, the `List` class, is found in the `System.Collections.Generics` namespace.

The `List` class contains an internal array and exposes methods and properties that, among other things, allow read and write access to the elements of the internal array. The `List` class, like an array, is a homogeneous data structure, meaning that you can only store items of the same type or from a derived type within a given `List`. The `List` utilizes Generics, a new feature in version 2.0 of the .NET Framework, in order to let the developer specify at development time the type of data a `List` will hold.

Therefore, when creating a `List` instance, you must specify the data type of the `List`'s contents using the Generics syntax:

```
// Create a List of integers
List<int> myFavoriteIntegers = new List<int>();

// Create a list of strings
List<string> friendsNames = new List<string>();
```

Note that the type of data the `List` can store is specified in the declaration and instantiation of the `List`. When creating a new `List`, you don't have to specify a `List` size, although you can specify a default starting size by passing in an integer into the constructor, or through the `List`'s `Capacity` property. To add an item to a `List`, simply use the `Add()` method. The `List`, like the array, can have its elements directly accessed via an ordinal index. The following code snippet shows creating a `List` of integers, populating the list with some initial values with the `Add()` method, and then reading and writing the `List`'s values through an ordinal index.

```
// Create a List of integers
List<int> powersOf2 = new List<int>();

// Add 6 integers to the List
powersOf2.Add(1);
powersOf2.Add(2);
powersOf2.Add(4);
powersOf2.Add(8);
powersOf2.Add(16);
powersOf2.Add(32);

// Change the 2nd List item to 10
powersOf2[1] = 10;

// Compute 2^3 + 2^4
int sum = powersOf2[2] + powersOf2[3];
```

The `List` takes the basic array and wraps it in a class that hides the implementation complexity. When creating a `List`, you don't need to explicitly specify an initial starting size. When adding items to the `List`, you don't need to concern yourself with resizing the data structure, as you do with an array. Furthermore, the `List` has a number of other methods that take care of common array tasks. For example, to find an element in an array, you'd need to write a `for` loop to scan through the array (unless the array was sorted). With a `List`, you can simply use the `Contains()` method to determine if an element exists in an array, or `IndexOf()` to find the ordinal position of an element. The List class also contains a `BinarySearch()` method to efficiently search a sorted array, and methods like `Find()`, `FindAll()`, `Sort()`, and `ConvertAll()`, which can utilize delegates to perform operations that would require several lines of code using arrays.

The asymptotic running time of the `List`'s operations are the same as those of the standard array's. While the `List` does indeed have more overhead, the relationship between the number of elements in the `List` and the cost per operation is the same as the standard array.

Conclusion

This article, the first in a series of six, started our discussion on data structures by identifying why studying data structures was important, and by providing a means of how to analyze the performance of data structures. This material is important to understand, as being able to analyze the running times of various data structure operations is a major tool used when deciding what data structure to use for a particular programming problem.

After studying how to analyze data structures, we turned to examining two of the most common data structures in the .NET Framework Base Class Library: `System.Array` and `System.Collections.Generics.List`. Arrays allow for a contiguous block of homogeneous types and derived types. Their main benefit is that they provide lightning-fast access to reading and writing array elements. Their weak point lies in searching arrays, as each and every element must potentially be visited (in an unsorted array), and the fact that resizing the array requires writing a bit of code.

The `List` class wraps the functionality of an array with a number of helpful methods. For example, the `Add()` method adds an element to the List and automatically re-dimensions the array if needed. The `IndexOf()` method aids the developer by searching the `List`'s contents for a particular item. The functionality provided by a List is nothing that you couldn't implement using plain old arrays, but the List class saves you the trouble of having to write the code to perform these common tasks yourself.

In the next part of this article series we'll turn our attention first to two "cousins" of the **List**: the **Stack** and **Queue** classes. We'll also look at associative arrays, which are arrays indexed by a string key as opposed to an integer value. Associative arrays are provided in the .NET Framework Base Class Library using the **Hashtable** and **Dictionary** classes.

Happy Programming!

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since January 1998. Scott works as an independent consultant, trainer, and writer, and holds a Masters degree in Computer Science from the University of California – San Diego. He can be reached at mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.