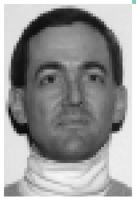
GLEN MCCLUSKEY

making use of c# collections



Glen McCluskey is a consultant with 20 years of experience who has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

glenm@glenmccl.com

THE TERM "COLLECTION" IN MODERN programming languages typically refers to a group of elements, such as integers or strings, organized in a list or table of some sort. Languages such as C++ and C# provide library facilities for managing collections. In this column we'll look at C# collections and illustrate some of the techniques you can use in your applications.

Managing Lists of Integers

To help tie down the idea of what a collection is, consider a two-part example. Suppose that you're doing some C# programming, and you need to keep a list of integer values around. Your first solution to this problem looks like so:

```
using System;
public class ListDemo1 {
  static int[] list = new int[10];
  static int currlen = 0;

  static void add(int elem) {
    if (currlen == list.Length) {
      int[] newlist = new
      int[(int)(currlen * 1.5)];
      for (int i = 0; i < currlen;
      i++)
        newlist[i] = list[i];
      list = newlist;
    }

    list[currlen++] = elem;
  }

  public static void Main(string[]
  args) {
    for (int i = 1; i <= 20; i++)
      add(i);

    Console.WriteLine("length = " +
    currlen);

    Console.Write("elements = ");
    for (int i = 0; i < currlen; i++)
      Console.Write(list[i] + " ");
    Console.WriteLine();
  }
}
```

The idiom used in this example is a common one. An array is allocated, and values are inserted into it. When the array overflows, a new array is allocated, and the contents of the old array are copied to it.

This code works, is efficient, and is easy to follow. Unfortunately, however, you may end up re-solving this problem over and over again in your applications, possibly introducing errors along the way. For example, suppose that if instead of:

```
if (currlen == list.Length)
  ...
```

I'd said:

```
if (currlen > list.Length)
  ...
```

The code in this case might work most of the time, but occasionally it would trigger an exception caused by going off the end of the array.

An alternative is to use the C# collection class `ArrayList`, like this:

```
using System;
using System.Collections;

public class ListDemo2 {
  public static void Main(string[] args) {
    ArrayList list = new ArrayList();

    for (int i = 1; i <= 20; i++)
      list.Add(i);

    Console.WriteLine("length = " + list.Count);

    Console.Write("elements = ");
    foreach (int i in list)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

All the details are handled automatically for you when using `ArrayList`. The list is grown automatically, there is a standardized interface, and so on. Most of the time, such convenience is exactly what you want, but, of course, the default handling may be different from what you as a programmer might expect. For example, in the first demo above, we created the integer list with room for 10 elements, and increased its size by 50% when needed. In contrast, `ArrayList` starts with a capacity of 16, and doubles the size when the list is internally reallocated.

An object of type `ArrayList` contains an internal array, which holds the actual elements. Strictly speaking, our example does not create an `ArrayList` of integers but, rather, an `ArrayList` of references to wrapper objects, each of which contains an integer. In C#, elements of primitive type are automatically converted ("boxed") to object type as needed, and then can be converted back ("unboxed") using a cast.

Note also the use of the "foreach" statement, a convenient shorthand for iterating across the elements of a collection.

Sorting

Suppose that we want to go further with our example and sort the list of integers into descending order. To do so, we make use of a comparer class that implements the standard interface `IComparer`. The code looks like this:

```
using System;
using System.Collections;
```

```
public class MyComparer : IComparer {
  public int Compare(object aobj, object bobj) {
    int a = (int)aobj;
    int b = (int)bobj;
    return a < b ? 1 : (a == b ? 0 : -1);
  }
}

public class SortDemo {
  public static void Main(string[] args) {
    ArrayList list = new ArrayList();

    for (int i = 1; i <= 20; i++)
      list.Add(i);

    list.Sort(new MyComparer());

    Console.WriteLine("length = " + list.Count);

    Console.Write("elements = ");
    foreach (int i in list)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

The `ArrayList` sort method must have some standardized way of calling back to a comparison method. If a class implements a specific interface, such as `IComparer`, then that class is guaranteed to have a method `Compare(object,object)` defined in it, and the sort routine can call this method through the passed-in `MyComparer` object.

Making Copies of Collection Objects

We said above that an `ArrayList` of integers is really an array of references to integer wrapper objects. Such internal details matter in a case where we make a copy of a collection. For example, suppose that we have an `ArrayList`, and we add an object to it, and then copy the list, and then modify the object previously added.

We can observe what happens by running this code:

```
using System;
using System.Collections;

public class MyClass {
  private int val;

  public MyClass(int i) {
    setVal(i);
  }

  public void setVal(int i) {
    val = i;
  }

  public override string ToString() {
    return val.ToString();
  }
}

public class CloneDemo {
  public static void Main(string[] args) {
    ArrayList list1 = new ArrayList();
```

```
      MyClass obj = new MyClass(10);
      list1.Add(obj);
      Console.WriteLine(list1[0]);

      ArrayList list2 = (ArrayList)list1.Clone();
      obj.setVal(20);
      Console.WriteLine(list2[0]);
    }
  }
```

When the list is cloned, a shallow copy is made, that is, a new internal array is allocated, and the object references are copied to the new array. If an object is modified after creation, then modifying the original object will modify the copy as well, because there are multiple references to the same object.

The output of the program is:

```
10
20
```

and we observe that modifying the object does indeed have effect in the list copy.

BitArrays and Iteration

Let's go on and look at another collection class, used to manage lists of bits. In this example we create two `BitArrays`, set some of the bits, and then perform an `XOR` operation:

```
using System;
using System.Collections;

public class BitDemo {
  public static void Main(string[] args) {
    BitArray ba1 = new BitArray(8);
    ba1.Set(0, true);
    ba1.Set(2, true);

    BitArray ba2 = new BitArray(8);
    ba2.Set(1, true);
    ba2.Set(3, true);

    ba1.Xor(ba2);

    for (int i = 0; i < ba1.Count; i++) {
      if (ba1.Get(i))
        Console.Write(i + " ");
    }
    Console.WriteLine();

    IEnumerator e = ba1.GetEnumerator();
    while (e.MoveNext())
      Console.Write(e.Current + " ");
    Console.WriteLine();
  }
}
```

There are a couple of ways to display the result of the XOR. The first looks at all the bits, and prints the integer index for each one that is set.

The second approach uses an enumerator, a standardized way of iterating across all the values in the collection. When we run this program, the result is:

```
0 1 2 3
True True True True False False False False
```

The enumerator code is generic, and can be used to iterate across other types of collections.

Hashtables

A final example of C# collections is the use of hashtables. This program illustrates how a phone directory might be built up and then accessed:

```
using System;
using System.Collections;

public class HashDemo {
  public static void Main(string[] args) {
    Hashtable ht = new Hashtable();
    //ht = Hashtable.Synchronized(ht);

    ht.Add("Bill Smith", "123-4567");
    ht.Add("Mary Jones", "234-5678");
    ht.Add("John Davis", "345-6789");

    ICollection keys = ht.Keys;

    foreach (string key in keys)
      Console.WriteLine(key + "   " + ht[key]);
  }
}
```

The `Keys` property is used to obtain a list of all the keys in a hashtable. This list is enumerated and each key looked up in the table using the `[]` operator (“indexer”) for the hashtable.

Note that the demo has a line:

```
//ht = Hashtable.Synchronized(ht);
```

If we uncomment this line, the effect is to put a wrapper around the hashtable reference, such that access to the hashtable is synchronized, that is, made thread-safe. By default, collections are not thread-safe, and if you are concerned about such an issue, you need to take the appropriate steps when writing C# code.

C# collections are an easy to use and powerful tool to solve many common programming problems. They are part of the standard C# library, and you can use them in place of writing your own code.