Lecture 12 — More About Arrays

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Part I

The `foreach` Loop

Iterating Over an Array

Last time, we iterated over entries of an array using the `for` loop.

C# provides another construct to iterate over all the entries in an array: the `foreach` statement.

The `foreach` statement is just some slightly more convenient syntax; anything we do with it we could do with a regular `for` loop.

Let's assume we have an integer array (`int[]` grades) defined.

Use of the `foreach` Statement

The syntax is: foreach (*type identifier* in *arrayIdentifier*)

```
int[] grades;

// Initialization of grades not shown

foreach ( int num in grades )
{
    // Loop Body
}
```

The *type* in the expression must be the same as the type of the array.

Let's look at an example with the loop body filled in.

Using the foreach Statement

```
string[] months = { "January", "February", "March",
        "April",  "May", "June", "July", "August",
        "September", "October", "November", "December"
         };

foreach ( string s in months )
{
    Console.WriteLine( s );
}
```

Use of the foreach Statement

Any foreach loop could also be replaced with a for loop.

Here's a foreach loop to print all the grades in the array grades.

```
foreach ( int num in grades )
{
    Console.WriteLine( num );
}
```

Now, rewrite this using for:

```
for ( int index = 0; index < grades.Length; index++ )
{
    Console.WriteLine( grades[index] );
}
```

Using the foreach Statement

Rewriting the months example:

```
string[] months = {"January", "February", "March", "April",
        "May", "June", "July", "August", "September",
        "October", "November", "December" };

for ( int i = 0; i < months.Length; i++ )
{
    Console.WriteLine( months[i] );
}
```

Comparing `for` and `foreach`

The `for` and `foreach` loops are often functionally equivalent.

The `break` and `continue` work as expected in `foreach`.

The `foreach` syntax is focused on iterating over all the entries of an array; the `for` loop is more flexible.

Recall that the `for` loop can have any stopping condition (and therefore behave like the `while` loop).

The `for` loop can also go backwards, skip every second item, etc.

Part II

The String Revisited

The `string` Revisited

We have already learned about a string type, but we haven't examined it in much detail.

The string was just a bunch of text, but there is much more to it than we might think at first glance...

```
string svar1; // Creates uninitialized string
string svar2 = "Literal"; // Initialized.
string svar3 = ""; // Initialized to the empty string.
```

The `string` is a complex type (there's a reason it wasn't introduced in the simple types alongside `int` and `double`).

String Concatenation

Strings can appear in expressions using the + operator. It does not add up the values, but instead performs concatenation.

```
string verb = "fore" + "see";
```

This means the variable `verb` contains `"foresee"`.

The use of += can also be used for concatenation:
`verb += "n";` → `verb` is now `foreseen`.

Remember that for concatenation, like an arithmetic expression, it is necessary to assign the value somewhere.

Strings are Complex

It turns out that the string has a member variable `Length` that tells you the number of characters in the `string`.

This information, combined with the fact that a `string` is a bunch of text characters should lead you to the conclusion that...

The `string` is really an array of `chars`.

An Array of `char`

This means we can access individual characters within a `string` using their index values (just as if they were entries of an array).

If the string is `string ex1 = "example"`, the `char` at `ex1[3]` is `'m'`.

We could use a `for` (or `foreach`) loop to iterate over all the characters of the `string` if we are looking for something specific.

Iterating Over the `string`

```
string s = "Hello World!";

for ( int i = 0; i < s.Length; i++ )
{
    Console.WriteLine( s[i] );
}

for ( int j = s.Length -1; j >= 0; j-- )
{
    Console.Write( s[j] );
}
```

Mutating `string` Elements

Characters within a `string` cannot be changed using index values.

This is different from an array of `int` where we could assign `array[7] = -98;`

The string type is immutable. `string` variables cannot be changed.

Every time a string must be "changed", a new `string` must be created.

Immutable `string` Variables

If we have a statement `string text = name + suffix;` after this statement, there are 3 strings in memory: `text`, `name`, `suffix`.

That's not surprising, but consider this: `name += suffix;`

There are still three strings stored in memory after this statement. `suffix`, the new value of `name`, and the old value of `name`.

The += operation did two things:

1. Created a new string and (the concatenation of `name` and `suffix`) and stored it in memory.
2. Changed the memory location `name` is associated with to the location of the new string.

Part III

Multi-Dimensional Arrays

Arrays of Arrays

You may have wondered if we can have an array of any type, can we have an array of arrays? Yes!

A multi-dimensional array is an array of array types.

The following statement declares and instantiates a multi-dimensional array of `int` named `day`: `int[][] day = new int [n][];`

Is this syntax confusing? Perhaps imagine it like this: `(int[])[]`

The type is `int[]` (in brackets) and when we declare an array of a type, write `[]` after the type.

Hence, we declare an array of type `int[]` (integer array).

Arrays of Arrays

`day[0]` is a reference to the first integer array
`day[1]` is a reference to the second integer array
`day[n-1]` is a reference to the nth integer array

The length of `day[0]` and `day[1]` may be different.

Setting up A Multi-Dimensional Array

```
int[] daysInMonth = { 31, 28, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31 };

int[][] year = new int[12][];

for ( int month = 0; month < 12; ++month )
{
    year[month] = new int[ daysInMonth[month] ];

    for ( int day = 0; d < daysInMonth[month]; ++d )
    {
        year[month][day] = 1;
    }

}
```

Setting up A Multi-Dimensional Array

Now let's print out this calendar.

```
for ( int month = 0; month < 12; ++month )
{
    for ( int day = 0; d < daysInMonth[month]; ++d )
    {
        Console.Write( year[month][day] );
        Console.Write( " " );
    }
    Console.WriteLine( "" );
}
```

Further Initialization of Multi-Dimensional Arrays

It is possible to initialize a multi-dimensional array when it is declared.

For example, the following code defines a multi-dimensional array of characters named `myChars`:

```
char[][] myChars = {
          new char[] { 'B', 'i', 'l', 'l' },
          new char[] { 'D', 'a', 'v', 'e' },
          new char[] { 'G', 'e', 'o', 'r', 'g', 'e' }
          };
```

Arrays of Arrays of Arrays...

The multi-dimensional array examples we have shown so far are all "two dimensional".

We could have more, such as `int[][][] coordinates;` to describe x, y, and z co-ordinates.

In C# the term for arrays of these kinds is "jagged arrays".

C# also supports rectangular arrays, which we'll come back to later.