Lecture 11 — Arrays

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Array Motivation

Up to now, when we declare a variable, we get one of the built-in data types, an `enum` or `struct`.

Very often we are interested in a collection of some items.

We've seen an enumeration already, which is a list of fixed elements/options, like the days of the week.

What if we wanted to have a list of 11 integers, the values of which will change from time to time?

About Arrays

Like the `struct`, we could live without an array by declaring each variable as many times as it takes:

```
int assignment1;
int assignment2;
int assignment3;
...
```

This is... highly suboptimal.

Instead, we could define a list of items using a single variable name.

The concept for this is an array.

Array Definitions

An array is an indexed list of items of the same type.

Arrays can be declared for any data type. Even one we have defined.

Think of an array as a list of variables that can easily be manipulated.

Array Declaration

When declaring `int x;` this creates a single integer variable.

Adding [ ] after the type signals intent to create an integer array:

```
int[] assignments;
```

The variable `assignments` will be a list of `int` types.

Array Declaration

A simple array declaration and initialization uses the format:

type[ ] identifier = new type[capacity];

For example, an array of 11 integers can be declared as follows:

```
int[] assignments = new int[11];
```

The square brackets [ ] are used for many array operations.

Array Allocation

When declaring a regular `int` we saw we don't have to initialize it at the beginning, and later we can assign it a value directly.

When declaring an integer array, before we can use the array, we need to initialize it in a special way.

The keyword `new` is used to allocate memory for something (in this case, an array).

Initializing the Array

Allocation is the process where the computer designates an area of memory to as the location of variable.

Recall the analogy that when we wrote `int x;` we draw a box (of integer size) and labelled it `x`.

When we initialize the array, we need to specify a capacity: the maximum number of entries in the list.

Extend the analogy: the capacity tells us how many boxes (of integer size) to draw under the heading of `assignments`.

Without the capacity, we don't know how many boxes to draw.

Initializing the Array

For `assignments` we drew 11 boxes. Each of these boxes is labelled with an index (to tell them apart).

Index always starts at 0; increments by 1. The last is (`capacity` - 1).

The index is an "offset" from the start of the list.

For `assignments`, because capacity is 11, indexing goes from 0 to 10.

This is a very common source of errors in programming: forgetting that 0 is the first entry and (capacity-1) is the last.

At this point we only created the array in memory; we have not yet initialized each of the values of that array.

The [ ] Operator

To access any entry of `assignments`, such as the 2^{nd}, use the [ ] operator (sometimes called the indexing operator).

To access an entry of `assignments`, put the index of the item in the [ ] brackets: `int secondValue = assignments[1];`

Note that the second entry has an index of 1, because the index goes from 0 to 9.

This statement is an error:

```
int lastValue = assignments[11];
```

Because 11 is outside the bounds of the array.

The [ ] Operator

C# has automatic bounds checking for arrays.

An attempt to access `assignments[11]` is checked.

Only if the value is in range will the access be permitted.

Otherwise, we will get a run-time error immediately.

Other languages like C++ do not have array bounds checking.

If we attempt to read at an index after the end of the array:

1. If we're "lucky" we may read some bogus value.
2. Or it might result in a program crash (now or later).

Automatic bounds checking seems like a good precaution. The trade-off is that it has a negative impact on performance.

The [ ] Operator

You may use a variable as the index in the indexing operator.
To print all values of an array, use a `for` loop:

```
int capacity = 3;
int[] array = new int[capacity];
array[0] = 7;
array[1] = 99;
array[2] = 856;

for ( int index = 0; index < capacity; index++ )
{
    Console.Write( "Array at index ")
    Console.Write( index )
    Console.Write( " = " )
    Console.WriteLine( array[index] );
}
```

The Assignment Example, Continued

Suppose you are an instructor, and you've collected 11 weekly quizzes from students.

You have a number of different marking schemes where a grade will be decided on the maximum of:

1. The average of the eleven grades,
2. The average of the last nine grades, and
3. The average of the best ten grades with a penalty of 2%.

As a student, how would you determine your grade?

Calculating Assignment Grades

First approach: do the calculation with pen and paper.

But this is a boring activity and the kind of thing a computer can do.

... Especially if you must do this for every student in a class of 360.

The Assignment Example, Continued

Make the computer do it:

```
int[] assignments = new int[11];
// Filling in the array with Console.ReadLine() not shown

double sum = 0.0;

for ( int k = 0; k < 11; ++k )
{
    sum += assignments[k];
}

double grade_1 = sum/11;
```

This calculated the average of the 11 grades.

The Assignment Example, Continued

```
double sum = 0.0;

for ( int k = 2; k < 11; ++k ) {
    sum += assignments[k];
}

double grade_2 = sum/9;
```

This is the second formula.

The Assignment Example, Continued

```
double minimum_grade = assignments[0];

for ( int k = 1; k < 11; ++k ) {
    if ( assignments[k] < minimum_grade ) {
        minimum_grade = assignments[k];
    }
}

double grade_3 = (sum - minimum_grade)/10 - 0.2;
```

Note that a 2% penalty on a grade out of 10 is 0.2.

The Assignment Example

Now we have worked out the three different grades:
`grade_1`, `grade_2`, `grade_3`.
Let's print the maximum one to the screen.

```
double maxGrade = grade_1;

if ( grade_2 > maxGrade )
{
    maxGrade = grade_2;
}
if ( grade_3 > maxGrade )
{
    maxGrade = grade_3;
}

Console.Write( "Assignment Grade: " );
Console.WriteLine( maxGrade );
```

Dynamic Array Allocation

The array capacity does not need to be specified at compile time.

You may declare and initialize an array of capacity *n* as follows:

```
int[] nSizeArray = new int[n];
```

This applies even when variable `n` was given by user input.

However, `n` must be an integer. You can't have 2.5 items in an array.

Similarly, you can use an expression to set the capacity:

```
double[] n10SizeArray = new double[n + 10];
```

More Advanced Example

```
int[] grades = new int[5];
for ( int i = 0; i < 5; i++ )
{
    do
    {
        Console.Write( "Enter Grade: ");
        grades[i] = int.Parse( Console.ReadLine( ) );
    }
    while( (grades[i] < 0) || (grades[i] > 100) );
}

for ( int i = 0; i < 5; i++ )
{
    Console.Write( i )
    Console.Write( ": " )
    Console.WriteLine( grades[i] );
}
```

Alternative Array Initialization

There is another syntax for initializing an array and it looks just like how we defined an `enum`.

```
int[] fixedArray = { 1, 2, 3, 4, 1 };
```

This allocates an array of capacity 5, and initializes each of the values of the array (according to the values in the braces).

The compiler figures out the capacity based on the number of entries in braces, allocates and initializes the array.

Unlike an `enum` or mathematical set, duplicates are allowed.

The Array `Length`

Remember from the structure that we accessed `today.year` using the dot operator.

It turns out that in C# the array is in its own way, a kind of structure.

Every array therefore has the member variable `Length` that tells you the length of the array (without looking at the allocation).

If the array was allocated as `float[10]` the `Length` will be 10.

But remember the last index of the array is 9 (`Length` -1);
Trying to access `temperature[temperature.Length]` is an error.

More Advanced Example Using Length

```
int[] grades = new int[5];
for ( int i = 0; i < grades.Length; i++ )
{
    do
    {
        Console.Write( "Enter Grade: ");
        grades[i] = int.Parse( Console.ReadLine( ) );
    }
    while( (grades[i] < 0) || (grades[i] > 100) );
}

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

Array of Enumerated Types

We can have an array of an enumerated type, just as a simple type.

```
enum Months = { January, February, March, April, May, June,
July, August, September, October, November, December };
```

```
Months[] courseMonths = new Months[4];
```

To assign an entry: `courseMonths[0] = Months.September;`

Array of Structures

It's also allowed to have an array of structures.

```
struct Date
{
    public int day;
    public int month;
    public int year;
};
```

```
Date[] lectureDates = new Date[36];
```

Note the syntax: `lectureDates[0].year = 2014;`

Summing Up Basic Arrays

```
int[] grades = new int[100];
```

Observations about this code:

1. The array is named `grades`
2. The type of the array is `int[]`
3. The type of an entry is `int`
4. Each entry value can be any value of type `int`
5. The range of indices is 0 to 99

```
grades[50] = 36;
```

6. An index of the array is 50
7. An entry of the array is `grades[50]`
8. A value of an element is 36
9. The value of the element at index 50 is 36.