Lecture 28 — Advanced Arrays

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Arrays in Memory

Now that we are familiar with references, objects, and some basics about memory organization, we can return to the subject of arrays.

It is important to remember the difference between a value type and a reference type when working with arrays:

- An array of value types provides storage for values;
- An array of reference types provides storage for references.

This has an important implication: A declaration of an array of reference types does not provide storage for member fields.

Reference Type Array Example

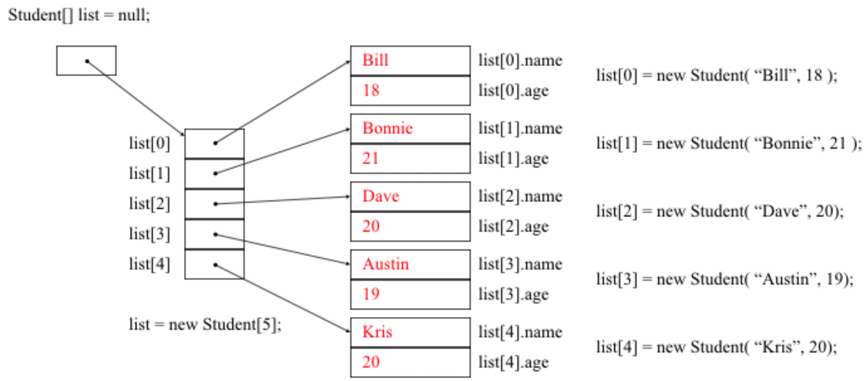
Consider an array of Student entries where each `Student` object has a name member field and an age member field:

```
Student[] list; // Declare an array of references to Student objects

list = new Student[5]; // Instantiates an array
                       // of 5 references to Student objects

// At this point, list[0], list[1], list[2],
// list[3], and list[4] are null references

list[0] = new Student( "Bill", 18 );
list[1] = new Student( "Bonnie", 21 );
list[2] = new Student( "Dave", 20 );
list[3] = new Student( "Austin", 19 );
list[4] = new Student( "Kris", 20 );
```

Notice that creating the array was not enough.
    We also had to create the `Student` objects using `new`.

Reference Type Array Example

And here’s what that code will produce in memory:



Note that there are six references defined in total.

Rectangular Arrays

Remember earlier in the term our examination of arrays included multidimensional arrays.

```
int[][] jag;
```

Declares a jagged array.

C# is unusual; it has rectangular arrays and jagged arrays.

Many languages (such as C, C++, and Java) only have jagged arrays.

The syntax for the rectangular array is:

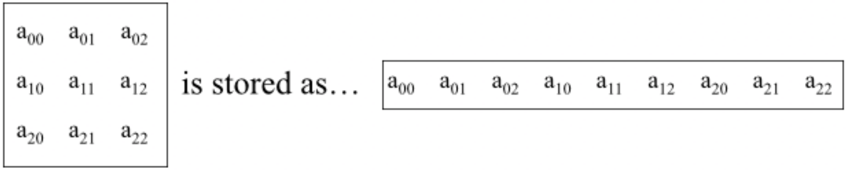
```
int[,] matrix = new int[4,3];
```

This defines a rectangular array with 3 columns and 4 rows.

Rectangular Array Storage

Internally, C# stores arrays using row-major order.

A 3×3 array of integers like `new int[3,3]` is stored in contiguous memory locations like this:


a_{00} a_{01} a_{02}
a_{10} a_{11} a_{12}
a_{20} a_{21} a_{22}
is stored as…
a_{00} a_{01} a_{02} a_{10} a_{11} a_{12} a_{20} a_{21} a_{22}

It may not be obvious, but this is a major advantage to using a rectangular array over a jagged array.

Jagged Array Storage

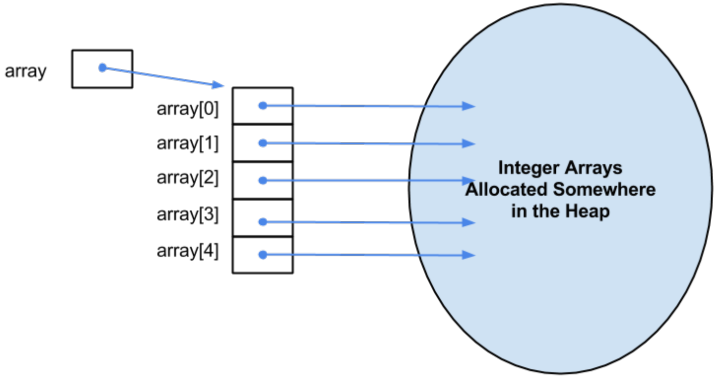
Let’s contrast that against a jagged array declared as follows:

```
int n = 5;
int m = 7
int[][] array = new int[n][];

for( int i = 0; i < n; ++i )
{
    array[i] = new int[m];
}
```

What does this look like in memory?

Jagged Array in Memory

Each of the arrays is allocated somewhere on the heap, but it could be anywhere in that relatively large area.



(The first level of the array is also allocated somewhere on the heap).

Rectangular and Jagged Arrays

Rectangular arrays have nicer syntax and are compact in memory.

Being compact in memory is advantageous because you can do pointer arithmetic to move around within the array in unsafe code.

It is also advantageous because of how CPUs work (which you'll examine in a future course).

Jagged arrays allow more flexibility since each of the arrays need not be the same size.

For a sufficiently large rectangular array, the system may struggle to allocate a single contiguous block of memory.

Rectangular Jagged Arrays

It is possible to create a rectangular array of jagged arrays.
For example: `int[,][] a = new int [n,m][];`

This declares a $n \times m$ two-dimensional array of integer arrays.

It is also possible to create a jagged array of rectangular arrays.
For example: `int[][,] = new int [n][,];`

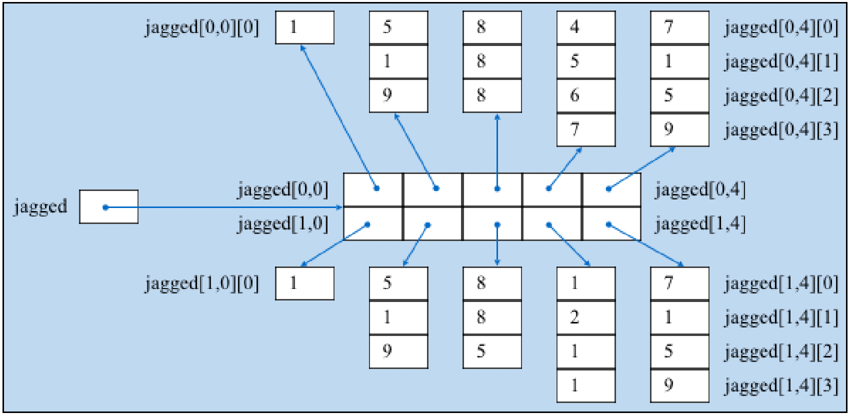
This statement declares a one dimensional array of capacity n of two-dimensional arrays of integers.

Conceptual View of Array Storage

Imagine a rectangular array of jagged arrays: `int[,][] jagged`.

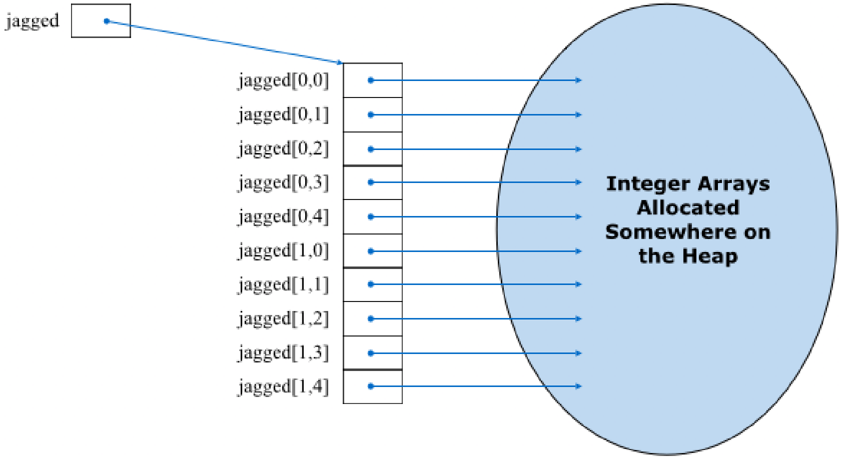
```
int[,][] jagged = new int[2, 5][];
jagged[0, 0] = new int[1];
jagged[0, 1] = new int[3];
jagged[0, 2] = new int[3];
jagged[0, 3] = new int[4];
jagged[0, 4] = new int[4];
jagged[1, 0] = new int[1];
jagged[1, 1] = new int[3];
jagged[1, 2] = new int[3];
jagged[1, 3] = new int[4];
jagged[1, 4] = new int[4];
```

What does `jagged` look like when allocated in memory?
Let's assume we have assigned values to these integers.



Memory View of Array Storage

The storage allocated in this example will look, however, like this:



(Once again, the first level of the array is allocated on the heap).

Memory View of Array Storage

A few notes about the previous diagram:

1. The two-dimensional array is actually stored as a one-dimensional array.
2. The rectangular jagged array is not contiguous but each of the individual arrays are contiguous.
3. If starting address of the array of references is x, the ending address is $x + 9s$ where s is the size of a reference.

Comments on Array Storage

Rectangular arrays:

- Implement 1 contiguous storage block that contains all entries of an array.
- May have entries that are value or reference type
- Allow efficient computation of the starting location of any entry using the indices of the array.

Jagged arrays:

- Implement several contiguous storage blocks that contain all entries of an array.
- Define one or more arrays of references that refer to storage locations of the actual entries of an array
- Require more time to access entries due to the additional level(s) of indirection

Understanding Properties of Arrays

Arrays have the following properties of interest:

Property	**Description**
Rank	The number of dimensions of an array
Length	The size of a particular dimension
Lower Bound	The first index of a particular dimension
Upper Bound	The last index of a particular dimension

C# provides the following properties and instance methods:

Method	**Returns**
`Rank`	Number of dimensions
`GetLength( int i )`	Length of dimension *i*
`GetLowerBound( int i )`	Lower bound on dimension *i*
`GetUpperBound( int i )`	Upper bound on dimension *i*

Examining the Properties of Arrays

Suppose we define a jagged integer array as follows:

```
int[][] jagged = {  new int[] {1},
                    new int[] {5,1,9},
                    new int[] {8,8,8},
                    new int[] {4,5,6,7},
                    new int[] {7,1,5,9}
                 };
```

Now let us examine the properties of this jagged array.

Examining the Properties of Arrays

```
Console.WriteLine( "jagged Rank = "
                           + jagged.Rank );
Console.WriteLine( "jagged Length = "
                   + jagged.GetLength(0) );
Console.WriteLine( "jagged Lower Bound = "
                   + jagged.GetLowerBound(0) );
Console.WriteLine( "jagged Upper Bound = "
                   + jagged.GetUpperBound(0) );
Console.Write( "\n" );
```

[In class demo: output of this code fragment]

Examining the Properties of Arrays

Now let's examine the arrays referenced by `jagged`:

```
for( int row = 0; row <  jagged.GetLength( 0 ); row++ )
{
    Console.WriteLine( "jagged[" + row + "] Rank = "
                                 + jagged[row].Rank );
    Console.WriteLine( "jagged[" + row + "] Length = }",
                                 + jagged[row].GetLength(0) );
    Console.WriteLine( "jagged[" + row + "] Lower Bound = ",
                                 + jagged[row].GetLowerBound(0) );
    Console.WriteLine( "jagged[" + row + "] Upper Bound = "
                                 + jagged[row].GetUpperBound(0) );
    Console.Write( "\n" );
}
```

[In class demo: output of this code fragment]

Examining Rectangular Jagged Arrays

Suppose we redefine `jagged` as a rectangular array of jagged arrays.

```
int[,][] jagged =
{
    {
        new int[] {1},
        new int[] {5,1,9},
        new int[] {8,8,8},
        new int[] {4,5,6,7},
        new int[] {7,1,5,9}
    },

    {
        new int[] {1},
        new int[] {5,1,9},
        new int[] {8,8,5},
        new int[] {1,2,1,1},
        new int[] {7,1,5,9}
    }
};
```

The initialization is a non-trivial task. Whenever possible, use whitespace characters to make your task easier.

Examining Rectangular Jagged Arrays

Now, let’s examine the output of the output property programs fragments on the rectangular jagged array.

[In-class demo: output of the two array analysis code fragments]

Enlarging Arrays

An array is of fixed capacity (even if the capacity is user input).

Plan ahead: Allocate an array of size 999 when we aren’t sure how many we’ll need, and hope that’s enough?

What do we do if the array is “full” but we’d like to add more entries?

Reactively: Create a new, bigger array if you need it and copy all the data to the bigger one...?

Enlarging Arrays

The general procedure to enlarge an array is:

1. Request memory for a new array
2. Copy the values over
3. Reassign the original reference

In C#, the original array will go out of scope and therefore become subject to garbage collection.

(Shrinking the array follows the same sequence, but it doesn't happen nearly as often as enlarging.)

Enlarging Array Code

First idea – increase the capacity of the array by one.

```
public static void increaseCapacity( ref int[] array )
{
    int newCapacity = array.Length + 1;
    int[] largerArray = new int[newCapacity];

    for( int i = 0; i < array.Length; ++i )
    {
        largerArray[i] = array[i];
    }

    array = largerArray;
}
```

But this is really inefficient if we have to do it many times.

Enlarging Array Code

A second idea: enlarge the array to twice the original capacity.

```
public static void doubleCapacity( ref int[] array )
{
    int newCapacity = array.Length * 2;
    int[] largerArray = new int[newCapacity];

    for( int i = 0; i < array.Length; ++i )
    {
        largerArray[i] = array[i];
    }

    array = largerArray;
}
```

Collections

Enlarging the array can take a while if the array is large.

Enlarging the array to increase the capacity by one will probably be very inefficient as we'd likely end up enlarging the array many times.

Yet, doubling the capacity when enlarging may result in wasting a lot of memory, such as an array of capacity 200 000 that's only half full...

It would be nice to have a collection of arbitrary capacity...
And can accommodate however many objects we want to add.