Inft2012 Application Programming

Lecture 7

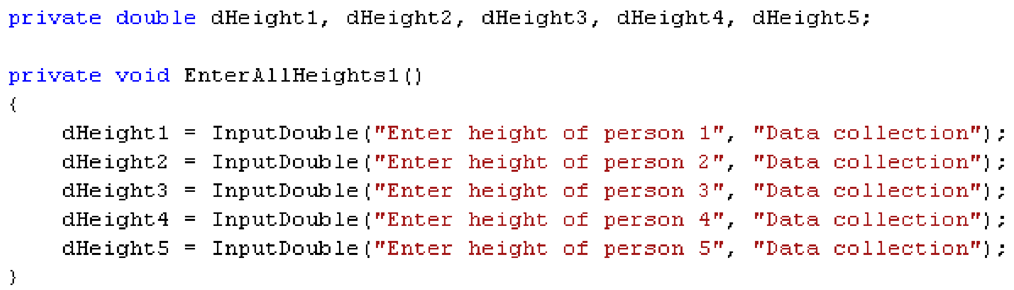
Arrays and related structures

Arrays

- You must remember the array, which stores many values of the same type in a single variable with just one name
- For example, an array called dHeight might have 5 values called dHeight[0], dHeight[1], dHeight[2], dHeight[3], and dHeight[4]
- They're all called dHeight, but each value has a different *index* – the number after it in brackets
- We could just as easily have an array with 500 or 5000 heights
- So what are the benefits of an array as against a number of distinct variables?

One array vs 5 variables

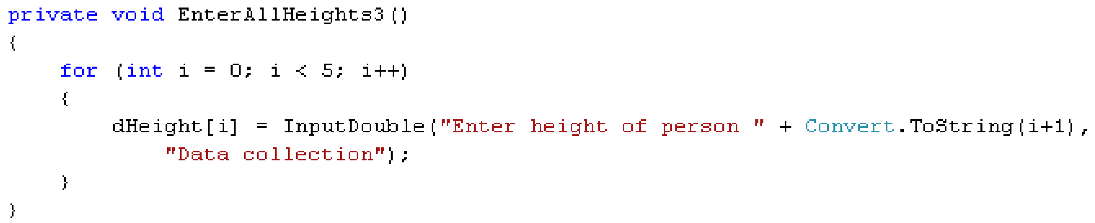
- Compare this . . .



```
private double dHeight1, dHeight2, dHeight3, dHeight4, dHeight5;

private void EnterAllHeights1()
{
    dHeight1 = InputDouble("Enter height of person 1", "Data collection");
    dHeight2 = InputDouble("Enter height of person 2", "Data collection");
    dHeight3 = InputDouble("Enter height of person 3", "Data collection");
    dHeight4 = InputDouble("Enter height of person 4", "Data collection");
    dHeight5 = InputDouble("Enter height of person 5", "Data collection");
}
```

- with this . . .



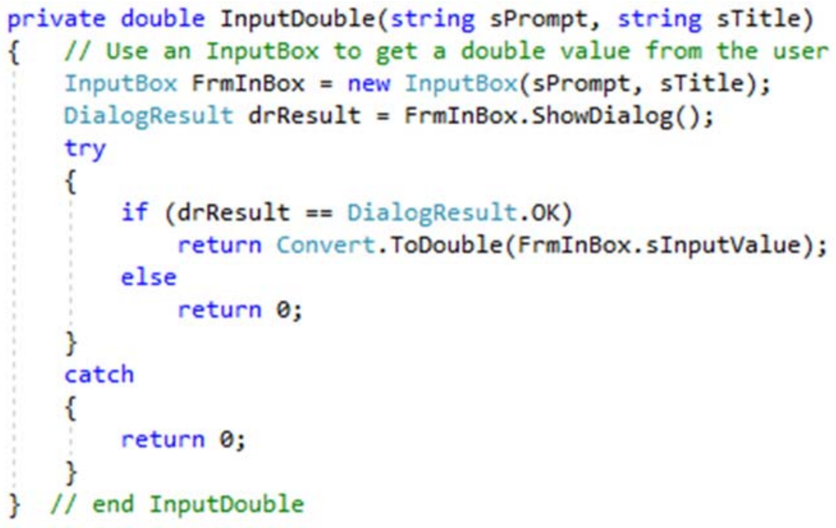
```
private void EnterAllHeights3()
{
    for (int i = 0; i < 5; i++)
    {
        dHeight[i] = InputDouble("Enter height of person " + Convert.ToString(i+1),
            "Data collection");
    }
}
```

The power of the array

- Now imagine a program to deal with 500 or 5000 heights
- The first method would be 10 or 100 times as big, while the second one would remain exactly the same size
- The power of the array lies in the programmer’s ability to use a variable as its index . . .
- . . . and thus to use loops to process each element in turn with the same small piece of code

What’s this InputDouble?

- InputDouble is not an inbuilt method; I had to write it before I could use it
- It uses an InputBox (see lecture 6), checks to see that it really does have a double in it, and returns the value



```
private double InputDouble(string sPrompt, string sTitle)
{   // Use an InputBox to get a double value from the user
    InputBox FrmInBox = new InputBox(sPrompt, sTitle);
    DialogResult drResult = FrmInBox.ShowDialog();
    try
    {
        if (drResult == DialogResult.OK)
            return Convert.ToDouble(FrmInBox.sInputValue);
        else
            return 0;
    }
    catch
    {
        return 0;
    }
}  // end InputDouble
```

Declaring and instantiating arrays

- We declare an array by giving its type, then a pair of brackets, then its name
  `double[] dHeight;`
- However, we normally instantiate the array at the same time as declaring it; to do this we call its constructor, giving the number of elements we want it to have
  `double[] dHeight = new double[26];`
- The index of the first element is always zero . . .
- . . . so the index of the last element is one less than the number of elements; the array declared and instantiated above has elements from dHeight[0] to dHeight[25]

Indexes from 0

- In C#, as in many programming languages, array indexes start at 0
- This can be confusing: the first element has an index of 0, the 53rd element has an index of 52, etc
- Why is it so? Because in another programming language from the 1960s, the designers realised that it was more *efficient* to start counting from 0 than from 1
- Now that computers are faster it would make sense to start counting from 1, but it’s hard to overcome inertia

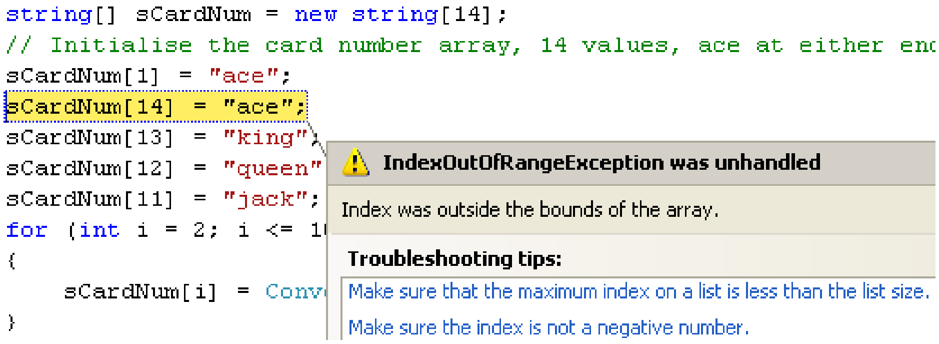
A picture of an array

- For people who prefer pictures to words, here’s a picture of a dHeight array with 5 elements. . .

the indexes	the values
0	1.52
1	1.75
2	1.93
3	1.45
4	1.72

Arrays – common errors

- Remember the common errors we mentioned with loops? Going once too many, once too few, etc?
- Those problems are particularly common with arrays, because of the difference between the number of elements in the array and the highest index in the array.
- The commonest runtime error is an attempt to access the element beyond the last one. Stay alert!



```
string[] sCardNum = new string[14];
// Initialise the card number array, 14 values, ace at either enc
sCardNum[1]  = "ace";
sCardNum[14]  = "ace";
sCardNum[13]  = "king";
sCardNum[12]  = "queen"
sCardNum[11]  = "jack";
for (int i = 2; i <= 1
{
    sCardNum[i] = Conv
}
```

IndexOutOfRangeException was unhandled

Index was outside the bounds of the array.

Troubleshooting tips:

Make sure that the maximum index on a list is less than the list size.

Make sure the index is not a negative number.

Avoiding the 0-index problem

- Because people generally count from 1, not from 0, most of us would expect an array of 10 elements to have indexes from 1 to 10, not from 0 to 9
- The good news is there’s no rule that says we have to use every element in turn . . .
- . . . so we could declare an array of *11* elements (`double[] dWeight = new double[11];`) *and pretend the first element isn’t there*!
- Then we have just what we want – a set of 10 values from dWeight[1] to dWeight[10]
- This is especially good when the indexes have a real-world meaning, as in 1-12 for the months of the year

Arrays – declare and initialise

- It’s possible to declare and initialise an array in a single statement
- If we do this, we don’t specify the size: C# deduces this from the number of elements provided

```
string[] sBeatle = {"John", "Paul",
  "George", "Ringo"};
double[] dWeight = {62.5, 70, 72.1, 85,
  92.4, 105.6, 123.4};
```

- Note that this is another use of braces, also known as ‘curly brackets’
- This sort of initialisation is of somewhat limited use – the values will more often come from input – but it can be handy now and then

Arrays – constants for bounds

- The size of an array can often be a magic number: where it appears in code, it’s not obvious why it’s that number and not some other number
- Therefore it’s always worth considering using a constant to represent the size of an array
- If you declare the constant before you declare the array, you can use the constant rather than the literal number when declaring the array . . .

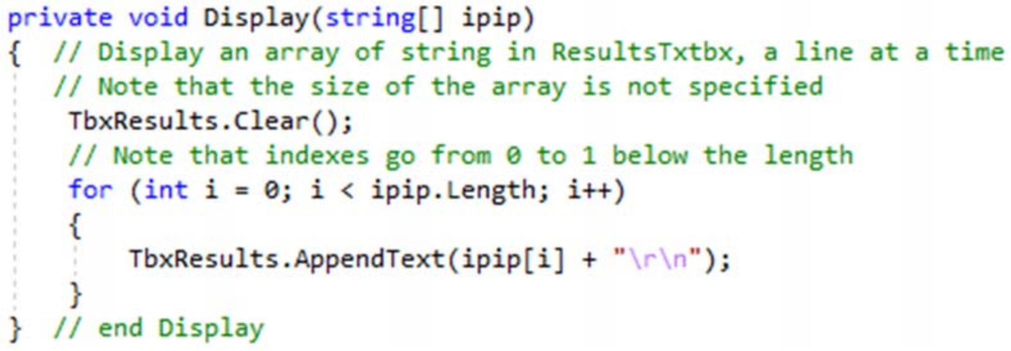
```
const int iDeckSize = 52;
string[] sCard = new string[iDeckSize];
```

Arrays as parameters

- Imagine we wanted a method to display an array of string in a textbox
- It makes sense to specify the array as a parameter, and to pass it in as an argument when calling the method
- If we specify the array size and type, the method would presumably not permit us to pass an argument of a different size and type
- C# insists that we specify the *type* of the parameter, but allows a little slack with the *size* – we don't specify that

Arrays as parameters

- Because we don't specify the size of the array, we generally have to be able to find it to know how many times to loop
- <array>.Length is one way of finding this



```
private void Display(string[] ipip)
{  // Display an array of string in ResultsTxtbx, a line at a time
   // Note that the size of the array is not specified
    TbxResults.Clear();
    // Note that indexes go from 0 to 1 below the length
    for (int i = 0; i < ipip.Length; i++)
    {
        TbxResults.AppendText(ipip[i] + "\r\n");
    }
}  // end Display
```

- (Why do you think Simon chose the name ipip? Does it mean something to him? Some generic array?)

Parallel arrays

- If we store people’s names in one array, their heights in a second array, their weights in a third array . . .
- . . . so long as we don’t shuffle the elements in any array, a particular index value refers to the same person in each array
- The person whose name is in sName[35] has the height that’s in dHeight[35] and the weight that’s in dWeight[35] – assuming we stored them that way to start with
- Arrays set up like this are called *parallel arrays*. There are better ways to achieve the same result, and we’ll see them later in this course.

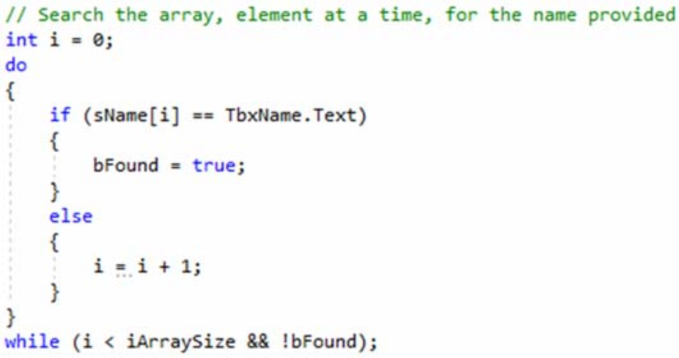
A picture of parallel arrays

- For people who prefer pictures to words, here’s a picture of our parallel arrays . . .

indexes	sName		dHeight		dWeight
0	Alexis	0	1.67	0	62.5
1	Susan	1	1.75	1	75.9
2	Devi	2	1.93	2	56.7
3	Benjamin	3	1.69	3	58.4
4	Simon	4	1.72	4	105.0
5	Mei Hui	5	1.67	5	66.3
etc		etc		etc	

Searching arrays

- If we want to know whether a particular value is stored in an array, we can examine every element (with a for loop), setting some boolean to true if it's found
- If we also want to know its index in the array – eg because we then want to look up corresponding information in parallel arrays – it makes sense to stop looping as soon as we find it



```
// Search the array, element at a time, for the name provided
int i = 0;
do
{
    if (sName[i] == TbxName.Text)
    {
        bFound = true;
    }
    else
    {
        i = i + 1;
    }
}
while (i < iArraySize && !bFound);
```

Array demonstration form

- In Lec7Demo, the Array demonstration form demonstrates most of the features of arrays that have been mentioned so far in this lecture
- It's well commented, but you should still examine the code thoroughly to see what it's doing and how
- You should also alter the code to create errors such as index overflows
- Creating errors in a controlled environment (single adjustments to a program known to be working) is a great way to learn what causes them and how they're reported
- In fact we've left a subtle bug in there for you to find!

List boxes and combo boxes

- List boxes are rather like multiline textboxes, except that the program sees each line as a separate item
- Combo boxes are very similar, but with a single-line appearance and a drop-down menu to expand them
- You can think of these boxes as rather like visual arrays of strings, but with a few more features
- The elements are actually objects, not strings, so if you want to process one as a string, you need to convert it to a string first
- And the ‘array’ is actually a *collection*, which has some rather different properties that can be really useful

Listbox properties

- Consider a list box control called *LstbxCourse* (with entries such as Inft1001, Inft1004, Inft2009, Inft2012, Desn2270…).
- Some useful properties of list boxes are
  - *LstbxCourse.Items* – the collection of objects whose strings are displayed in *lstbxCourse*
  - *LstbxCourse.SelectedIndex* – the index of the selected item; as with arrays, indexes start at 0; if no item is selected, this property returns –1
  - *LstbxCourse.SelectedItem* – the selected item itself, as an Object
  - *LstbxCourse.Text* – the string form of the selected item

Listbox Items properties

- The Items collection has many array-like properties, such as
  - *LstbxCourse.Items[2]* – array-like indexing: this is the third item in the collection
  - *LstbxCourse.Items.Count* – the number of items in the collection

Listbox Items methods

- Add

```
LstbxCourse.Items.Add("some string");
```

  - adds "some string" to the end of the collection
- Insert

```
LstbxCourse.Items.Insert(2, "string");
```

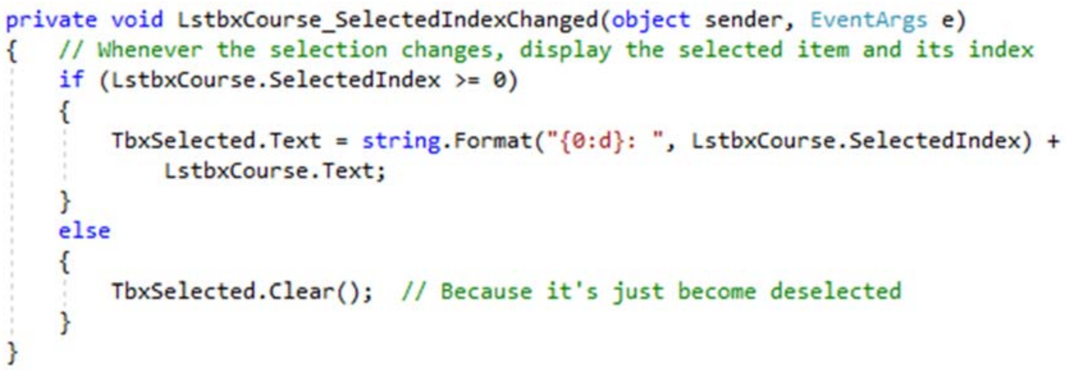
  - inserts "string" at index 2 in the collection. The item previously at index 2, and all subsequent elements, are moved up the list by one position.
- RemoveAt

```
LstbxCourse.Items.RemoveAt(5);
```

  - removes the item at index 5 in the list, moving the others down to fill the gap

Listbox events

- SelectedIndexChanged is the most common Listbox event; it is used to start code execution when a user selects a value in the Listbox
- The SelectedIndex (or the other Selected… properties) can then be used to find which element was selected



```
private void LstbxCourse_SelectedIndexChanged(object sender, EventArgs e)
{   // Whenever the selection changes, display the selected item and its index
    if (LstbxCourse.SelectedIndex >= 0)
    {
        TbxSelected.Text = string.Format("{0:d}: ", LstbxCourse.SelectedIndex) +
            LstbxCourse.Text;
    }
    else
    {
        TbxSelected.Clear();  // Because it's just become deselected
    }
}
```

Listboxes – the commonest error

- The most common mistake with listboxes?
- Some properties and methods are properties and methods of the listbox itself . . .
- . . . but many more are properties and methods of its Items collection.
- If you leave out the ".Items", it won't work!

```
LstbxCourse.Add("some string");
LstbxCourse.Insert(2, "string");
LstbxCourse.RemoveAt(5);
iNumber = LstbxCourse.Count;
```

- See the Listboxes form in Lec7Demo

2-dimensional arrays

- Imagine we want to hold the rainfall for every day of the year
- Will we declare rainfall as an int or double array with 366 elements?
- It’s not a good idea, because we identify days of the year by month and day of month, not by day of year. What date would iRainfall[290] refer to?
- Instead we use a 2-dimensional array, which has 2 indexes separated by commas

```
 int[,] iRainfall = new int[32,13];
 iRainfall[17,10] = iDayRain;
```

- It’s easier to see that this is the rainfall for 17 October

2D array continued

- It can help to consider 2D arrays as tables, grids, spreadsheets, or some other tabular format
- `iRainfall` could be considered as this table

	0	1	2	3	4	5	6	7	8	9	…….	30	31
0													
1													
: :													
12													

- (of which the first row and column would never be used)

Order of indexes

- Which order should the indexes go in?
- It doesn’t matter!
- We could equally well have declared

```
int[,] iRainfall = new int[13,32];
```

- What does matter is that we always use the indexes consistently. If we declare the array as above, but then try to access iRainfall[17, 10], we’ll get an index out of range error.
- Comments can remind us which index is which
- So can well-named variables, such as iDay and iMonth for the array index counters

Nested loops for 2D arrays

- Just as we can use a for loop to access all the elements of a 1D array, we can use nested for loops to access all the elements of a 2D array

```
for (iMonth = 1; iMonth <= 12; iMonth++)
{
   for (iDay = 1; iDay <= 31; iDay++)
   {
     iTotal=iTotal+iRainfall[iMonth, iDay];
   }
}
```

- Is this going to count extra rainfall for non-existent days such as 31 June?
- Not if the rainfall for those days remains on 0!

2D arrays as parameters

- Remember, to pass a 1D array as an argument to a method, we declared the parameter without a size

```
private void Process(string[] ipip)
```

- We do the same with 2D arrays, but we have to let C# know there are two dimensions, so we put a comma between the brackets

```
private void Process(string[,] ipip)
```

- The size of a dimension is determined by the function `ipip.GetLength(0)` for the first dimension (the first index) and `ipip.GetLength(1)` for the second dimension. Of course the highest index of the dimension is one less than the size of the dimension.

A rainfall array demonstrated

- The Rainfall form demonstrates a 2D rainfall array and illustrates a number of the points that we've made
- Note that the initial data is 'entered' by way of a tedious array initialisation
- This would never happen in real life; instead the values would be stored in a file, and the program would load them from the file when it started running
- Likewise, if the user made any changes to the data, eg adding the past week's rainfall, the program would be able to save the changes to the file
- We'll look at file access from C# programs next week, and database access in a subsequent course

John Conway’s Game of Life

- John Conway’s Game of Life is more a simulation than a game. It’s also a great illustration of a 2D array.
- It consists of a grid of cells, some of which are alive.
- To get from one generation to the next . . .
  - a living cell with 2 or 3 of its 8 neighbours living stays alive
  - a living cell with fewer than 2 living neighbours dies from isolation
  - a living cell with more than 3 living neighbours dies from overcrowding
  - a dead cell with exactly 3 living neighbours springs to life in the miracle of birth

A boolean array

- The game of life calls for a boolean array – true in a location means it’s alive, false means it’s dead
- Operations on the array cells include counting their live neighbours and changing their state from dead to alive or vice versa
- Care must be taken not to access cells beyond the edge
- Operations on the array as a whole include working out the next generation and then changing to that generation

Game of life demonstrated

- The Game of life form in Lec7Demo demonstrates the game of life
- The game has been studied at great length (eg see http://math.com/students/wonders/life/life.html) and many interesting patterns have been discovered
- The demo allows you to create your own starting pattern, or to explore 7 starting patterns that have been programmed in
- Have fun with the demo, but also explore its use and illustration of arrays – there's a lot to take in!
- And if you decide to explore the Game of Life a little more, try golly.sourceforge.net – it's amazing