

# PCP — Lecture 14

Fall 2020 November 6, 2020

## 1 Arrays

### 1.1 Motivation

Arrays are collection, or grouping, of values held in a single place. They can store multiple values of the same datatype, and are useful, for instance,

- When we want to store a collection of related values,
- When we don't know in advance how many variables we need.

### 1.2 Declaration and Initialization of Arrays

Declaration and assignment

```

1  int[] myArray;
2  myArray = new int[3]; // 3 is the size declarator
3  // We can now store 3 ints in this array,
4  // at index 0, 1 and 2
5
6  myArray[0] = 10; // 0 is the subscript, or index
7  myArray[1] = 20;
8  myArray[2] = 30;
9
10 // the following would give an error:
11 //myArray[3] = 40;
12 // Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of
   ↳ the array at Program.Main()
13 // "Array bound checking": happen at runtime.
```

As usual, we can combine declaration and assignment on one line:

```
1  int[] myArray = new int[3];
```

We can even initialize *and* give values on one line:

```
1  int[] myArray = new int[3] { 10, 20, 30 };
```

And that statement can be rewritten as any of the following:

```

1  int[] myArray = new int[] { 10, 20, 30 };
2  int[] myArray = new[] { 10, 20, 30 };
3  int[] myArray = { 10, 20, 30 };
```

But, we should be carefull, the following would cause an error:

```

1  int[] myArray = new int[5];
2  myArray = { 1, 2 ,3, 4, 5}; // ERROR

```

If we use the shorter notation, we *have to* give the values at initialization, we cannot re-use this notation once the array was created.

Other datatype, and even objects, can be stored in arrays:

```

1  string[] myArray = { "Bob", "Mom", "Train", "Console" };
2  Rectangle[] arrayOfRectangle = new Rectangle[5];

```

### 1.3 Custom Size and Values

```

1  Console.WriteLine("What is size of the array that you want?");
2  int size = int.Parse(Console.ReadLine());
3  int[] customArray = new int[size];

```

How can we fill it with values, since we do not know its size? Using iteration!

```

1  int counter = 0;
2  while (counter < size)
3  {
4      Console.WriteLine($"Enter the {counter + 1}th value");
5      customArray[counter] = int.Parse(Console.ReadLine());
6      counter++;
7  }

```

We can use `length`, a property of our `array`. That is, the integer value `myArray.Length` is the length (= size) of the array, we can access it directly.

To display an array, we need to iterate as well (this time using the `Length` property):

```

1  int counter2 = 0;
2  while (counter2 < customArray.Length)
3  {
4      Console.WriteLine($"{counter2}: {customArray[counter2]}.");
5      counter2++;
6  }

```

### 1.4 Changing the Size

Array is actually a class, and it comes with methods!

```

1  Array.Resize(ref myArray, 4);
2  myArray[3] = 40;
3  Array.Resize(ref myArray, 2);

```

`Resize` shrinks (and content is lost) and extends (and store the default value, i.e., 0 for `int`, etc.)!

## 2 For Loops

### 2.1 for Loops

```
1  int i = 0;
2  while (i <= 5)
3  {
4      Console.Write(i + " ");
5      i++;
6  }

1  int j = 0;
2  do
3  {
4      Console.Write(j + " ");
5      j++;
6  } while (j <= 5);

1  int k = 0;
2  for (k = 0; k <= 5; k++)
3  {
4      Console.Write(k + "");
5  }

1  for (int l = 0; l <= 5; l++)
2  {
3      Console.Write(l + "");
4  }
```

Structure : initialization / condition / update

### 2.2 Ways Things Can Go Wrong

Don't:

- Increment the counter in the body of the for loop!
- Assume that a variable declared in the header of a for loop will be accessible in the rest of the code. / Use `for` if you want to use the counter for anything else.
- Declare the variable twice.

### 2.3 For loops With Arrays

`for` loops actually go very well with arrays:

```
1  for (int i = 0; i < size; i++)
2  {
3      Console.WriteLine($"Enter the {i + 1}th value");
4      customArray[i] = int.Parse(Console.ReadLine());
5  }
```

Remember that we can use the `Length` property of our `array`. The previous code could become (only the first line changed):

```
1  for (int i = 0; i < customArray.Length; i++)
2  {
3      Console.WriteLine($"Enter the {i + 1}th value");
4      customArray[i] = int.Parse(Console.ReadLine());
5  }
```

## 2.4 Nested Loops

Of course, exactly as we could nest `if` statements, we can nest looping structures!

```
1  for (int o = 0; o < 11; o++)
2  {
3      for (int p = 0; p < 11; p++)
4          Console.Write($"{o} × {p} = {o * p} \t ");
5      Console.WriteLine();
6  }
```

## 2.5 Mixing Control Flows

And we can use `if` statements in the body of `for` loops:

```
1  for (int m = 0; m < 10; m++)
2  {
3      if (m % 2 == 0) Console.WriteLine("This is my turn.");
4      else Console.WriteLine("This is your turn.");
5  }
```

## 2.6 Iterations

There is another, close, structure that allows to iterate over the elements of an array, but can only access them, not change their values (they are “read only”).

```
1  for (int i = 0; i < myArray.Length; i++)
2      Console.Write(myArray[i] + " ");
3
4  foreach (int i in myArray) // "Read only"
5      Console.Write(i + " ");
```

Difference is w.r.t. to modifying the array “read Vs write”. Having `i = 2` in the `foreach` would cause an error!

That last structure is given for the sake of completeness, but it’s ok if you’d rather not use it.