Contents

Arrays

Arrays are structures that allow you to store multiple values in memory using a single name and indexes.

- Usually all the elements of an array have the same type.
- You limit the type of array elements when you declare the array.
- If you want the array to store elements of any type, you can specify object as its type.

An array can be:

- Single-Dimensional,
- Multidimensional,
- Jagged.

Arrays are useful, for instance,

- When you want to store a collection of related values,
- When you do not know in advance how many variables will be needed,
- When you need a large number of variables (say, 10) of the same type.

Single-Dimensional Arrays

You can define a single-dimensional array as follow:

```
<type>[] arrayName;
```

where

- `<type>` can be any data-type and specifies the data-type of the array elements.
- `arrayName` is an identifier that you will use to access and modify the array elements.

Before using an array, you must specify the number of elements in the array as follows:

```
arrayName = new <type>[<number of elements>];
```

where `<type>` is a type as before, and `<number of elements>`, called the *size declarator*, is a strictly positive integer which will correspond to the size of the array.

- An element of a single-dimensional array can be accessed and modified by using the name of the array and the index of the element as follows:

  ```
  arrayName[<index>] = <value>;  // Assigns <value> to the <index> element of the

  Console.WriteLine(arrayName[<index>]); // Display the <index> element of the a
  ```

The index of the first element in an array is always *zero*; the index of the second element is one, and the index of the last element is the size of the array minus one. As a consequence, if you specify an index greater or equal to the number of elements, a run-time error will happen.

Indexing starting from 0 may seem surprising and counter-intuitive, but this is a largely respected convention across programing languages and computer scientists. Some insights on the reasons behind this (collective) choice can be found in this answer on Computer Science Educators[1].

Example

In the following example, we define an array named *myArray* with three elements of type integer, and assign 10 to the first element, 20 to the second element, and 30 to the last element.

```
int[] myArray;
myArray = new int[3]; // 3 is the size declarator
// We can now store 3 ints in this array,
// at index 0, 1 and 2

myArray[0] = 10; // 0 is the subscript, or index
myArray[1] = 20;
myArray[2] = 30;
```

If we were to try to store a fourth value in our array, at index 3, using e.g.

[1] https://cseducators.stackexchange.com/a/5026

```
myArray[3] = 40;
```

our program would compile just fine, which may seems surprising. However, when executing this program, *array bounds checking* would be performed and detect that there is a mismatch between the size of the array and the index we are trying to use, resulting in a quite explicit error message:

```
Unhandled Exception: System.IndexOutOfRangeException:
 ↪  Index was outside the bounds of the array at
 ↪  Program.Main()
```

Abridged Syntaxes

If you know the number of elements when you are defining an array, you can combine declaration and assignment on one line as follows:

```
<type>[] arrayName = new <type>[<number of elements>];
```

So, we can combine the first two lines of the previous example and write:

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

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

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

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

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

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

```
int[] myArray = new int[5];
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 has been created.

Other datatypes, and even objects, can be stored in arrays in a perfectly similar way:

```
string[] myArray = { "Bob", "Mom", "Train", "Console" };
Rectangle[] arrayOfRectangle = new Rectangle[5]; // Assume there is a class called
```

Default Values

If we initialize an array but do not assign any values to its elements, each element will get the default value for that element's data type. (These are the same default values that are assigned to instance variables if we

do not write a constructor, as we learned in "More Advanced Object Concepts"). In the following example, each element of `myArray` gets initialized to 0, the default value for `int`:

```
int[] myArray = new int[5];
Console.WriteLine(myArray[2]); // Displays "0"
myArray[1]++;
Console.WriteLine(myArray[1]); // Displays "1"
```

However, remember that the default value for any *object* data type is `null`, which is an object that does not exist. Attempting to call a method on a `null` object will cause a run-time error of the type `System.NullReferenceException`;

```
Rectangle[] shapes = new Rectangle[3];
shapes[0].SetLength(5);  // ERROR
```

Before we can use an array element that should contain an object, we must instantiate an object and assign it to the array element. For our array of `Rectangle` objects, we could either write code like this:

```
Rectangle[] shapes = new Rectangle[3];
shapes[0] = new Rectangle();
shapes[1] = new Rectangle();
shapes[2] = new Rectangle();
```

or use the abridged initialization syntax as follows:

```
Rectangle[] shapes = {new Rectangle(), new Rectangle(), new Rectangle()};
```

Custom Size and Loops

One of the benefits of arrays is that they allow you to specify the number of their elements at run-time: the size declarator can be a variable, not just an integer literal. Hence, depending on run-time conditions such as user input, we can have enough space to store and process any number of values.

In order to access the elements of whose size is not known until run-time, we will need to use a loop. If the size of `myArray` comes from user input, it wouldn't be safe to try to access a specific element like `myArray[5]`, because we cannot guarantee that the array will have at least 6 elements. Instead, we can write a loop that uses a counter variable to access the array, and use the loop condition to ensure that the variable does not exceed the size of the array.

Example

In the following example, we get the number of elements at run-time from the user, create an array with the appropriate size, and fill the array.

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

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

Observe that:

- If the user enters a negative value or a string that does not correspond to an integer for the `size` value, our program will crash: we are not performing any user-input validation here, to keep our example compact.
- The loop condition is `counter < size` because we do *not* want the loop to execute when `counter` is equal to `size`. The last valid index in `customArray` is `size - 1`.
- We are asking for the `{counter +1}th` value because we prefer not to confuse the user by asking for the "0th" value. Note that a more sophisticated program would replace "th" with "st", "nd" and "rd" for the first three values.

The Length Property

Every single-dimensional array has a property called `Length` that returns the number of the elements in the array (or size of the array).

To process an array whose size is not fixed at compile-time, we can use this property to find out the number of elements in the array.

Example

```
int counter2 = 0;
while (counter2 < customArray.Length)
{
   Console.WriteLine($"{counter2}: {customArray[counter2]}.");
    counter2++;
}
```

Observe that this code does not need the variable `size`.

Note: You *cannot* use the length property to change the size of the array, that is, entering

```
int[] test = new int[10];
test.Length = 9;
```

would return, at compile time,

```
Compilation error (line 8, col 3): Property or indexer 'System.Array.Length' cannot
-it is read only.
```

When a field is marked as 'read only,' it means the attribute can only be initialized during the declaration or in the constructor of a class. We receive this error because the array attribute, 'Length,' can not be changed once the array is already declared. Resizing arrays will be discussed in the section: Changing the Size.

Loops with Arrays of Objects

In the following example, we will ask the user how many `Item` objects they want to create, then fill an array with `Item` objects initialized from user input:

```
Console.WriteLine("How many items would you like to stock?");
Item[] items = new Item[int.Parse(Console.ReadLine())];
int i = 0;
while(i < items.Length)
{
   Console.WriteLine($"Enter description of item {i+1}:");
    string description = Console.ReadLine();
    Console.WriteLine($"Enter price of item {i+1}:");
    decimal price = decimal.Parse(Console.ReadLine());
    items[i] = new Item(description, price);
    i++;
}
```

Observe that, since we do not perform any user-input validation, we can simply use the result of `int.Parse()` as the size declarator for the `items` array - no `size` variable is needed at all.

We can also use `while` loops to search through arrays for a particular value. For example, this code will find and display the lowest-priced item in the array `items`, which was initialized by user input:

```
Item lowestItem = items[0];
int i = 1;
while(i < items.Length)
{
```

```
    if(items[i].GetPrice() < lowestItem.GetPrice())
    {
        lowestItem = items[i];
    }
    i++;
}
Console.WriteLine($"The lowest-priced item is {lowestItem}");
```

Note that the `lowestItem` variable needs to be initialized to refer to an `Item` object before we can call the `GetPrice()` method on it; we cannot call `GetPrice()` if `lowestItem` is `null`. We could try to create an `Item` object with the "highest possible" price, but a simpler approach is to initialize `lowestItem` with `items[0]`. As long as the array has at least one element, `0` is a valid index, and the first item in the array can be our first "guess" at the lowest-priced item.

Changing the Size

There is a class named `Array` that can be used to resize an array. Upon expanding an array, the additional indices will be filled with the default value of the corresponding type. Shrinking an array will cause the data in the removed indices (those beyond the new length) to be lost.

Example

```
Array.Resize(ref myArray, 4); //myArray[3] now contains 0
myArray[3] = 40;
Array.Resize(ref myArray, 2);
```

In the above example, all data starting at index 2 is lost.

For Loops With Arrays

- Previously, we learned that you can iterate over the elements of an array using a `while` loop. We can also process arrays using `for` loops, and in many cases they are more concise than the equivalent `while` loop.
- For example, consider this code that finds the average of all the elements in an array:

```
int[] homeworkGrades = {89, 72, 88, 80, 91};
int counter = 0;
int sum = 0;
while(counter < 5)
{
    sum += homeworkGrades[counter];
```

```
    counter++
}
double average = sum / 5.0;
```

- This can also be written with a `for` loop:

```
int sum = 0;
for(int i = 0; i < 5; i++)
{
    sum += homeworkGrades[i];
}
double average = sum / 5.0;
```

- In a `for` loop that iterates over an array, the counter variable is also used as the array index
- Since we did not need to use the counter variable outside the body of the loop, we can declare it in the loop header and limit its scope to the loop's body
- Using a `for` loop to access array elements makes it easy to process "the whole array" when the size of the array is user-provided:

```
Console.WriteLine("How many grades are there?");
int numGrades = int.Parse(Console.ReadLine());
int[] homeworkGrades = new int[numGrades];
for(int i = 0; i < numGrades; i++)
{
    Console.WriteLine($"Enter grade for homework {i+1}");
    homeworkGrades[i] = int.Parse(Console.ReadLine());
}
```

- You can use the `Length` property of an array to write a loop condition, even if you did not store the size of the array in a variable. For example, this code does not need the variable `numGrades`:

```
int sum = 0;
for(int i = 0; i < homeworkGrades.Length; i++)
{
    sum += homeworkGrades[i];
}
double average = (double) sum / homeworkGrades.Length;
```

- In general, as long as the loop condition is in the format `i < <arrayName>.Length` (or, equivalently, `i <= <arrayName>.Length - 1`), the loop will access each element of the array.