Lesson 9: Arrays

In a nutshell, this lesson will cover:

- What are Arrays?
- Introducing User-Defined Data Types
- 2 Dimensional and Multi-Dimensional Arrays

What are Arrays?

An array is a highly useful data structure that stores variable data having the *same*data type. It is just like a small fixed number of boxes linked together one after the other storing things that are related to each other. An array is said to be a *static* data structure because, once declared, its original size that is specified by the programmer will remain the same throughout the whole program and cannot be changed.

Up until now, we have used single variables only as a tool to store data. Now we will be using the array data structure and here is how it is declared:

Var

*<arrayName> : **Array**[n..m] **of** <Data Type>;*

*myArray : **Array**[1..20] **of** Integer;*

An array data structure defines the size of the array and the data type that it will use for storing data. In the above example, the array stores up to 20 integers however I may have used 30 integers or more. This size depends on your program requirements.

Arrays are used just like ordinary variables. They are used to store typed data just like the ordinary variables. You will now learn how to assign data to arrays and read data from arrays.

In the example above, I have declared 20 integers and I should be able to access each and one of them and here is how I do it.

To assign values to a particular integer of an array, we do it like this:

```
myArray[5] := 10;
myArray[1] := 25;
<arrayName>[index] := <value>;
```

You just take the array in subject, specify the index of the variable of the array and assign it a value relevant to the data type of the array itself.

Reading a value from an array is done as follows:

```
Var
          myVar : Integer;
          myArray : Array[1..5] of Integer;

Begin
          myArray[2] := 25;
          myVar := myArray[2];
End.
```

Just like ordinary variables, arrays can be initialised with any value you want, otherwise they will be assigned with their default value. If we want to intialise 2 entire integer and boolean arrays of size 20 to 1 and true respectively, we do it like this:

```
Var
          i : Integer;
          myIntArray : Array[1..20] of Integer;
          myBoolArray : Array[1..20] of Boolean;

Begin
          For i := 1 to Length(myIntArray) do
          Begin
                    myIntArray[i] := 1;
                    myBoolArray[i] := True;
          End;
End.
```

Note that if these are arrays were not initialised, integer array elements would be assigned to 0 and false to boolean arrays.

An interesting function which has been just introduced in the code above is the `Length` function. This function very much co-operates with collections, meaning can be both used with arrays and even Strings! The `Length` function returns the size of the number of items which was allocated for a collection. For example, in the code above it will return 20 and therefore the loop will iterate through every item in the array.

When working with Strings, the `Length` function will simply return the length of the string (that is, number of characters).

Introducing User-Defined Data Types

Now that we have used various built-in data types, we have arrived at a point were we might want to use our defined data types. Built-in data types are those which come part and parcel with the programming language, such as **Integer**, **Boolean** and **String**. Now you should learn how to specify our own customised data types and here is the syntax:

Type
<myDataType> = <particularDataType>;

The "***Type***" keyword is a Pascal reserved word used to define our own data types. So you start defining your own data types by using this keyword, specify a name for your data type and then create a *customized* data type. After defining your own data types, you may start using them just like the other built-in data types as follows:

Var
<myVar> : <myDataType>;

Now lets define a new simple data type and note how it will be used in the program below:

```
Type
          nameType = String[50];
          ageType  = 0..150; { age range: from 0 to 150 }

Var
          name : nameType;
          age  : ageType;

Begin
          Write('Enter your name: ');
          Readln(name);
          Write('Enter your age: ');
          Readln(age);
          Writeln;
          Writeln('Your name:', name);
          Writeln('Your age :', age);
          Readln;
End.
```

In the above example we defined a String[50] and a 0..150 data type. The nameType only stores strings up to 50 characters long and the ageType stores numbers only from 0 to 150.

We can define more complex user-defined data types. Here is an example of more complex user-defined data types:

```
Type
          i = 1..5;
          myArrayDataType = Array[1..5] of Byte;
          byteFile        = File of Byte; { binary file }

Var
          myArrayVar : myArrayDataType;
          myFile     : byteFile;

Begin
          Writeln('Please enter 5 numbers from (0..255): ');

          For i := 1 to Length(myArrayVar) do
                    Readln(myArrayVar[i]);

          Writeln('You have entered the following numbers: ');

          For i := 1 to Length(myArrayVar)do
                    Writeln('Number ',i,': ',myArrayVar[i]);

          Writeln('Now writing them to file...');   {store the numbers in a file}
          Assign(myFile, 'example.dat');
          ReWrite(byteFile);
          Write(myFile, myArrayVar[i]);
          Close(myFile);
          Writeln('Done, you may exit..');
          Readln;
End.
```

In the above example I showed you how to incorporate arrays as user-defined data types. Note that you may use user-defined data types more than once.

Back To Top ⇧

2 Dimensional and Multi-Dimensional Arrays

2 Dimensional arrays and multi-dimensional are arrays which store variables in a second or nth dimension having n*m storage locations. Mutli dimensional arrays including the 2 dimensional array, are declared by using multiple square brackets placed near each other or using commas with one sqaure brackets as an alternative.

Here is how multi-dimensional are declared:

```
my2DArray : Array[i..j, k..l] of <DataType>;
myMultiDimArray : Array[m..n, o..p, q..r, s..t]... of <DataType>;
```

Let us have the 2 dimensional array defined first. Think of a grid where each box is located by using horizontal and vertical coordinates just in the example below:

1	2	3	4	5
2				
3			3,4	
4				
5		5,3		

Let us declare an array having 3 by 5 dimensions, assign a value to a particular variable in the array and illustrate this on a grid just like the one above:

```
Var
        my2DArray : Array[1..3, 1..5] of Byte;

Begin
        my2DArray[2][4] := 10;
End.
```

Having the vertical axis as the 1st dimension and the horizontal one as the 2nd dimension, the above example can be illustrated as follows:

1	2	3	4	5
2			**10**	
3				

Multi-dimensional arrays are rare and are not important. The single and 2D dimensional arrays are the 2 most frequent dimensions.

The following example is a bit more complex example than the previous examples and it also uses a 2 dimensional array to illustrate their use.

```
Uses Crt;
Type
          myRange      = 1..5;
          arrayIntType = Array[myRange] of Integer;
          myFileType   = File of arrayIntType;

Var
          i        : myRange;
          myFile   : myFileType; { the next array is 2 dimensional }
          arrayInt : Array[1..2] of arrayIntType;

Begin
          Clrscr;
          Randomize;

          For i := 1 to 5 do { or we can use Length(arrayInt[1][i]) }
          Begin
                    arrayInt[1][i] := Random(1000);
                    Writeln('rand num: ',arrayInt[1][i]);
          End;

          Assign(myFile, 'test.dat');
          ReWrite(myFile);
          Write(myFile, arrayInt[1]);
          Close(myFile);
          ReSet(myFile);
          Read(myFile, arrayInt[2]);
          Close(myFile);

          For i := 1 to 5 do
                    Writeln(i,': ', arrayInt[2][i]);

          Readln;
End.
```

This concludes the arrays lesson. In the next lesson we will learn about Record data structures, their uses and we will also learn how to use binary files used record data structures.