Introduction to Systematic Programming
Unit 10 - FOR Loops and arrays

10.1 Pre-determinable repetitions

We have seen throughout the earlier units that the WHILE construction allows us to program any sort of repetition. However, it is often the case in programming that the number of repetitions required is known before the repetition process starts. When this is the case, it is usually neater to use a different Ada repetition construction, namely the FOR step (or **FOR loop**), instead of a WHILE. Here is a simple example. Suppose we want to produce five lines of stars in our output. Following the approach of Unit 5 we would write:

```
LineNo := 1;
WHILE LineNo <= 5 LOOP
     Put("**************************************");
     New_Line;
     LineNo := LineNo + 1;
END LOOP;
```

Using a FOR loop instead, we could write:

```
FOR LineNo IN 1..5 LOOP
     Put("**************************************");
     New_Line;
END LOOP;
```

The effect of this is to repeat the Put and New_Line steps, with LineNo set successively (on each repetition) to the values 1, 2, 3, 4, and 5; ie. to obey the Put and New_Line steps 5 times. The variable used to do the 'counting' here LineNo, is often known as the **loop control** (or **index**) **variable**.

10.1.1 FOR loops - basic form

The simplest kind of Ada FOR statement has the form:

```
FOR Index_variable IN Start_expression .. Finish_expression LOOP
     Step(s)_to_be_repeated;
END LOOP;
```

Such a statement means:

i) Evaluate the *Start_expression*.
ii) Evaluate the *Finish_expression*.
iii) Assign the value of the *Start_expression* to the *Index_variable*.
iv) Compare the index variable's value with the finishing value;
if the *Index_variable* is greater than the value of the *Finish_expression*, the execution of the FOR step is terminated; otherwise (ie. if the *Index_variable* is less than or equal to the value of the *Finish_expression*) the *Step(s)_to_be_repeated* part is obeyed, and then 1 is added to the *Index_variable*.
v) The repetition process is then continued (from (iv)).

We see from the above that the start expression and the finish expression do not have to be literal constants (as they were in our first example above), but rather they can be expressions. Thus for example, if the Integer variables Margin and Length hold the values 8 and 10 respectively, then the program fragment (taken from the DrawLine procedure in [5.7]):

```
-- Output a blank left margin
FOR Index IN 1..Margin LOOP
     Put(" ");
END LOOP;
-- Output a line of stars
FOR Index IN Margin + 1 .. Margin + Length LOOP
     Put("*");
END LOOP;
New_Line;
```

will, in the first repetition output 8 (ie. the value of Margin) spaces, and then in the second repetition output 10 stars. Notice that the second repetition starts off with Index initially set to 9 (ie. the value of Margin+1), and has Index incremented by one on each repetition until it reaches 18 (ie. the value of Margin+Length), giving 18 - 9 + 1 = 10 repetitions.

It is important to note from the above description that if the value of the finishing expression is less than that of the starting expression, then the step(s) to be repeated part will not be repeated at all, not even once. Hence in the fragment:

```
FOR Index IN Margin + 1 .. Margin + Length LOOP
     Put("*");
END LOOP;
```

if the variables Margin and Length hold the values 25 and 0 respectively, then the value of the starting expression will be 26, the value of the finishing expression will be 25, and hence the Put step will not be performed at all.

It may also be noted that the above description states that the finish expression is evaluated *before* the repetitions start (*not each time through the loop*). For example, assuming the variable Margin has the value 10, the loop below is executed 10 times (not 6 times).

```
FOR Index IN 1..Margin LOOP
   Put(Item => " ");
   Margin := Margin - 1;   -- ?? Very poor style in Ada
END LOOP;
```

Although the description of a general FOR loop above makes the meaning of this example clear, the careless reader may incorrectly infer that the step which reduced Margin by 1 is affecting the number of repetitions of the FOR loop. For this reason, the above example is considered to be very poor Ada program style.

10.1.2 Using the index variable

In the above examples, the steps being repeated do not make any reference to the index variable, however this doesn't have to be the case. For example we could use the fragment:

```
FOR Line IN 1..60 LOOP
     Put(Item => Line, Width => 2);
     New_Line;
END LOOP;
```

to output the numbers 1, 2, 3, 4, ..., 60 at the start of each line on a page of output, ie. to number the first 60 lines on a page.

This illustrates that the current *value* of the index variable can be *used* (eg. written out, or employed in some calculation) in the *Step(s)_to_be_repeated* part. However, *no attempt may be made to change it explicitly* (whether by an assignment or a Get statement). Only the "behind the scenes" 'counting' mechanism of the FOR loop (which automatically arranges for it to take each value in turn on each repetition) is allowed to change the index variable. Thus for example:

```
FOR Line IN 1..60 LOOP
   Put(Item => Line, Width => 2);
   Line := Line + 1;                    -- !? Illegal in Ada
END LOOP;
```

would result in a compilation error. In fact the term index variable is somewhat misleading, it isn't a variable at all, but rather its behaviour inside the loop is that of a local constant (ie. local to the *Step(s)_to_be_repeated* part of the loop), whose value is re-defined on each repetition to be one more than its previous value. Hence the index variable must not be used as an OUT or IN OUT actual parameter in a procedure call (since otherwise the procedure would be able to change the value of the index variable).

We see that the index variable is established and manipulated in a special way in the Ada language, a way quite unlike any other declared constant or variable. It also has two other unusual features:

i) The index variable is not declared in the usual way, its existence within the FOR loop is deduced by the computer purely by stating its name between the keywords FOR and IN in the leading part of the FOR step. A consequence of this implied declaration is that the index variable only exists while the loop is being executed, and hence cannot be referred to outside the FOR loop.

ii) The start expression and the finish expression should yield values of the same type, and the type of the index variable will be inferred from this type. Thus in the example:

```
FOR Year IN StartYear..FinishYear LOOP
     Put(Year);
     New_Line;
END LOOP;
```

if the variables StartYear and FinishYear are Integer variables (holding the values 1988 and 1993 respectively), then Year will be inferred to also be of type Integer, and the loop will write out the integer values 1988, 1989, 1990, 1991, 1992 and 1993, each at the start of a new line.

We will see later in the course what other types of values may be used as the 'counting' mechanism of a FOR loop, but for the present we will only consider examples where the index variable is of type Integer. However we should note at this point that it is not permitted for the index variable to be a real value, ie. it is not permitted to be of type Float.

Example to illustrate the above

A particular illustration of (i) above occurs when a FOR loop is used within a procedure. In this case the index variable *must not* be declared as a local variable of that procedure (as might normally be expected for other variables which are used locally within the body of the procedure). Thus to define a procedure corresponding to the first example of [10.2] we would write:

```
PROCEDURE DrawStarLine (Margin : IN Integer := 0;
                        Length : IN Integer := 20) IS
     -- To output a margin and a line of stars
BEGIN
     -- Output a blank left margin
     FOR Index IN 1..Margin LOOP
          Put(" ");
     END LOOP;
     -- Output a line of stars
     FOR Index IN Margin + 1 .. Margin + Length LOOP
          Put("*");
     END LOOP;
     New_Line;
END DrawStarLine;
```

and it is not necessary to declare the index variable Index locally within the procedure (or anywhere else for that matter). We could then define the procedure:

```
PROCEDURE DrawTriangle (Base : IN Integer);
     -- To output a triangle of stars, Base lines deep
     -- with a margin of 10 blank spaces
BEGIN
     FOR LineNo IN 1..Base LOOP
          DrawStarLine(Margin => 10, Length => LineNo);
     END LOOP;
END DrawTriangle;
```

This example again illustrates that it is not necessary to declare the index variable LineNo locally within the procedure. Also observe that this procedure uses the index variable as an actual parameter in a call to the procedure DrawStarLine. This is consistent with the fact that we may only use the *value* of the index variable, ie. as an actual parameter to a formal IN parameter. If we were to call the DrawTriangle procedure to output a triangle of stars using:

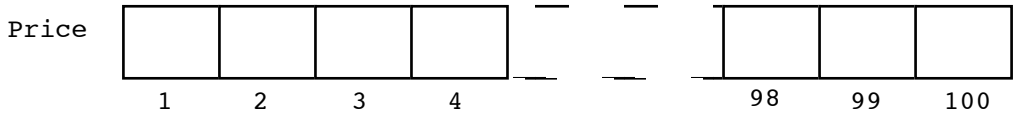
```
Get(Size);
DrawTriangle(Size);
```

(assuming the existence of an `Integer` variable `Size`) then notice that although we don't know the number of repetitions when the program is *written*, the number of repetitions is known during program execution *before* the loop starts - it is simply whatever value is read and stored in the variable `Size` and then passed to `Base` on entry to the procedure `DrawTriangle`. (Of course, if this value is zero, the *Step(s)_to_be_repeated* part of the loop (ie. the `DrawStarLine` step) is *not executed at all* - and hence no triangle of stars will be output).

10.2 Arrays

So far, we have dealt with individual data values which could be stored in individual variables; one data value per variable. However, in reality it is very common that data consists not of individual values, but of collections of similar or related values. We may have a collection of observations, eg. the 365 values comprising the rainfall on each day of the year, or a collection of selling prices (of perhaps a 100 different lines sold in a supermarket). It is neither convenient nor sensible to declare a large number of individual variables to hold such related values; rather it is preferable to set up one variable, with a single overall name, which can store *many* similar values; this will allow us to handle our related values in a simple, systematic manner. Such a variable is called an **array variable** (or simply an **array**).

Thus for the example of the selling prices of a 100 different lines sold in a supermarket, we might have an array variable called `Price` (say), within which 100 values could be stored, each of these 100 values being of type `Float` (say). This could be visualised as:



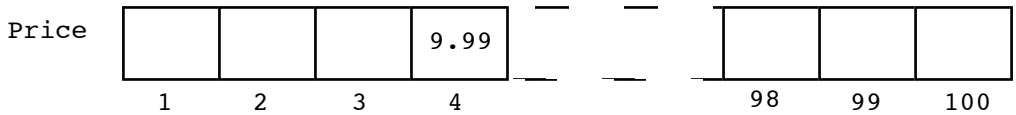
The individual storage locations of an array variable are known as (**array**) **elements**. Each element can be used exactly like an ordinary individual variable containing a single value; thus elements can be given values (eg. by using assignment or `Get` steps) and values of elements can be used (eg. in some calculation, or output by using a `Put` step). In order that the individual elements can be referred to conveniently, they are numbered; in this example we might number them from `1` to `100`. To access any particular element, its reference number is written in *round* brackets after the array name, eg:

```
Price(4)
```

refers to the element labelled with the number `4` in the above structure. Hence the assignment step:

```
Price(4) := 9.99;
```

can be used to assign a value to `Price(4)` (ie. to store the value `9.99` in the element labelled with `4`), and thus produce the storage arrangement:



Subsequently the assignment step:

```
Charge := Price(4) * (1.0 + VATRate / 100.0);
```

shows how the value of `Price(4)` (ie. the value `9.99`) can be used in a calculation (this assumes that `Charge` and `VATRate` are of course `Float` variables).

The reference number of an array element is known as its **subscript** (or sometimes its **index**), and the process of referring to a particular element of an array is known as **subscripting** (or **indexing**).

A subscript does not need to be a constant (such as `4` above); it can also be a variable or indeed any expression provided that it yields a value for the subscript which lies in the correct range; in our example above, in the range `1` to `100`. So we might refer to:

`Price(Line)` - assuming that `Line` is an `Integer` variable.

and:

`Price(2*Count+1)` - assuming that `Count` is an `Integer` variable.

A subscript expression is worked out afresh each time that the program step in which it appears is executed. Thus for example:

```
Price(Line) := 0.0;
```

means (on whatever occasion that it is executed):

i) Get the current value of the variable `Line`.
ii) See which element of `Price` has that value as its subscript.
iii) Store the value `0.0` in that element.

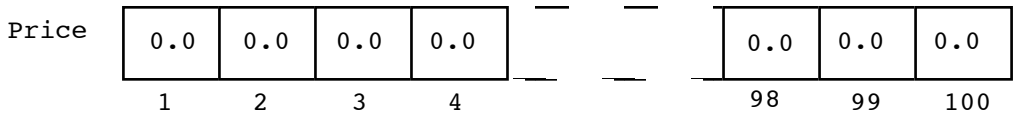
It is obvious that this only makes sense if the value of `Line` is between `1` and `100` (inclusive), as only elements numbered in this range exist. Attempting to access a non-existent array element causes a run-time error. This is frequent programming error; so be careful!

10.2.1 Processing all the elements of an array using a FOR loop

Since the logic of using arrays is that they are used to store related values, it is very common to want to apply the same operation to all of the elements of an array, ie. to each of the related values. This is very easily accomplished using a 'counting' (ie. `FOR`) repetition. For example to set all the elements of the array `Price` to zero we would write:

```
FOR Line IN 1..100 LOOP
    Price(Line) := 0.0;
END LOOP;
```

which would produce the storage arrangement:



To see how this is achieved, observe that the first time around the `FOR` loop the index variable `Line` holds the value `1`, and hence the assignment step causes `0.0` to be stored in element `1` of the `Price` array. The second time around the `FOR` loop the index variable `Line` holds the value `2`, and hence the assignment step causes `0.0` to be stored in element `2` of the `Price` array. The third time around the `FOR` loop the index variable `Line` holds the value `3`, and hence the assignment step causes `0.0` to be stored in element `3` of the `Price` array. And so on. The last time around the `FOR` loop the index variable `Line` holds the value `100`, and hence the assignment step causes `0.0` to be stored in element `100` of the `Price` array. Thus producing in the array `Price` the values as shown above.

This example illustrates some of the simplification achieved by using an array. If the `Price` array were not used, instead we would first of all have to declare `100` separate variables (say `Price1`, `Price2`, `Price3`, `Price4`, ... etc.) and then write `100` separate assignment steps (ie. `Price1:=0.0; Price2:=0.0; Price3:=0.0;` ... etc.). Very tedious!

We can use a similar repetition to input 100 data values (corresponding to the prices of the 100 different lines sold in a supermarket) and store them in elements of the `Price` array as follows:

```
FOR Line IN 1..100 LOOP
    Put("Type the price of a line : ");
    Get(Price(Line));
END LOOP;
```

which assumes the program is interactive, and hence includes prompts to the user.

As a further example, suppose we wished to determine the number of prices which were more than £100. Assuming the `Integer` variable `NumExpensive` (to hold this number) we could write:

```
NumExpensive := 0;
FOR Line IN 1..100 LOOP
     IF Price(Line) > 100.0 THEN
          NumExpensive := NumExpensive + 1;
     END IF;
END LOOP;
```

10.2.2 Declaring array variables, and type declarations

Array variables are declared in a similar manner to ordinary variables, and we know that the declaration of a variable must include its type. How are we to do this? We will adopt the approach in this course of defining a new type name when we want to declare an array variable. Thus for the `Price` example we would write:

```
TYPE SalesList IS ARRAY (1..100) OF Float;
Price : SalesList;
```

The first of these steps is a **type declaration** which defines the identifier `SalesList` as a new type, while the second step declares `Price` as a variable of the type `SalesList`. These declarations are analogous to the declaration of an `Integer` variable `Number` using:

```
Number : Integer;
```

except that in the latter case we are using the fact that the type `Integer` is pre-defined in Ada, whereas in the former case we have first defined a new type called `SalesList`. Once `SalesList` has been defined in this way, we may use the type name `SalesList` in our program in just the same way we would use any of the pre-defined type names like `Integer`, `Float` or `Boolean`.

So what does the above type declaration tell the computer about the array variable `Price`? The declaration of `SalesList` states that it is an array type, it states the *type of the values* to be stored in the array elements, and what *range of subscript values* are used to index the array elements (and hence *how many values* are to be stored). These are all needed so that when the variable `Price` is declared the computer can allocate an appropriate amount of memory. Thus in the above example the computer would set up an array variable called `Price`, for which 100 storage locations would be reserved, each one capable of storing a `Float` value. The 100 storage locations being indexed with the values `1`, `2`, `3`, ..., `100`.

Note that the declaration of the type `SalesList` does not reserve any storage for the array, it is not like a variable declaration. Rather it just provides a mechanism for defining the structure of a new (array in this case) type. It is the declaration of the variable `Price` which requires storage to be allocated by the computer.

10.2.3 Array declaration in general (for now, more to come later)

Arrays can have any number of elements, numbered consecutively from some (`Integer`) starting value. The starting value need not be `1`, it could be any value. However the most commonly used starting values in practice are `0` and `1`.

Even though the subscript of an array is an `Integer` value, the values stored in the array elements can be of any type. Thus in the example of the `Price` array, `Integer` values (in the range `1` to `100`) are used when *subscripting* the array, but the elements are defined to hold `Float` values. Thus in general we will see array type definitions of the form:

```
TYPE Type_name IS ARRAY (First..Last) OF Element_type;
```

where *`Element_type`* (eg. `Integer`, `Float` or `Boolean`, etc.) specifies the type of the values to be stored in the elements. *`First`* and *`Last`* are normally `Integer` constants (ie. literal values or identifiers defined as constants). Notice that for a sensible array we must have *`First`* ≤ *`Last`*, and hence an array has (*`Last`*–*`First`*+1) elements. Although in general the array elements can be of any type, in any one array all the array elements must be of the same type. Thus it is not possible (for example) to have an array whose first element holds an `Integer` value, and whose second element holds a `Float` value, and whose third element holds a `Boolean` value, and so on.

Several array variables can be declared together if they are of the same array type. The following illustrates the declaration of the array variables `Balances`, `OldWeights`, `NewWeights`, `BitPattern` and `NumberOfWetDays`:

```
FirstAccountNo : CONSTANT Integer := 21000;
LastAccountNo  : CONSTANT Integer := 23000;
TYPE AccountList IS ARRAY (FirstAccountNo..LastAccountNo) OF Float;
Balances : AccountList;

NumObservations : CONSTANT Integer := 96;
TYPE Readings IS ARRAY (1..NumObservations) OF Float;
OldWeights, NewWeights : Readings;

TYPE Byte IS ARRAY (0..7) OF Boolean;
BitPattern : Byte;

TYPE BadWeather IS ARRAY (1980..1990) OF Integer;
NumberOfWetDays : BadWeather;
```

Array declarations and type declarations can be mixed up with other variable declarations as desired; the only restriction is that a type must be declared *before* it is used to declare variables. Arrays can also be declared as local variables within subprograms (i.e. procedures or functions) if this is required.

10.3 Programming example - an opinion poll

Problem:

The attitudes of 100 respondents when presented with some proposition are represented by integers in the range 1 (strongly disagree) to 7 (strongly agree). A set of 100 coded responses have been placed in a data file, and it is required to process this data to find the number of responses, and the percentage of the total they form, in each of the 7 categories.

First thoughts:

Open the input file in which the responses are stored and then read in and process the data values one-by-one. This needs a repetition which must repeat 100 times, so should use a `FOR` loop. On each repetition need to add 1 to the appropriate one of seven counts (one for each category). When the repetition is complete can then output final counts and percentages.

Using the methods of previous units, we would need seven separate variables for the counts (say `Count1`, `Count2`,..., `Count7`), and then to update the appropriate count we would have to write something like:

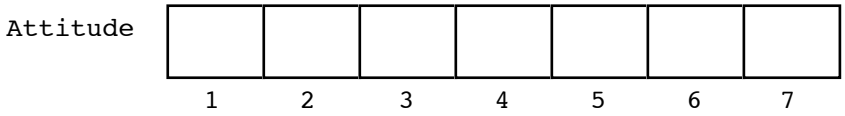
```
IF Response = 1 THEN
     Count1 := Count1 + 1;
ELSIF Response = 2 THEN
     Count2 := Count2 + 1;
.............
ELSIF Response = 6 THEN
     Count6 := Count6 + 1;
ELSE
     Count7 := Count7 + 1;
END IF;
```

This is very repetitious and tedious, and as we are dealing with a group of items which all need to be treated in the same way, it is much better to use an array with 7 elements, accumulating each count in one element of it. Bearing this in mind, we could arrive at:

Data structure(s)

```
MinCode : CONSTANT Integer := 1;
MaxCode : CONSTANT Integer := 7;
TYPE Frequency IS ARRAY (MinCode..MaxCode) OF Integer;
Attitude : Frequency;
```

The array `Attitude` contains 7 elements, each element being a count for the number of responses in each of the 7 categories.



In the top-down design of a program, it is always helpful to include specification of arrays which are important to the whole program together with the top level design (call this a new section 'Data structure(s)' for example). Notice that the limits of the array have been represented using appropriate constants. It is both more obvious what is going on and easier to introduce modifications (say change to a code range 1 to 9) if a program is developed using this technique.

Final program:

```
WITH Ada.Text_IO; USE Ada.Text_IO;
WITH CS_Int_IO;   USE CS_Int_IO;
WITH CS_Flt_IO;   USE CS_Flt_IO;
WITH CS_File_IO;  USE CS_File_IO;

PROCEDURE Opinion IS
     -- Program for Unit 10 of the ISP Ada course
     -- Written by L J Hazlewood, October 1993
     -- Modified by A Barnes, September 1994
     -- Modified by L J Hazlewood, October 1994

     NumResponses : CONSTANT Integer := 100;

     MinCode    : CONSTANT Integer := 1;
     MaxCode    : CONSTANT Integer := 7;

     TYPE Frequency IS ARRAY (MinCode..MaxCode) OF Integer;

     Attitude     : Frequency;
     Response     : Integer;

BEGIN

     OpenInput;

     -- Initialise all the counts to zero
     FOR Code IN MinCode .. MaxCode LOOP
          Attitude(Code) := 0;
     END LOOP;

     -- Input and process and the responses
     FOR Count IN 1 .. NumResponses LOOP
          Get(Response);
          Attitude(Response) := Attitude(Response) + 1;
     END LOOP;

     -- Output the percentages
     Put("Category  No. in category  Percent of total");
     New_Line;
     Put("========  ===============  ================");
     New_Line(2);

     FOR Code IN MinCode .. MaxCode LOOP
          Put(Item => Code, Width => 5);
          Put(Item => Attitude(Code), Width => 14);
          Put(Item => Float(Attitude(Code)*100)/Float(NumResponses),
              Fore => 14, Aft => 2);
          Put(Item => "%");
          New_Line;
     END LOOP;

     CloseInput;

END Opinion;
```