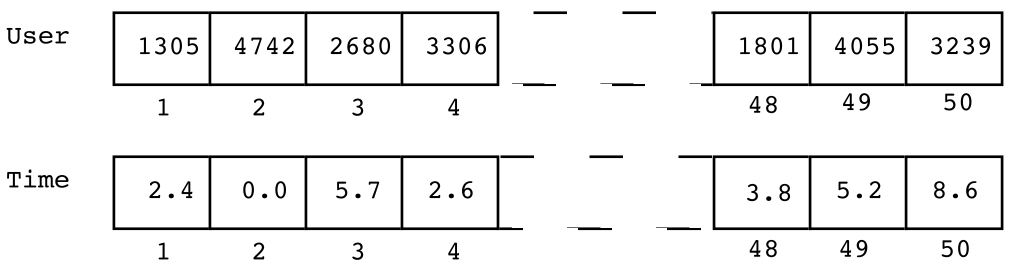
Introduction to Systematic Programming
Unit 11 - More on arrays and FOR loops

11.1 Parallel arrays

What happens when we want to store two or more collections of related data values? For example, suppose we wished to store the student identification numbers (a four-digit integer say) and the corresponding amount of computer CPU time used (as a real value in seconds) for a class of 50 students. We know that we should try to use arrays, since we need to store a large number of data values, but how can we arrange this given that one collection of values (the student identification numbers) are all of type `Integer`, and the other (CPU time used) are of type `Float`? The answer here is to use two arrays, where there is a one-to-one correspondence between the values stored in the array elements. For the above example we might use the arrays `User` and `Time` defined as follows:

```
NumStudents : CONSTANT Integer := 50;
TYPE StudentIDs IS ARRAY (1..NumStudents) OF Integer;
TYPE CPUTimes   IS ARRAY (1..NumStudents) OF Float;
User : StudentIDs;
Time : CPUTimes;
```

Assuming that appropriate values had been input into these arrays, we might have the storage arrangement:



These arrays are not intended to be used independently, but rather they are intended to be used in parallel, ie. the values in each element with the same subscript are intended to be used together. For example the above would indicate that the student with identification number `1305` had used `2.4` seconds of computer time, the student with identification number `4742` had not used any computer time, the student with identification number `2680` had used `5.7` seconds of computer time, and so on. Thus for any given array subscript `Index` (an `Integer` variable say), the array elements `User(Index)` and `Time(Index)` contain the identification number and the corresponding amount of CPU time used by one of the students. In Unit 16 we will see a somewhat better way of grouping related data values together, but for now parallel arrays provide quite a convenient way of organising related data values.

11.2 Input and output involving arrays

Since array elements behave like simple variables, it is possible to read values (of the correct type) into array elements, and to write out the values of array elements. The most common requirement is to read values into, or output values from all (occasionally some) of the elements of an array. Hence array input/output will usually occur in a `FOR` loop. For example, suppose our program contained the array declarations (taken from [10.2.3]):

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

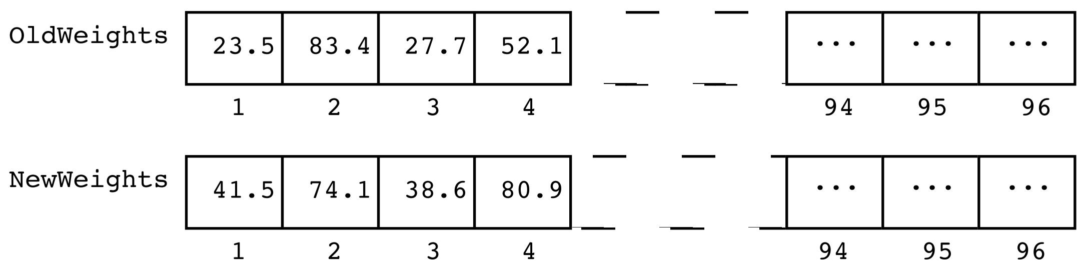
and further suppose that our program contained the following steps (to input data values into the elements of these arrays):

```
FOR Index IN 1 .. NumObservations LOOP
     Get(OldWeights(Index));
     Get(NewWeights(Index));
END LOOP;
```

This assumes that the values to be input into the `OldWeights` and `NewWeights` arrays are merged in the input data (ie. the first old weight value, then the first new weight value, the second old weight value, then the second new weight value, etc). Thus if the program were to be presented with input data values:

```
23.5    41.5
83.4    74.1
27.7    38.6
52.1    80.9
....    ....
```

This would produce the storage arrangement:



If the two collections of data are sequential, ie. in the form:

```
23.5  83.4  27.7  52.1  ....  41.5  74.1  38.6  80.9  ....
```

then two `FOR`'s will be required to produce the same storage arrangement, ie:

```
FOR Index IN 1 .. NumObservations LOOP
    Get(OldWeights(Index));
END LOOP;
FOR Index IN 1 .. NumObservations LOOP
    Get(NewWeights(Index));
END LOOP;
```

As another example, consider the following fragment which will output the values in the array `NewWeights`, five values to a line:

```
FOR Index IN 1 .. NumObservations LOOP
    Put(Item => NewWeights(Index), Width => 6);
    IF Index REM 5 = 0 THEN
        New_Line;
    END IF;
END LOOP;
```

11.3 Passing arrays as subprogram parameters

It is often necessary to pass arrays as procedure or function (i.e. subprogram) parameters. To illustrate what is involved, let's write a function which finds the largest value stored in an array of type `Readings`.

```
FUNCTION FindLargest (Weights : IN Readings) RETURN Float IS
    -- To find the largest of the values stored
    -- in the elements of the array Weights
    Largest : Float := Weights(1);  -- To store the largest so far
BEGIN
    FOR Index IN 2 .. NumObservations LOOP
        IF Weights(Index) > Largest THEN
            Largest := Weights(Index);
        END IF;
    END LOOP;
    RETURN Largest;   -- Return the largest value found
END FindLargest;
```

We see that to pass an array as a parameter of a subprogram we simply choose a formal parameter name (ie. `Weights` in this example) and describe its type (ie. `Readings` in this example) in the parameter specification of the subprogram definition. This is precisely what we would do if we were specifying any other type of parameter. Of course the array *type* `Readings` must have been declared before it is used in the parameter specification of function `FindLargest`. For this reason it is recommended that type declarations are placed together with constant declarations *before* any subprogram definitions so that the constants and types are available for use in the subprograms.

We could then call this function using:

```
MaxOld := FindLargest(Weights => OldWeights);
MaxNew := FindLargest(Weights => NewWeights);
```

to determine (in two `Float` variables `MaxOld` and `MaxNew`) the largest values in the two arrays `OldWeights` and `NewWeights`. It is important to note that these function calls are only valid because the actual parameter arrays `OldWeights` and `NewWeights` have the same type (`Readings`) as appears in the function definition. Whereas:

```
Max := FindLargest(Weights => Time);  -- !? Illegal in Ada
```

since the actual parameter `Time` is not of type `Readings` (it is of type `CPUTimes`).

If any of the elements of an array are to be altered by the actions of a procedure, then the parameter should be an `IN OUT` parameter (even if only some of the array elements are updated). For example to add 5 seconds of computer time to every element of an array of type `CPUTimes` we could write:

```
PROCEDURE Update (CPU : IN OUT CPUTimes) IS
    -- To add 5 seconds to every element of the array CPU
BEGIN
    FOR Index IN 1 .. NumStudents LOOP
        CPU(Index) := CPU(Index) + 5.0;
    END LOOP;
END Update;
```

Of course it is not always necessary to pass the whole of an array to a procedure as a parameter. Sometimes, if the procedure is just to use one particular element of the array, we need only pass that one element. For example the procedure:

```
PROCEDURE Special (Amount : IN OUT Float) IS
    -- To add 5.0 to this Amount
BEGIN
    Amount := Amount + 5.0;
END Special;
```

could be called using:

```
Special(Amount => Time(Student));
```

to add 5 seconds to the (type `Float`) element of the array `Time` with `Student` (assumed to be an `Integer` variable) as its subscript. In this case only one element of the array `Time` is being passed to the procedure as an actual parameter, so we only need to describe the type of that single element (ie. the type `Float`) in the formal parameter specification of the procedure.

11.4 Whole array assignment

Most array processing involves manipulating the individual elements. However, occasionally we want to make an assignment to all the elements of an array at once, perhaps by copying all the values stored in the elements of one array into the elements of another array. This is possible in Ada using an **array assignment** step, but *only if the two arrays have identical types*. Thus in the example:

```
NewWeights := OldWeights;
```

all the values in the elements of the array `OldWeights` are copied into the elements of the array `NewWeights`. The above step is thus equivalent to:

```
            FOR Index IN 1..NumObservations LOOP
               NewWeights(Index) := OldWeights(Index);
            END LOOP;
```

but is more concise. Note that the above array assignment is only possible because `NewWeights` and `OldWeights` are of the exactly the same type `Readings`.

11.5 Array aggregates

It is also possible in an array assignment, to have on the right hand side of the assignment a list of the values to be assigned to all the array elements. In this case the right hand side is known as an **array aggregate**. There are several possible ways of expressing an array aggregate, so we will start with the basic method which is to list all the array subscript values together with the values to be assigned to those elements. A simple example to illustrate this approach is as follows:

```
TYPE WorkingHours IS ARRAY (1..5) OF Float;
HoursWorked : WorkingHours;
.....................
HoursWorked := (1=>8.0, 2=>8.5, 3=>8.5, 4=>8.5, 5=>8.0);
```

which will assign the value `8.0` to the first and fifth elements of `HoursWorked`, and `8.5` to the remaining elements. When the same value is being assigned to more than one element, a shorthand notation can be used, which groups the together subscripts of those elements which are to be assigned the same value. Thus we could replace the last of the above steps by:

```
HoursWorked := (1|5 => 8.0, 2|3|4 => 8.5);
```

Notice that a vertical bar character is used to separate the subscripts which are grouped together. In addition, when the grouped subscripts are consecutive subscripts of the array, we may use a further shorthand notation, ie. we could re-write this step as:

```
HoursWorked := (1|5 => 8.0, 2..4 => 8.5);
```

where the notation `2..4` means all the subscripts from `2` to `4` inclusive. Hence if we wanted to assign the same value to all the elements of the array we could write:

```
HoursWorked := (1..5 => 8.25);
```

which would assign the value `8.25` to all five elements of the `HoursWorked` array. In [8.4] we saw how it was possible to declare a variable and to assign an initial value to the variable all in the same declaration. With the use of an array aggregate, we can combine the declaration of an array variable and the initialisation of its elements into a single declaration. Thus if we wished to declare a variable `YourHoursWorked`, and initialise all the elements of the array to the value `10.0`, we could write the combined declaration and initialisation as:

```
YourHoursWorked : WorkingHours := (1..5 => 10.0);
```

11.6 Using arrays to hold lists of varying size

One important use of arrays is to hold lists of values. For example, the array `User` in [11.1] above is used to hold a list of `NumStudents` student identification numbers. This works satisfactorily because we know that there are 50 (the value held in the constant `NumStudents`) values to be stored in the list. But how can we do this if the size of the list is not known at the time we write the program. For example suppose a program was to be presented with some input data which consisted of a list of student identification numbers followed by the value `-1` to act as a terminator, ie. some data of the form:

```
         1305  4742  2680  3306  2567  1895  -1
```

We cannot declare an array of exactly the size needed to hold this list of student identification numbers because the size of the list (and hence the size of the array) is not known when we write the program. Rather it will only be known when the program executes and inputs the data values provided.

Thus we cannot attempt to tackle this problem by writing something like:

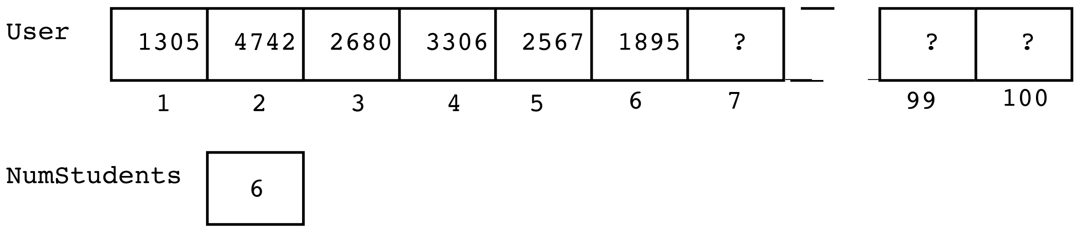
```
NumStudents : Integer;
TYPE StudentIDs IS ARRAY (1..NumStudents) OF Integer;
                                  -- !? Illegal in Ada
User  : StudentIDs;
Next  : Integer;
Index : Integer;
.............
Index := 0;
Get(Next);
WHILE Next /= -1 LOOP
     Index := Index + 1;
     User(Index) := Next;
     Get(Next);
END LOOP;
NumStudents := Index;
```

because `NumStudents` does not hold a value at the point in the program where the type `StudentIDs` is defined.

The only way we can use an array to hold such a list would be if we knew beforehand the maximum number of student identification numbers which were to be input. If this were the case we could then write:

```
MaxStudents : CONSTANT Integer := 100;
TYPE StudentIDs IS ARRAY (1..MaxStudents) OF Integer;
NumStudents : Integer;
User        : StudentIDs;
Next        : Integer;
.............
NumStudents := 0;
Get(Next);
WHILE Next /= -1 LOOP
     NumStudents := NumStudents + 1;
     User(NumStudents) := Next;
     Get(Next);
END LOOP;
```

which for the data values given above would produce the storage arrangement:



Notice that at the point in the program where the array type is defined, the size of the array is known (it is the value held in the constant `MaxStudents`). The above does of course mean that part of the array `User` is not actually used by the program, in fact only the array elements subscripted from `1` to `NumStudents` contain meaningful values, while those elements subscripted from `NumStudents+1` to `MaxStudents` remain undefined. There is nothing wrong with this (apart from the fact that there may be some waste of storage) so long as the subsequent steps of the program only make use of those elements of the `User` array which are subscripted from `1` to `NumStudents`.

11.7 Searching for particular values in an array

A common operation is to search an array for the presence (or otherwise) of a particular value stored somewhere in the array. For example, we might want to search the `User` array above for a particular student identification number held in the `Integer` variable `Wanted` (say). We could do this using (where `Found` and `Position` are `Boolean` and `Integer` variables respectively):

```
          Position := 1;
          Found := False;
          WHILE NOT Found AND Position <= NumStudents LOOP
               IF User(Position) = Wanted THEN
                    Found := True;
               ELSE
                    Position := Position + 1;
               END IF;
          END LOOP;
          -- After the search, if Found is True then Position holds the
          -- subscript of the array element holding the value in Wanted
```

For example, applying this fragment with the data values shown above in `User` and `NumStudents`, and with `Wanted` equal to `3306` would result in `Found` being set to `True` and `Position` to `4`.

Subsequently, for example, we may wish to output the amount of computer time used by that student using:

```
IF Found THEN
     Put("CPU time used was ");
     Put(Time(Position));
     Put(" Seconds");
ELSE
     Put("Searched for student number was not present");
END IF;
```

Notice that in the search method above we use a `WHILE` form of repetition rather than a form based upon the use of a `FOR` loop. This is because we want the repetition to stop whenever the searched for value in `Wanted` has been found. It would be possible to perform a search using:

```
          Found := False;
          FOR Index IN 1 .. NumStudents LOOP
               IF User(Index) = Wanted THEN
                    Found := True;
                    Position := Index;
               END IF;
          END LOOP;
```

However, notice here that after the searched for value has been found, ie. after `User(Index)` has been found to equal `Wanted`, the `FOR` loop continues to repeat until all `NumStudents` elements of the array `User` have been tested - even though these further tests are unnecessary.

Even the search method based on the use of a `WHILE` loop is potentially a very expensive process. Notice that we may have to search the array until we locate the searched for value, or search the entire list of values stored in the array until it is revealed that the searched for value is not present. Notice too that we have to make two tests on each repetition of the loop. One test to see if the current array element contains the value we are searching for, and another test to make sure that we have not 'run-off' the end of the list of values stored. In fact it is possible to improve this search process by including an extra (sometimes called a sentinel) element of the array, which is used solely to aid the searching process. So instead of the above declaration we would define the array with an extra, index 0, element as follows:

```
MaxStudents : CONSTANT Integer := 100;
TYPE StudentIDs IS ARRAY (0..MaxStudents) OF Integer;
NumStudents : Integer;
User        : StudentIDs;
```

Here elements numbered from `1` to `NumStudents` still contain the identification numbers of the students, and the element numbered `0` is just used as a temporary store to aid the searching process.

We then apply the search as follows:

i) Place the searched for value in the sentinel element.

ii) Start the search at the other end of the list stored in the array.

iii) Search the array elements, in a direction towards the sentinel element, until an element is encountered which contains the searched for value.

Note that we don't need to check whether or not the search has 'run-off' the end of the list of values, since even if the searched for value isn't present in the rest of the array, it will eventually be encountered in the sentinel element. We may thus re-write the above steps (and output the amount of computer time used by that student) using this more efficient search process as:

```
User(0) := Wanted;       -- Set up sentinel value
Position := NumStudents;
WHILE User(Position) /= Wanted LOOP
     Position := Position - 1;
END LOOP;
IF Position = 0 THEN
     Put("Searched for student number was not present");
ELSE
     Put("CPU time used was ");
     Put(Time(Position));
     Put(" Seconds");
END IF;
```

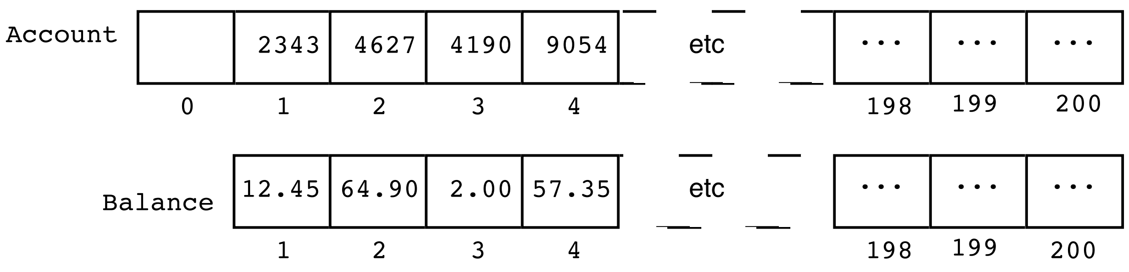
11.10 Programming example - a banking system

Two data files `oldaccts.dat` and `transactions.dat` have been prepared. The first file contains account information of the current account holders of a bank, terminated by a `-1`. It may be assumed that the bank does not have more than 200 current account holders. Each item of account information comprises an account number (a four digit integer) followed by an amount representing the present balance of the account (a real value in pounds and pence). The second file contains transactions which have taken place during a day's trading, again terminated with a `-1`. Each transaction consists of an account number followed by an amount by which the account should be credited (if the amount is positive) or debited (if the amount is negative). There may of course be several transactions for any given account. It is required to process the data files and output a new file `newaccts.txt` containing account information for all the account holders, but with the balances of the accounts updated with the transactions. Typical data files would thus be:

```
oldaccts.dat          transactions

2343   12.45          4190   200.00
4627   64.90          9054   -50.00
4190    2.00          4190    50.00
9054   57.35          4190  -150.00
2479   36.26          ....    .....
....   .....          4190   -60.00
3725   47.28          -1
-1
```

Thus we need two (parallel) arrays to store the account numbers and balances. These arrays can be used to hold the initial account details and the updated details. Note that we will also need to search the `Account` array for particular account numbers, so a sentinel element has been added. For example after input of the `oldaccts.dat` file the arrays would contain :



```
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 Bank IS

    -- Program for Unit 11 of the ISP Ada course.
    -- Written by L J Hazlewood, October 1994.
    -- Updated by A Barnes, August 1996.

    DataTerminator : CONSTANT Integer := -1;
    MaxAccounts    : CONSTANT Integer := 200;

    TYPE AccountList IS ARRAY (0..MaxAccounts) OF Integer;
        -- The zero element is just used for searching.
    TYPE BalanceList IS ARRAY (1..MaxAccounts) OF Float;

-- *********************************************************************

    PROCEDURE InputAccountDetails (Account : OUT AccountList;
                                   Balance : OUT BalanceList;
                                   NumAccounts : OUT Integer) IS
        -- To input the initial account details, ie. the account
        -- number and balance of each account holder. The data
        -- values being terminated with DataTerminator.
        Number : Integer;  -- An account number.
        Amount : Float;    -- An amount of money (in pounds/pence).
    BEGIN
        OpenInput("oldaccts.dat");
        NumAccounts := 0;   -- Initialise the number of accounts.

        Get(Number);
        WHILE Number /= DataTerminator LOOP
            Get(Amount);
            NumAccounts := NumAccounts + 1;
            Account(NumAccounts) := Number;
            Balance(NumAccounts) := Amount;
            Get(Number);
        END LOOP;

        CloseInput;
    END InputAccountDetails;

-- *********************************************************************

    PROCEDURE OutputAccountDetails (Account : IN AccountList;
                                    Balance : IN BalanceList;
                                    NumAccounts : IN Integer) IS
        -- To output the account details,  ie. the account
        -- number and balance of each account holder.
    BEGIN
        OpenOutput("newaccts.txt");

        FOR Index IN 1 .. NumAccounts LOOP
            Put(Item => Account(Index), Width => 6);
            Put(Item => Balance(Index), Fore => 5, Aft => 2);
            New_Line;
        END LOOP;

        CloseOutput;
    END OutputAccountDetails;

-- *********************************************************************
```

```
    PROCEDURE ProcessTransactions (Account : IN OUT AccountList;
                                   Balance : IN OUT BalanceList;
                                   NumAccounts : IN Integer) IS
         -- To process the transactions and update the account details,
         -- ie. input a series of account numbers and corresponding
         -- amounts, and update the details of the account holders.
         Number : Integer;    -- An account number.
         Amount : Float;      -- An amount of money (in pounds/pence).
         Position : Integer;  -- Used to record the outcome of a search.
    BEGIN
         OpenInput("transactions.dat");

         Get(Number);
         WHILE Number /= DataTerminator LOOP
              Get(Amount);
              -- Search the account details for this account number
              Account(0) := Number;       -- Set up sentinel value
              Position := NumAccounts;
              WHILE Account(Position) /= Number LOOP
                   Position := Position - 1;
              END LOOP;
               -- Position holds the array index of where this
               -- account number is stored (unless Position = 0).
              IF Position /= 0 THEN
                   Balance(Position) := Balance(Position) + Amount;
              ELSE
                   Put("There must be an error in the data");
              END IF;
              Get(Number);
         END LOOP;

         CloseInput;
    END ProcessTransactions;

-- ************************************************************************

    UserAccount : AccountList;  -- To hold the numbers and corresponding
    UserBalance : BalanceList;  -- balances of the accounts.
    NumberOfAccounts : Integer; -- The number of account holders.

BEGIN  -- Main program.

    -- Input the initial account details.
    InputAccountDetails (Account => UserAccount,
                         Balance => UserBalance,
                         NumAccounts => NumberOfAccounts);

    -- Process the transactions and update the account details.
    ProcessTransactions (Account => UserAccount,
                         Balance => UserBalance,
                         NumAccounts => NumberOfAccounts);

    -- Output the updated account details.
    OutputAccountDetails (Account => UserAccount,
                          Balance => UserBalance,
                          NumAccounts => NumberOfAccounts);

END Bank;
```