Introduction to Systematic Programming
Unit 6 - More on Procedures

6.1 Introduction

In Unit 5 we discussed how a name could be given to a fragment of program by means of a procedure definition placed in the declaration section of the main program. Then every time it is required to execute that program fragment we could simply place a procedure call step in our program at the required spot (instead of writing out the whole fragment again). Next procedures with one or more `IN` (or 'copy-in') parameters were discussed. Such procedures could be made to perform a varying task by passing values into the procedure. For example one might define (in the declaration section of the main program) a procedure `PrintMax` to determine and output the larger of two `Integer` values as follows:

```
PROCEDURE PrintMax (First, Second : IN Integer) IS
     Max : Integer;
BEGIN
     IF First > Second THEN
          Max := First;
     ELSE
          Max := Second;
     END IF;
     Put(Item => Max);
END PrintMax;
```

Then in the main program one could print the larger of the two values of two `Integer` variables `A` and `B` (say) simply by inserting the procedure call step:

```
     PrintMax(First => A, Second => B);
```

Recall that the values of the actual parameters supplied in the call step are copied into the corresponding formal parameters as the procedure is entered (ie. the value of `A` is copied into `First` and the value of `B` into `Second`) and then the procedure body is executed to determine and output the value of `Max`.

6.2 Getting values back from a procedure

The procedure `PrintMax` is fine but suppose that, instead of printing the value of `Max`, we wanted to use its value back in the main program in subsequent program steps. How can this be achieved? The procedure parameters described so far serve only one purpose - they allow values to be passed *into* procedures, which can then utilise those values as required. Clearly what is needed is a mechanism for passing values back from the procedure to the calling program. In Ada this is provided by defining a procedure to have one or more formal **`OUT`** (or **'copy-out'**) **parameters**. `OUT` parameters are defined in the procedure heading in the same way as `IN` parameters except that the keyword `IN` is replaced by the keyword `OUT` in the parameter specification. For example we might define a procedure `FindMax` with two `IN` parameters and one `OUT` parameter (all of type `Integer`) as follows:

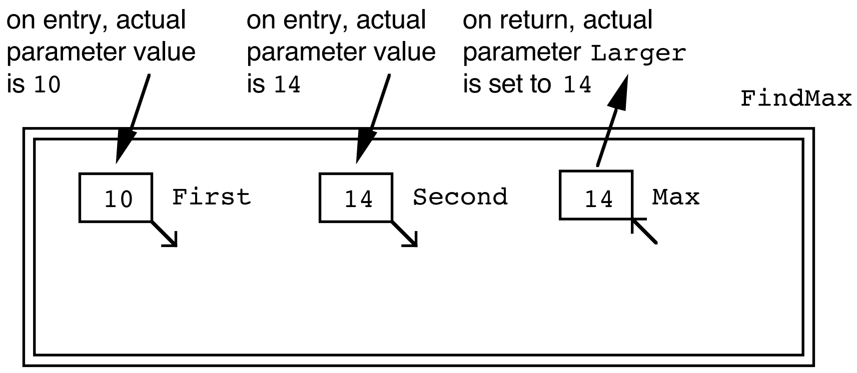
```
PROCEDURE FindMax (First, Second : IN Integer;
                   Max : OUT Integer) IS
BEGIN
     IF First > Second THEN
          Max := First;
     ELSE
          Max := Second;
     END IF;
END FindMax;
```

Note that the local variable `Max` (in the earlier version) has been replaced by a formal `OUT` parameter.

Then in a program we could determine the larger value of two `Integer` variables `A` and `B` (which for example contain the values `10` and `14` respectively) and assign it to another `Integer` variable `Larger` simply by using the procedure call step:

```
     FindMax(First => A, Second => B, Max => Larger);
```

On entry to the procedure storage locations are allocated for the three formal parameters `First`, `Second` and `Max`. The values of the actual `IN` parameters `A` and `B` supplied in the call step are copied into the corresponding formal parameters `First` and `Second` in the normal way. Next the body of `FindMax` is executed and so assigns the required value to `Max` and the procedure then returns to the calling program. As the procedure returns, the value of the formal `OUT` parameter `Max` is copied to the corresponding actual parameter, namely `Larger`. The value of `Larger` can then be used by the calling program in the normal way.


on entry, actual parameter value is 10
on entry, actual parameter value is 14
on return, actual parameter Larger is set to 14
FindMax
10
First
14
Second
14
Max

Notes:

i) No copying of the value of the actual parameter `Larger` into the formal `OUT` parameter `Max` takes place on entry to the procedure. Thus like any newly established storage location, `Max` initially contains an undefined or 'junk' value. However, once a value has been assigned to `Max` it can then be used within the procedure just as if it were a normal (but local) variable of the procedure.

ii) When a procedure is called the value of the actual parameter corresponding to a formal `OUT` parameter is irrelevant, and it is perfectly permissible for an actual parameter (such as `Larger` above) to be an uninitialised variable containing a 'junk' value.

iii) Of course, the values of the formal `IN` parameters are *not copied out* to the corresponding actual parameters as the procedure returns (this only happens for `OUT` parameters).

iv) As the procedure returns the storage locations allocated for the three formal parameters `First`, `Second` and `Max` are all freed and so they cease to exist after the procedure returns.

As a second example consider the following procedure which converts a time all in seconds to one in hours, minutes and seconds. As this needs to pass three values back to the calling program three `OUT` parameters are needed:

```
PROCEDURE Convert (Time : IN Integer;
                   Hrs, Mins, Secs : OUT Integer) IS
     FullMins : Integer;
BEGIN
     -- Note that Integer division is being used here so that
     -- the 'fractional' part of any quotient is discarded
     FullMins := Time/60;    -- Total number of full minutes in Time,
     Secs := Time REM 60;    -- and the number of seconds which remain.
     Hrs  := FullMins/60;       -- The number of full hours in Time
     Mins := FullMins REM 60;  -- and the number of minutes.
END Convert;
```

6.3 `IN OUT` parameters

Sometimes it is necessary to have a procedure change a value supplied to it as an actual parameter and to pass back the modified value to the calling procedure. To achieve this in Ada, it is necessary to use another sort of formal parameter which combines the features of `IN` and `OUT` parameters. Such parameters are called **`IN OUT` parameters** (or **'copy-in'/'copy-out' parameters**) and they are defined by a formal parameter specification of the form:

```
Formal_parameter_name : IN OUT Type_name
```

As a very simple example consider the following procedure:

```
PROCEDURE Inc (X : IN OUT Integer) IS
BEGIN
    X := X + 1;
END Inc;
```

Once this procedure has been defined it can be used to increment the value of a variable by one. For example the call step:

```
Inc(X => NumOfDivisors);
```

would cause the value of the `Integer` variable `NumOfDivisors` to be increased by one. As the procedure is called a storage location is allocated for the formal parameter `X` and the value of the actual parameter `NumOfDivisors` is copied into `X` (just as if `X` were an `IN` parameter), then the body of the procedure is executed which causes `X` to be incremented by `1`. The procedure then returns and as it does so, the updated value of `X` is copied back into the actual parameter `NumOfDivisors` (just as if `X` were an `OUT` parameter) and the storage location allocated to hold the formal parameter `X` is then freed.

Notice that although formal `IN OUT` parameters of a procedure behave like `IN` parameters on entry to the procedure, they do not have the same restrictions as those imposed on 'copy-in' parameters. In fact they may both have their contents retrieved and have their value assigned to, and thus may be used in expressions and as the target of assignments. For example in procedure `Inc`, `X` is used in the expression `X+1` (where its value is retrieved) and also on the left-hand side of the assignment (causing a new value to be assigned into `X`). Such formal parameters therefore behave like normal (but local) variables inside the procedure.

As a second slightly less trivial example consider the following:

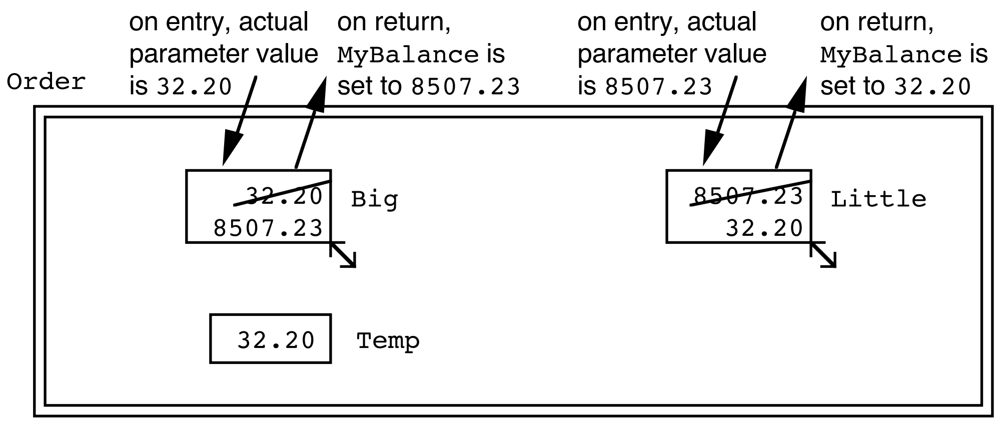
```
PROCEDURE Order (Big, Little : IN OUT Float) IS
   -- Orders a pair of Float values so that, on exit,
   -- the larger is in Big and the smaller in Little

    Temp : Float;  -- Temporary variable for the swap
BEGIN
    IF Big < Little THEN
        -- Swap the values
        Temp := Big;
        Big := Little;
        Little := Temp;
    END IF;
END Order;
```

Once this procedure has been defined it might be used as follows. Assume variables `MyBalance` and `YourBalance` of type `Float` have been declared and that certain values `32.20` and `8507.23` (say) have been stored in these variables. Now consider the procedure call:

```
Order(Big => MyBalance, Little => YourBalance);
```

The sequence of events during execution of this call are:

i)	Procedure entered	The value of `MyBalance` (`32.20`) is stored in `Big` and the value of `YourBalance` (`8507.23`) is stored in `Little` since `Big => MyBalance, Little => YourBalance`
ii)	`Temp := Big;`	`32.20` is stored in `Temp`
iii)	`Big := Little;`	The value in `Big` is overwritten with `8507.23`
iv)	`Little := Temp;`	The old value in `Little` is overwritten with `32.20`
v)	Procedure returns	Current value of `Big` (`8507.23`) is stored in `MyBalance` and current value of `Little` (`32.20`) is stored in `YourBalance` since `Big => MyBalance, Little => YourBalance` Thank you for your donation!


Order
on entry, actual parameter value is 32.20
on return, MyBalance is set to 8507.23
on entry, actual parameter value is 8507.23
on return, MyBalance is set to 32.20
~~32.20~~
8507.23
Big
~~8507.23~~
32.20
Little
32.20
Temp

Note that the swap cannot be written as:

```
Big := Little; Little := Big;    -- illogical in Ada
```

The use of a temporary variable is essential in the swap. Think about it!

The procedure `Order` could also be used to define a procedure to sort three values into descending order as follows:

```
PROCEDURE Sort (First, Second, Third : IN OUT Float) IS
   -- Sorts three Float values so that, on exit, they are
   -- ordered as:   First >= Second >= Third
BEGIN
     Order(Big => First, Little => Second);  -- order 1st & 2nd
     Order(Big => First, Little => Third);   -- order 1st & 3rd
     Order(Big => Second, Little => Third);  -- order 2nd and 3rd
END Sort;
```

It is a good exercise to hand trace this procedure for a variety of parameter values.

Note that the procedure `Sort` calls the procedure `Order` defined above. In order for this to work the procedures `Order` and `Sort` should both be defined in the declaration section of the main program with the definition of `Order` appearing *before* the definition of `Sort`. This is necessary since the Ada compiler needs to 'know' about the procedure `Order` (for example that it requires two `IN OUT` parameters of type `Float`) before it can compile the procedure `Sort` since (among other things) the Ada compiler checks that `Order` is being called with the correct number parameters, each of the correct type.

6.4 Formal parameter specification rules

The kind of a procedure parameter (`IN`, `OUT` or `IN OUT`) is referred to as the **mode** of the parameter. The formal parameter specifications for a procedure specify the name, mode and type of each parameter. Each such specification part is of the form:

```
        List_of_identifiers_separated_by_commas : Mode Type_name
```

where the *Mode* is `IN`, `OUT` or `IN OUT`. Here are some examples of FP specifications:

a) `Margin, Length : IN Integer`

b) `Item : IN Float; Fore, Aft : IN Integer`

c) `Total : IN Integer; Count : IN Integer; Average : OUT Float`

d)
```
Time      : IN Float;
Hrs, Mins : OUT Integer;
Sec       : OUT Float
```

Note that it is perfectly permissible to carry on a specification over more than one line if this improves the readability of the procedure heading (as was the case in the last of the examples above).

In theory there is no limit to the number of parameters of a procedure nor to the mix of modes and types of these parameters. However, in practice procedures rarely have more than about six or so parameters (and usually somewhat fewer than that). The use of procedures with large numbers of parameters is usually a sign that the program design has not be properly thought out, or that the procedurisation of the algorithm (ie. its subdivision into procedures) has been poorly done.

6.5 A restriction on the use of `OUT` and `IN OUT` parameters

In [5.5] we saw that for parameters of mode `IN` the actual parameter could be a literal constant, a variable or an expression. The only restriction was that the AP yielded a value whose type matched the type of the corresponding FP as specified in the parameter specification. This allows the values of the APs to be assigned to the corresponding FPs when the procedure is entered. However, for parameters of mode `OUT` or `IN OUT`, the actual parameter must refer to a storage location in the calling program. For the time being this means that the actual parameter *must* be a variable - it is not allowed to be a literal constant nor an expression. To see why this restriction is necessary we must consider what happens when the procedure returns. Recall that the value of a FP of mode `OUT` or `IN OUT` is *copied* to the corresponding AP as the procedure returns, therefore there must be a storage location associated with the AP to hold this value. It is clearly nonsense to attempt to copy a value to a literal constant (such as `2`) or to an expression (such as `X+Y`); what could these possibly mean?

Also it is still necessary for the types of the AP and its corresponding FP to match so that it is permissible to copy-out the value of the FP to the corresponding AP as the procedure returns (and in the case of `IN OUT` parameters to copy from AP to FP as the procedure is entered).

6.6 The main program and I/O procedures

In Unit 3 we saw that the heading of a main program takes the form:

```
        PROCEDURE Program_name IS
```

which is exactly the same form as the heading of a procedure with no parameters. In fact the main program can be regarded as a procedure which is called by the operating system of the computer. When the main program terminates it 'returns' to the operating system just as a normal procedure returns to the main program which called it.

In Units 3 and 5 the I/O procedures `Get` and `Put` in the library packages `Ada.Text_IO`, `CS_Int_IO` and `CS_Flt_IO` were discussed. You may have noticed that input and output steps take the same form as procedure calls. This is not surprising since `Get` and `Put` are, in fact, ordinary Ada procedures - but they are defined in their respective library packages and imported into the main program with the context clauses:

```
         WITH Package_name; USE Package_name;
```

instead of being defined in the declaration section of the main program. It is instructive to consider what the parameters of these I/O procedures might be.

The procedure `Get` (from the package `CS_Int_IO`) is defined to have a formal parameter `Item` of mode `OUT` and type `Integer`. Assuming the example call step:

```
        Get(Item => IntVar);
```

execution of the procedure body 'gets' an integer value from an input device (such as the keyboard) and stores it temporarily in the FP `Item`. As the procedure returns, it passes this value back to the calling program by copying the value of `Item` to the AP `IntVar` (assumed to be an `Integer` variable of the calling program). On the other hand, the procedure `Put` (from the package `CS_Int_IO`) is defined to have a formal parameter `Item` of mode `IN` and type `Integer`. Assuming the call step:

```
        Put(Item => IntVal);
```

the integer value specified as the AP (ie. the value of the `Integer` variable `IntVal`) is copied into the FP `Item` as the procedure is entered. Execution of the procedure body then outputs the value of the FP to an output device (such as a VDU screen) and returns to the calling program.

Similarly, `CS_Flt_IO.Get` is defined (in the package `CS_Flt_IO`) to have a formal parameter `Item` of mode `OUT` and type `Float` whereas the corresponding procedure `Put` has a formal parameter `Item` of mode `IN` and type `Float`.

6.7 Clarifying program structure

The problem of [5.10] provides an illustration of the use of procedures to clarify program structure. A slightly extended outline of an algorithm of the problem would be:

```
    Output prompt for user to input a number
    Get(Item => UsersNumber)
    WHILE UsersNumber > 1 LOOP
        Produce Factors of UsersNumber
        Output prompt for user to input a number
        Get(Item => UsersNumber)
    END LOOP
```

Since the prompt and read step are duplicated we might choose to write these as the procedure:

```
PROCEDURE GetNextInput (Number: OUT Integer) IS
BEGIN
    DrawLine;     -- output a line of stars
    New_Line;     -- blank-line for spacing
    Put(Item => "Type a number greater than 1 (0 or 1 to quit) :");
    Get(Item => Number);
END GetNextInput;
```

Note that the FP in this case (`Number`) must have mode `OUT`, since the actions of the procedure are to input an `Integer` value (using a call to `Get`), and pass it back to the main program (where the procedure was called) so that the value may be used in further calculations. Note also that the procedure `GetNextInput` in turn calls other procedures both user-defined (`DrawLine`) and library I/O procedures (`Get`, `Put` and `New_Line`).

As a consequence of this procedurisation, the body of the main program would then read:

```
GetNextInput(Number => UsersNumber);
WHILE UsersNumber > 1 LOOP
    ProduceFactors(Number => UsersNumber);
    GetNextInput(Number => UsersNumber);
END LOOP;
```

Note how the introduction of the two procedures (with meaningful names) clarifies the structure of the main program. The program appears in full on page 8.

6.8 Summary

At this point it is useful to review some of the material covered in Units 5 and 6 as the concepts introduced are some of the most important in the whole course. Essentially writing a procedure involves giving a name to a program fragment and so that, subsequently the program fragment can be executed simply by using the procedure name (plus parameters if appropriate) as a program step. Variables used within the procedure should be declared within the procedure as local variables. This makes the procedure self-contained and so it can be understood without reference to the rest of the program.

The advantages of using procedures and local variables are:

i) They avoid duplication of program steps by enabling the fragment of program to be called from many different places.

ii) They promote re-use of program fragments in other programs, either by building up library packages of useful procedures or even by simply 'cutting' and 'pasting' a procedure from one program to another using an editor.

iii) They can be used to clarify the structure of a program. Not only can this simplify the writing the program but it also makes it easier for other programmers to understand (and perhaps to correct or extend) the program.

iv) A large program may contain hundreds of variables and procedures. If most of the variables are declared as local variables (of the procedure in which they are used) rather than as global variables then the overall memory requirements of the program are reduced. Storage is allocated for local variables only for the duration of a procedure's execution and so once these storage locations have been released the locations can be used for the local variables of other procedures. Storage for global variables, on the other hand, is allocated for the duration of the whole program.

The use of procedures with parameters has the additional advantages that:

v) Procedures are more versatile and may be made to perform different tasks depending on the actual parameter values supplied in the procedure call.

vi) The parameters form a carefully controlled interface between the procedure and the calling program. By the use of `OUT` or `IN OUT` parameters a procedure may alter the values of variables in the calling program - but only in a clearly specified manner. By inspecting only the FP specification and the procedure call step it is clear exactly what information is being passed into the procedure, what information is being passed back out to the calling program and which variables in the calling program have been affected by the procedure call. Detailed inspection of the steps in the procedure body is not necessary.

The execution life-time of local variables and formal parameters of a procedure is for the duration of execution of the body of that procedure. Outside this period, ie. before the procedure is called, or after the procedure has returned, the local variables and formal parameters do not exist.

`IN` parameters are used to pass values into a procedure from the calling program. The value of the actual parameter is copied into the corresponding formal parameter as the procedure is called, thereby initialising the FP. The value of a formal `IN` parameter is constant (or 'retrieve-access-only'); its value cannot be changed by the procedure. The actual parameter may be a constant value, a variable or an expression; the only restriction is that it produces a value of the same type as the corresponding formal parameter.

`OUT` parameters are used to pass values back from a procedure to the calling program. The value of the formal parameter is copied into the corresponding actual parameter as the procedure returns, thereby making the value available for use in the calling program. A formal `OUT` parameter may be used like a local variable within the procedure, and have its value retrieved or be assigned a value. The actual parameter must be a *variable* of the same type as the formal parameter.

`IN OUT` parameters are used when it is required to pass a value into a procedure from the calling program, modify it in some way and to pass the new value back to the calling program. The value of the actual parameter is copied into the corresponding formal parameter as the procedure is called, thereby initialising the FP. When the procedure returns the value of the FP is copied into the corresponding AP and so is available for use by the calling program. A formal `IN OUT` parameter may be used like a local variable within the procedure, and have its value retrieved or be assigned a value. The actual parameter must be a *variable* of the same type as the formal parameter.

```ada
WITH CS_Int_IO;   USE CS_Int_IO;    -- For Get and Put for Integer I/O
WITH Ada.Text_IO; USE Ada.Text_IO;  -- For Put for text, and New_Line

PROCEDURE Factors IS

   PROCEDURE DrawLine IS   -- To output a line of stars
   BEGIN
      New_Line;
      Put(Item => "**********************************");
      New_Line;
   END DrawLine;

   PROCEDURE GetNextInput (Number : OUT Integer) IS
   BEGIN
      DrawLine;      -- Output a line of stars
      New_Line;      -- Blank line for spacing
      Put(Item => "Type in a number greater than 1 (0 or 1 to quit): ");
      Get(Item => Number);
   END GetNextInput;

   PROCEDURE ProduceFactors (Number : IN Integer) IS
      -- Produce and output the factors
      TrialDivisor  : Integer;  -- To hold each possible trial divisor
      NumOfDivisors : Integer;  -- To count the number of divisors
      -- Note that the two variables TrialDivisor and NumOfDivisors
      -- are only used in the ProduceFactors procedure, and hence
      -- are declared as local variables of the procedure
   BEGIN
      NumOfDivisors := 0; TrialDivisor := 2;
      New_Line;
      Put(Item => "Factorisations of ");
      Put(Item => Number);
      Put(Item => " are:"); New_Line;

      WHILE TrialDivisor <= Number / 2 LOOP
         IF Number REM TrialDivisor = 0 THEN
            -- ie. if an exact divisor is found
            Put(Item => TrialDivisor, Width => 8);
            Put(Item => Number / TrialDivisor, Width => 10);
            New_Line;
            NumOfDivisors := NumOfDivisors + 1;
         END IF;
         TrialDivisor := TrialDivisor + 1;
      END LOOP;

      IF NumOfDivisors > 0 THEN
         Put(Item => "There were ");
         Put(Item => NumOfDivisors);
         Put(Item => " divisors found.");
      ELSE
         Put(Item => "No divisors found - ");
         Put(Item => Number);
         Put(Item => " is prime.");
      END IF;
      New_Line;
   END ProduceFactors;

   UsersNumber : Integer;  -- To hold each number specified by the user

BEGIN   -- Main Program
   GetNextInput(Number => UsersNumber);
   WHILE UsersNumber > 1 LOOP
      ProduceFactors(Number => UsersNumber);
      GetNextInput(Number => UsersNumber);
   END LOOP;
END Factors;
```