Introduction to Systematic Programming
Unit 5 - Procedures - Giving a Process a Name

5.1 Introduction - motivation for using procedures

We might decide that we could improve the appearance of our `Factors` example's output [4.6] by splitting up the various sections using lines of stars, thus:

```
*********************************
Type in a number ........
Factorisations of .......
.........................
.........................
There were ..............
*********************************
Type another number .....
Factorisations of .......
.........................
.........................
There were ..............
*********************************
Type another number .....
```

A suitable Ada program fragment to produce one such line (surrounded by blank lines) is:

```
New_Line;
Put(Item => "*********************************");        A
New_Line;
```

We could of course include this program fragment several times, once for each line of stars required in the output; but a much more convenient method is to write the fragment down *once only* and *give it a name*. This will allow us to get the fragment obeyed whenever it is needed, simply by quoting the name that we have given to it. Named program fragments of this form are known as a **procedures**.

To name a program fragment, we have to introduce a **procedure definition** into the program; a suitable definition for this example is:

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

Now whenever we want our program to output a line of stars, instead of writing out the entire fragment **A** every time, we can write a single program step. This step consists solely of the name given to the procedure, thus:

```
DrawLine;
```

and is know as a **procedure call**.

5.2 Rules for defining and using procedures

A procedure definition must include a **heading** and a **body**. In its simplest form, a procedure heading consists of the keyword `PROCEDURE`, followed by an identifier which will be the name used to refer to the procedure. The heading must be separated from whatever follows by the keyword `IS`. A body of a procedure consists of one or more program steps (as always, terminated by semi-colons) enclosed between `BEGIN` and `END` followed by the name of the procedure (which must be the same as in the procedure heading) and a semi-colon. Definitions of procedures are written in the declaration part of a program. In Ada 95 procedure definitions and variable declarations can appear in any order in the declaration part of a program. However for improved program clarity we recommend

that procedure definitions precede all variable declarations; the variable declarations then appear near to where they are used, that is in the body of the main program.

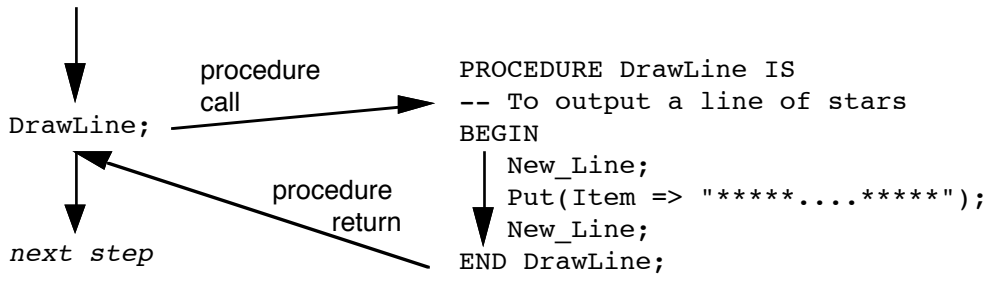
Once a procedure has been defined, its name can be quoted (ie. a procedure call step can be invoked) at any point in the program where any other type of step could be used. Its effect is to cause the body of the called procedure to be obeyed. When execution of the procedure body is complete, the procedure 'returns' and execution continues with whatever follows the procedure call step (*not whatever follows the procedure's definition*).

5.3 Use of `DrawLine` in `Factors`

The following outline shows the positions of the definition and various calls of `DrawLine`.

```
WITH CS_Int_IO; ........;                          Corresponding output

PROCEDURE Factors IS

    PROCEDURE DrawLine IS                  No output from the definition itself. The
        -- To output a line of stars       definition of DrawLine does not do
    BEGIN                                  anything until it is 'woken up' by a call step.
        New_Line;
        Put(Item => "**********************************");
    New_Line;
    END DrawLine;

    -- Variable declarations follow
    Number : ...........;
    ..................;

BEGIN
    DrawLine;                                      *****************......
    Put(Item => "Type ..............;              Type a number.........
    Get(Item => Number);
    WHILE Number >1 LOOP
        NumOfDivisors := 0;
        TrialDivisor := 2;
        ...................;
        WHILE TrialDivisor <= .... LOOP
            .................;                     .................
            .................;                     Factorisations output
            .................;                     .................
        END LOOP;
        IF NumOfDivisors .... THEN
            Put(Item => "There were ...;           There were.............
            ...........;
        ELSE
             ...........;
        END IF;
        ............;
        DrawLine;                                  ******************.......
        Put(Item => "Type ............;           Type another number......
        Get(Item => Number);
    END LOOP;
END Factors;
```

The effect of each `DrawLine` call may be visualised thus:



Note that the flow of execution through the program is shown by: ⟶

5.4 Declaring variables inside procedures - local variables

Suppose we want to leave a gap of 10 lines in our output; a suitable program fragment is:

```
LineNo := 1;
WHILE LineNo <= 10 LOOP
     New_Line;
     LineNo := LineNo + 1;
END LOOP;
```

Obviously we could make this into a procedure by wrapping it up inside `BEGIN...END` and giving it a procedure heading, but where should `LineNo` be declared? Logically it only has any significance within the procedure that we are constructing, so it would be clearer to declare it in a way which shows this. We can achieve this by declaring `LineNo` in a complete new declaration between our procedure's heading and its `BEGIN...END` body thus:

```
PROCEDURE LeaveGap IS
   -- To output 10 blank lines
   LineNo : Integer;   -- Declaration of variable LineNo
BEGIN
   LineNo := 1;
   WHILE LineNo <= 10 LOOP
      New_Line;
      LineNo := LineNo + 1;
   END LOOP;
END LeaveGap;
```

A variable declared within a procedure is said to be a **local variable** (of that procedure). The declaration part of a procedure can take exactly the same range of forms as the (main) program part, but whatever variables it declares are local to the procedure. Local variables (as their name suggests) can only be used *within* their own procedure. Thus the use of local variables makes the purpose of the variables (and the procedures that use them) more obvious and has the effect of making the procedure 'self-contained' (more on this later).

It is important to understand the status of local variables when a procedure is obeyed. When a procedure is called, on entry to the procedure but before the body of the procedure is obeyed, its local variables are established by the computer creating the necessary storage (with undefined or junk contents) to hold the value of each local variable. They then exist, and are available for use as the `BEGIN..END` body is obeyed. When the procedure finishes its execution the computer frees the storage allocated for their use so that the local variables no longer exist after the procedure returns. Thus the **execution life-time** of a local variable, ie. the period of its existence during which it may be used, is solely for the duration of the execution of the procedure within which it is declared. Also, local variables *do not retain their values* from one call of a procedure to the next; they are re-established afresh at each separate call of the procedure.

On the other hand, a variable which is declared in the declaration section of the main program is said to be a **global variable** (of that program). Unlike local variables, global variables have an execution life-time of the entire duration of the program - from the point in the program where they are declared to the end of execution of the program.

5.5 More general procedures

The procedures `DrawLine` and `LeaveGap` do exactly the same things each time they are called, but it is possible to write procedures which are more flexible, ie. which can be made to perform slightly different actions each time they are called.

If a procedure is to perform a varying task, such as (for example) drawing a line of stars of variable length, it must be given the relevant information, which in this case is the number of characters to be used to make up the line of stars. In Ada, the process of handing over a value (such as `20` in the example below) to a procedure is incorporated into the call step itself, thus we would write the step:

```
DrawLine(Length => 20);
```

Information passed to a procedure in this way is known as a **parameter** of the (procedure) call. For this to work, the definition of the procedure involved must also be amended so that its parameter requirements are specified:

```
PROCEDURE DrawLine (Length : IN Integer) IS
   -- To output a line of Length stars
   Num : Integer;  -- A local variable of the procedure
BEGIN
   New_Line;
   Num := 1;
   WHILE Num <= Length LOOP
      Put(Item => "*");
      Num := Num + 1;
   END LOOP;
   New_Line;
END DrawLine;
```

This states:

i) That the procedure `DrawLine` requires a parameter value of type `Integer` be provided whenever it is called.

ii) That when `DrawLine` is entered, a local `Integer` storage location named `Length` is allocated and the value supplied in the call step is copied into this storage location as a constant value.

`Length`	is known as a **formal parameter** (FP)
`Length : IN Integer`	is a (formal) **parameter specification**

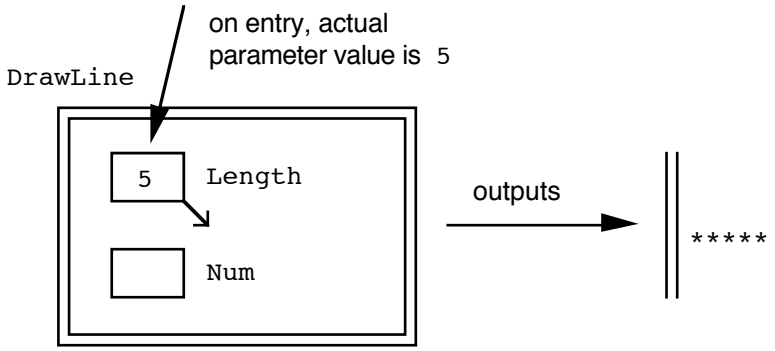
iii) The value supplied in a call step, eg. the value `20` in `DrawLine(Length => 20)` is known as an **actual parameter** (AP). An AP need not be a constant or a single variable (for example); it can be any expression, no matter how complicated, provided that, when evaluated, it gives a value of the type specified for that parameter in the procedure definition.

iv) When a call step is written in a program, the AP value to be supplied is related to the corresponding FP by an association of the form FP => AP, eg. `Length => 20`.

v) The execution life-time of a FP is like that of a local variable, ie. for the duration of the execution of the procedure within which it is specified. Thus when the procedure above finishes its execution, the storage allocated to hold the FP `Length` is freed, so that `Length` no longer exists after the procedure returns.

Example calls of `DrawLine`

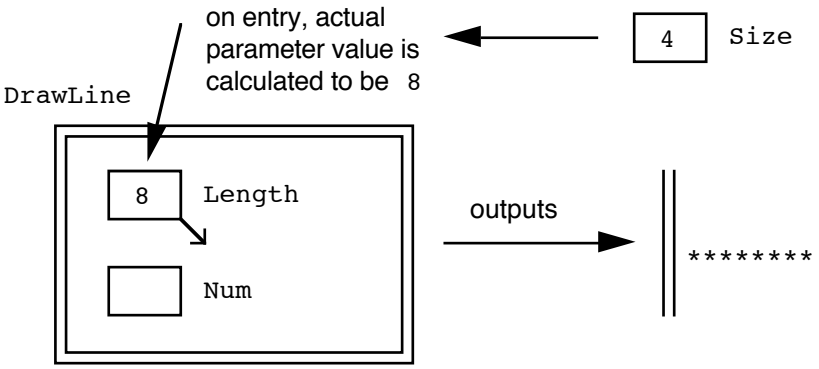
a) `DrawLine(Length => 5);`



b) Assuming the existence of the global variable `Size` defined by:

```
Size : Integer;
........;
Get(Item => Size);   -- Say 4 is read into Size
DrawLine(Length => 2*Size);
```

At the call to DrawLine, the computer takes the current value of Size, ie. 4, multiplies this value by 2 to obtain the value 8, which is then supplied as the actual parameter in the procedure call, ie:



5.6 'Copy-in' parameters

The IN keyword in the formal parameter specification above signifies that Length is said to be an **IN parameter** (sometimes referred to as a **'copy-in' parameter**) of the procedure, ie. a value is being supplied to the procedure from the point in the program where the procedure was called, and that value is copied into the FP Length.

It is important to realise this keyword IN also indicates that Length is a constant during each execution of the procedure (not a variable whose value may be altered). Thus Length can only be used where a *value* would normally be used, eg. in the expression on the right hand side of an assignment step, but never on the left hand side, since this would imply that it was a variable whose value could be changed by the assignment step. Parameters like this are sometimes referred to as **'retrieve-access-only'**; namely, that inside the procedure the formal parameter value may be retieved, ie. have its value extracted from its storage location, but never assigned, ie. may not have the value held in its storage location changed. (Note that this manner of use is emphasised by the notation used for the storage location for an IN parameter in the diagrams above, where an outward pointing arrow is used to indicate that the value stored may only be retieved).

5.7 More than one parameter

It is often necessary to pass two or more pieces of information to a procedure. For example we might output a blank margin of variable size before each line of stars. To permit this we would modify DrawLine thus:

```
PROCEDURE DrawLine (Margin : IN Integer; Length : IN Integer) IS
   -- To output Margin blanks, followed by
   -- a line of Length stars
   Count : Integer;
BEGIN
   New_Line;
   -- First output Margin blanks
   Count := 1;
   WHILE Count <= Margin LOOP
      Put(Item => " ");
      Count := Count + 1;
   END LOOP;
   -- Next output Length stars
   Count := 1;
   WHILE Count <= Length LOOP
      Put(Item => "*");
      Count := Count + 1;
   END LOOP;
   New_Line;
END DrawLine;
```

Example calls

a) `DrawLine(Margin => 3, Length => 5);` outputs → `ΔΔΔ*****`

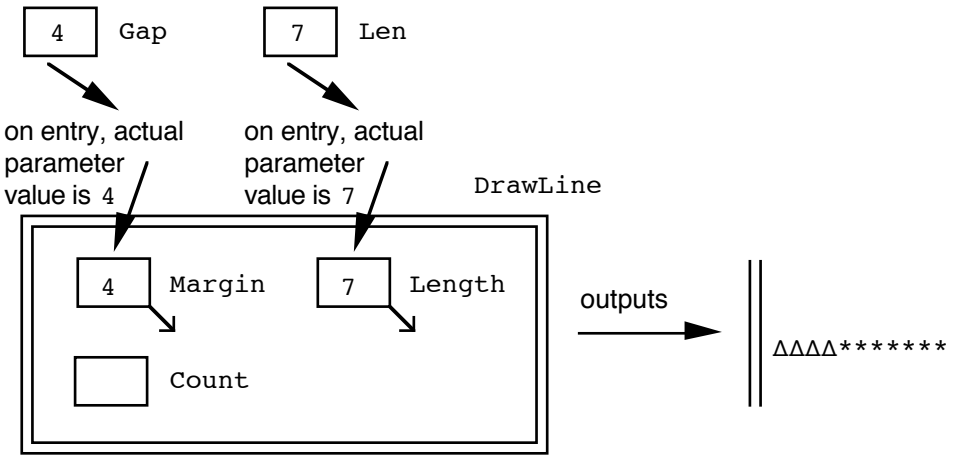
b) `DrawLine(Margin => 0, Length => Size/2);` outputs → `****`
(if the value of `Size` was `8` or `9`)

c) Assuming the following variables have been declared: `Gap, Len : Integer;`

and that by read (ie. `Get`) or assignment steps `Gap` and `Len` have been set to `4` and `7` (respectively). Then the procedure call:

```
DrawLine(Margin => Gap, Length => Len);
```

produces:



5.8 Parameter specification rules (for this unit - more to come later)

The parameter specifications for a procedure are always included in its definition heading. They are written in round brackets after the procedure's name. The brackets enclose a sequence of parameter specification parts *separated* by *semi-colons*. Each such specification part is of the form:

List_of_one_or_more_identifiers_separated_by_commas `:` `IN` *Type_name*

eg:
a) `Margin IN Integer; Length : IN Integer` {this is equivalent to (b)}
b) `Margin, Length : IN Integer` {this is equivalent to (a)}
c) `Quantity : IN Integer; UnitPrice : IN Float`

Each FP identifier specified indicates that, when called:

i) The procedure requires one AP parameter of the type specified.

ii) The FP identifier in question names a local constant (of the specified type) which is established, on entry to the procedure, by copying into the FP the value obtained by evaluating the corresponding AP.

iii) Each AP value is related to the corresponding FP by an association of the form:

FP => AP eg: `Margin => Gap` or `Length => Len`

These associations don't have to be in the same order as the order they appear in the procedure declaration, but it is good programming practice to do so. In a procedure call, each association should be separated from the next by a *comma* (notice this separator is not a *semi-colon* - which is used as a separator in the procedure definition).

Note: Identifiers used for FPs*can* be the same as the names of global (ie. main program) variables, but to avoid any risk of confusion, you are strongly urged to *avoid this practice*. Using the same identifier for FPs in two (or more) *different procedures* is reasonable, for FPs are *local* to their own procedure (ie. cannot be referenced from anywhere outside the relevant procedure), and hence confusion should not arise. Identifiers used for FPs should, of course, always be as meaningful as possible. Similar remarks apply to identifiers used for local variables.

5.9 Advantages of using parameters

a) They generally permit the construction of clearer, more easily understood programs. For example, on inspecting the *one* step:

```
DrawLine(Margin => 2, Length => 3);
```

We can see what values it uses and gain some understanding of what this step means without referring to any other part of the program (other than the procedure declaration).

b) They allow procedures to be 'self-contained', ie. to be understood on their own without looking at the rest of a program. Such 'self-contained' procedures are useful because they can be used to construct 'library' packages - ie. an Ada package comprising a library of procedures which perform a variety of useful functions, which can then be imported into other programs when required (by using `WITH` and `USE` context clauses). Thus procedures written by a programmer in the solution of one problem may be re-used (if appropriate) in the solution of later problems (either by the same programmer or by other programmers who have access to the package containing those procedures).

c) Their use enhances the clarity and ease of program development by the method of top-down design or stepwise refinement. Subsequent examples will be used to demonstrate this.

5.10 Using procedures to clarify program structure

The discussion so far has suggested writing procedures to avoid rewriting some groups of steps which need to be obeyed at more than one place in a program. But in fact there is another good reason for using procedures. It arises particularly in conjunction with stepwise refinement. In the example [4.6] we had in effect:

```
Get(Item => Number)
WHILE Number > 1 LOOP
     Produce the factors of this number ..... A
     Get(Item => Number)
END LOOP
```

In [4.6] we expanded **A** producing the 'trial divisor' loop preceded by initialisation steps and followed by the output of the number of divisors. We substituted this program fragment for **A**, and left *Produce the factors* (or something like it) as a comment. But with large steps such as **A** here, it is often better to express the expansion of the step as a separate procedure with something like '`ProduceFactors`' as its name. Now we don't need the comment (the name mentioned in the procedure call step conveys the relevant information), and the resulting main program has a simpler, more easily understood structure. The listing on the following page shows `Factors` amended in this way and also incorporating the `DrawLine` procedure.

5.11 Position of procedure calls

In general any procedure can be called from within another procedure as well as from the main program but for any particular call to be valid, the called procedure must be 'visible' to the call step. For the moment it suffices to know that this will be the case if:

i) the call step occurs in the main body and the called procedure is defined in the declaration part of the same program,

ii) the call step occurs in a procedure and this procedure and the called procedure are both defined in the declaration part of the same program, with the declaration of the called procedure preceding that of the procedure which includes the call,

iii) the called procedure is 'imported' (by using `WITH` and `USE` context clauses) into the program in which the call step occurs (as with standard input/output procedures such as `Get` and `Put`).

```
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 ProduceFactors (Number : IN Integer) IS
      -- Produce and output the factors, ie. deal with this Number
      TrialDivisor  : Integer;  -- To hold each possible
                                -- (ie. 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
   DrawLine;
   New_Line;
   Put(Item => "Type in a number greater than 1 (0 or 1 to quit): ");
   Get(Item => UsersNumber);
   WHILE UsersNumber > 1 LOOP
      ProduceFactors(Number => UsersNumber);
      DrawLine;
      New_Line;
      Put(Item =>
         "Type another number greater than 1 (0 or 1 to quit): ");
      Get(Item => UsersNumber);
   END LOOP;
END Factors;
```