

Chapter 6

□ PROCEDURES & FUNCTIONS

Procedures and functions are *sub-programs* that perform individual tasks within a main program. Most programming languages support the sub-program concept and have done so since programming languages were first developed. Large, complex software systems would scarcely be possible without using subprograms because they allow a large program to be broken into smaller manageable subprograms.

The Benefits of Sub-Programs

1. (*Modularization*) Allows large programs to be broken into smaller subprograms, each of which does a single task.
2. (*Code Re-Use*) Once a sub-program has been written it may be used in any part of the program.
3. (*Error Reduction*) Code re-use reduces code-writing and hence reduces errors.
4. (*Easy Debugging*) Tracing and finding bugs in a large program is easier if it has been modularized
5. (*Easy Revision*) Modularized programs are easier to revise, update and extend because not all sub-programs need to be changed.

Sub-programs in Pascal have two forms : **Procedures** and **Functions**. Procedures, as we will see, are more general than functions and although we can always write a procedure to take the place of a function , they both have their uses.

6.1 PROCEDURES

Procedures are *named* pieces of program code that may be used (called, invoked) from anywhere in a program that contains the procedure definition in its

<procedure-function declaration part>.

Procedures are general-purpose subprograms whereas functions are more specialized in that they do a calculation and *return* a single value to the calling program or subprogram.

Let us look at an example before we get into the formalities. Recall that in Chapter 4 we said that all programs have the general outline :

```
PROGRAM <Name>;
  < Declaration Section >
BEGIN
  < Input Section >
  < Process Section >
  < Output Section >
END.
```

A typical example is the QuadraticIO program we saw in Chapter 4. Here it is again.

```
program QuadraticIO;
{ This program calculates the roots of the quadratic equation  $ax^2 + bx + c = 0$ 
  and assumes  $b^2 > 4ac$  }
  var a : REAL;
      b : REAL;
      c : REAL;
      r1 : REAL;
      r2 : REAL;

  begin
    Readln(a);
    Readln(b);
    Readln(c);

    r1 := (-b + Sqrt(b*b - 4.0*a*c))/(2.0*a);
    r2 := (-b - Sqrt(b*b - 4.0*a*c))/(2.0*a);

    Writeln('a has the value :', a);
    Writeln('b has the value :', b);
    Writeln('The roots are :', r1, ' and ', r2);
  end.
```

We now re-write this program as a set of procedures : one for input, one for processing and one for output, as shown below. First we give the syntax for procedures.

```

PROCEDURE <identifier> ( <formal parameter list> );
    <CONSTant definition part>; optional
    <TYPE definition part>; optional
    <VARiable declaration part>; optional
    <PROCEDURE definition part>; optional
begin
    <statement part> optional
end;

```

The (<formal parameter list>) part is also optional. Notice that this syntax is essentially the same as the PROGRAM syntax.

```

PROGRAM QuadraticIOProcs;
{ This program calculates the roots of the quadratic equation  $ax^2 + bx + c = 0$ 
  and assumes  $b^2 > 4ac$  }
VAR
    a : REAL;
    b : REAL;
    c : REAL;
    r1 : REAL;
    r2 : REAL;

PROCEDURE InputValues (VAR x,y,z : REAL);
BEGIN
    Readln(x);
    Readln(y);
    Readln(z);
END; { PROC InputValues }

PROCEDURE Calculate (a, b, c : REAL; VAR v1, v2 : REAL);
BEGIN
    v1 := (-b + Sqrt(b*b - 4.0*a*c))/(2.0*a);
    v2 := (-b - Sqrt(b*b - 4.0*a*c))/(2.0*a);
END; { PROC Calculate }

PROCEDURE OutputResults (q, r, s, x1, x2 : REAL);
BEGIN
    Writeln('a has the value :', q);
    Writeln('b has the value :', r);
    Writeln('c has the value :', s);
    Writeln('The roots are :', x1, ' and ', x2);
END; { PROC OutputResults }

{ This is the start of the MAIN program statements }

BEGIN
    InputValues(a, b, c);
    Calculate(a, b, c, r1, r2);
    OutputResults(a, b, c, r1, r2);
END. {PROG QuadraticIOProcs}

```

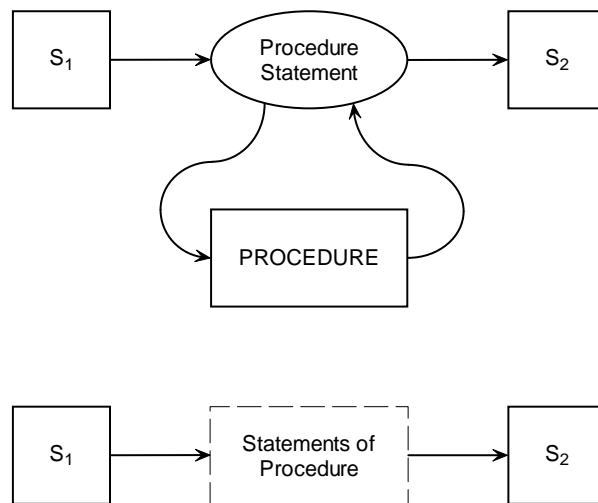


Figure 6.1. Using a Procedure.

6.1.1 Procedures without Parameters

These are procedures that do not require any ‘input’ in the form of parameters. For example, a standard operation in the output of most programs is to *Pause* after a certain amount of output, to allow the user to view the contents of the screen. Suppose for example, we wish to pause after every 15 lines of output, to view the output. A program to do this might be as follows :

```

PROGRAM Pauser;
< declaration part>;

BEGIN
  < Read in Data >;
  < Process Data >;
  < Output 15 lines >;

  Write('Press RETURN to continue...');
  Readln;

  < Output 15 lines >;

  Write('Press RETURN to continue...');
  Readln;

  < Output 15 lines >;

  Write('Press RETURN to continue...');
  Readln;

  < etc.>
end.
  
```

We can see that each time we need to pause we have to write two lines of code

```
Write('Press RETURN to continue...'); Readln;
```

A better way is to put these two lines in a parameterless procedure and call the procedure each time it is needed. This is done as follows :

```
PROGRAM Pauser;
  PROCEDURE Pause;
  BEGIN
    Write('Press RETURN to continue...');
    Readln;
  END; { PROC Pause }

  BEGIN
    < Read in Data >;
    < Process Data >;
    < Output 15 lines >;

    Pause;

    < Output 15 lines >;

    Pause;

    < Output 15 lines >;

    Pause;

    < etc.>
  end.
```

Here is a procedure for writing out a fixed message

```
PROCEDURE ClassHeading;
BEGIN
  Writeln('=====');
  Writeln('=          UNIVERSITY COLLEGE DUBLIN          =');
  Writeln('=');
  Writeln('=  MIS 387 :  Computer Programming Techniques  =');
  Writeln('=');
  Writeln('=          First Semester  1999          =');
  Writeln('=====');
END; { PROC ClassHeading }
```

6.1.2 Procedures with Parameters

These procedures allow us to control, to some extent, what a procedure does when it is called by sending information to the procedure.

Suppose that we need a program that skips 4 lines after every 10 lines of output. We could have 4 `Writeln;`'s after every 10 lines of output but this would be very tedious to type in and also error-prone.

A better way is to write a procedure to skip 4 line and then call this procedure every time we need to skip 4 lines. Here is a partial program that does this.

```
PROGRAM Skipper;
  < declarations >

  PROCEDURE Skip4Lines;
  BEGIN
    Writeln;
    Writeln;
    Writeln;
    Writeln;
  END; { PROC Skip4Lines }

  BEGIN
    < Read in Data >;
    < Process Data >;
    < Output 10 lines >;

    Skip4Lines;

    < Output 10 lines >;

    Skip4Lines;

    < Output 10 lines >;

    Skip4Lines;

    < etc.>
  end.
```

Suppose now that we want to vary the number of lines that we skip. We could write separate procedure for each different number of skipped lines but again, there's a better way – procedures with **parameters**. We write the procedure that can be told the number of lines to skip and we pass this information as a parameter to the procedure when we call it.

The following program uses a procedure with a single parameter and shows how it is used (called).

```
PROGRAM SkipperPar;
  < declarations >

  PROCEDURE SkipLines(NoLines : INTEGER);
    VAR
      i : INTEGER;
    BEGIN
      FOR i := 1 TO NoLines DO BEGIN
        Writeln;
      END
    END; { PROC SkipLines }

  BEGIN
    < Read in Data >;
    < Process Data >;
    < Output 10 lines >;

    SkipLines(4);

    < Output 10 lines >;

    SkipLines(3);

    < Output 10 lines >;

    N := 20;
    SkipLines(N);

    < Output 10 lines >;
    SkipLines(N/4);
    < etc.>
  end.
```

6.1.3 Value and Variable Parameters

The integer variable `NoLines` is a **value parameter** of the procedure and when the procedure is called, e.g., `SkipLines(4)`, the **actual parameter** 4 is copied into `NoLines` and the **for** loop is executed 4 times. The variables `NoLines` and `i` inside the procedure definition are called **local variable** and exists only while the procedure is being executed. When the procedure ends execution these variables disappear. For this reason value parameters are used when we want to *input* information to the procedure.

Notice that the actual parameter may be an constant, variable, or expression.

In the previous example we sent information to the procedure through a value parameter. This is a one-way information route : from the calling statement to the procedure.

What do we do if we want to send back information? We use **variable** or **var** parameters. Here is an example : we wish to write a procedure to solve the old quadratic equation problem. We need to send information `a,b,c` to the procedure and we want to get the roots `v1, v2` back from the procedure. Hence the procedure must have both *input* and *output* parameters.

In the following procedure `a,b,c` are input (value) parameters while `v1,v2` are output (variable) parameters.

```

PROCEDURE QuadRoot (a, b, c : REAL; VAR v1, v2 : REAL);
BEGIN
    v1 := (-b + Sqrt(b*b -4.0*a*c)/(2.0*a);
    v2 := (-b - Sqrt(b*b -4.0*a*c)/(2.0*a);
END; { PROC QuadRoot }

```

Notice that a, b, c are value parameters which have values copied into them when the procedure is used, while $v1, v2$ are VAR parameters that get values calculated in the procedure and this is sent back to the statement that uses the procedure.

We would use this procedure as follows :

```

PROGRAM QuadraticIOProcs;
{ This program calculates the roots of the quadratic equation  $ax^2 + bx + c = 0$ 
  and assumes  $b^2 > 4ac$  }
VAR
    q : REAL;
    r : REAL;
    s : REAL;
    r1 : REAL;
    r2 : REAL;

PROCEDURE QuadRoot(a, b, c : REAL; VAR v1, v2 : REAL);
BEGIN
    v1 := (-b + Sqrt(b*b -4.0*a*c)/(2.0*a);
    v2 := (-b - Sqrt(b*b -4.0*a*c)/(2.0*a);
END; { PROC QuadRoot }
{ This is the start of the MAIN program statements }

BEGIN
    Readln(q);
    Readln(r);
    Readln(s);
    QuadRoot(q, r, s, r1, r2);
    Writeln('a has the value :', q);
    Writeln('b has the value :', r);
    Writeln('c has the value :', s);
    Writeln('The roots are :', r1, ' and ', r2);
END. {PROG QuadraticIOProcs}

```

We can use the procedure any time we please. For example

```
PROGRAM QuadraticIOProcs;
{ This program calculates the roots of the quadratic equation  $ax^2 + bx + c = 0$ 
  and assumes  $b^2 > 4ac$  }
VAR
  a,q  :   REAL;
  b,r  :   REAL;
  c,s  :   REAL;
  r1   :   REAL;
  r2   :   REAL;

PROCEDURE QuadRoot(a, b, c : REAL; VAR v1, v2 : REAL);
BEGIN
  v1 := (-b + Sqrt(b*b -4.0*a*c))/(2.0*a);
  v2 := (-b - Sqrt(b*b -4.0*a*c))/(2.0*a);
END; { PROC QuadRoot }

BEGIN { This is the start of the MAIN program statements }
  Readln(q);
  Readln(r);
  Readln(s);

  QuadRoot(q, r, s, r1, r2);

  Writeln('a has the value :', q);
  Writeln('b has the value :', r);
  Writeln('c has the value :', s);
  Writeln('The roots are   :', r1, ' and ', r2);

  Readln(a);
  Readln(b);
  Readln(c);

  QuadRoot(a, b, c, r1, r2);

  Writeln('a has the value :', a);
  Writeln('b has the value :', b);
  Writeln('c has the value :', c);
  Writeln('The roots are   :', r1, ' and ', r2);
END. {PROG QuadraticIOProcs}
```

Scope and Extent of Global and Local variables

```

PROGRAM Scope;
VAR
    a,q : REAL;
    b,r : REAL;
    c,s : REAL;
    r1 : REAL;
    r2 : REAL;

PROCEDURE XXX (VAR x,y,z : REAL);
BEGIN
END; { PROC XXX }

PROCEDURE YYY (a, b, c : REAL; VAR v1, v2 : REAL);
BEGIN
END; { PROC YYY }

PROCEDURE ZZZ (q, r, s, x1, x2 : REAL);
BEGIN
END; { PROC ZZZ }

BEGIN
    XXX(a, b, c);
    YYY(a, b, c, r1, r2);
    ZZZ(a, b, c, r1, r2);
END. {PROG Scope}
    
```

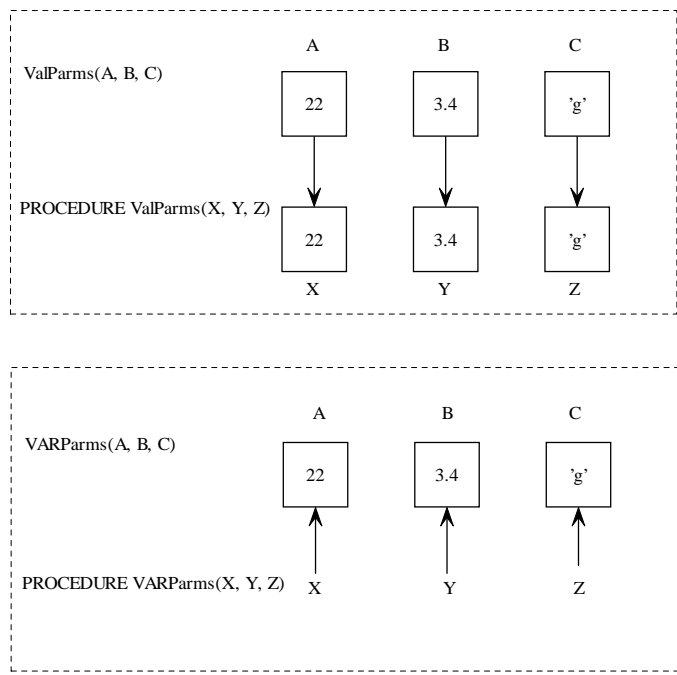


Figure 6.2. Value and VAR parameters.

```
PROGRAM QuadraticIOProcs;
{ This program calculates the roots of the quadratic equation  $ax^2 + bx + c = 0$ 
  and assumes  $b^2 > 4ac$ . It uses local variables only.}

PROCEDURE InputValues (VAR x,y,z : REAL);
BEGIN
  Readln(x);
  Readln(y);
  Readln(z);
END; { PROC InputValues }

PROCEDURE Calculate (a, b, c : REAL; VAR v1, v2 : REAL);
BEGIN
  v1 := (-b + Sqrt(b*b -4.0*a*c)/(2.0*a);
  v2 := (-b - Sqrt(b*b -4.0*a*c)/(2.0*a);
END; { PROC Calculate }

PROCEDURE OutputResults (q, r, s, x1, x2 : REAL);
BEGIN
  Writeln('a has the value :', q);
  Writeln('b has the value :', r);
  Writeln('c has the value :', s);
  Writeln('The roots are :', x1, ' and ', x2);
END; { PROC OutputResults }

PROCEDURE Main;
VAR
  a,q : REAL;
  b,r : REAL;
  c,s : REAL;
  r1 : REAL;
  r2 : REAL;

BEGIN
  InputValues(a, b, c);
  QuadRoot(a, b, c, r1, r2);
  OutputResults(a, b, c, r1, r2);

  InputValues(q, r, s);
  QuadRoot(q, r, s, r1, r2);
  OutputResults(q, r, s, r1, r2);

  { This is the start of the MAIN program statements }
BEGIN
  Main;
END. {PROG QuadraticIOProcs}
```

Menus and Interface Design *to be expanded later*

6.2 FUNCTIONS

Functions are similar to procedures except they are used to do calculations. These result in a single value which is *returned* by the function that was called by the main program.

Let us look at an example first and then give the formal syntax.

```
PROGRAM FuncCaller;
VAR
  x, y, z : REAL;
  FUNCTION Average3 ( a,b,c : REAL ) : REAL;
    VAR
      sum : REAL;
    BEGIN
      Sum := a + b + c;
      Average3 := Sum/3.0
    END; {FUNC Average3}
BEGIN { start of main program }
  Write('Type in 3 real values :');
  Readln(x,y,z);
  Avg := Average3(x,y,z);
  Writeln('The average of 'x:5:2, y:5:2, z:5:2, ' = ', Avg:7:3);
END. {PROG Average3}
```

The syntax of a function is very similar to that of a procedure or program :

```
FUNCTION <identifier> ( <formal parameter list> ) : < type>;
  <CONSTant definition part>; optional
  <TYPE definition part>; optional
  <VARiable declaration part>; optional
  begin
    <statement part> Must include Function assignment
  end;
```

There are two important differences between a function and a procedure :

1. A function, because it returns a value, must have a **type** associated with it. This type is specified in the function *header*.
2. The function statements must include an assignment statement in which the function name appears on the left-hand-side of an '=' operation. This where the function is given its value.

Laboratory Exercise No. 6: *Using and Creating Procedures .*

This exercise is about using some additional built-in procedures of Turbo Pascal to write our own procedures.

Turbo Pascal has various procedures and functions that allow the programmer to get the Date and Time. Some of these are below. Check the Turbo Pascal Programmer's Guide for others.

The program below shows the use of the `GetDate` and `GetTime` procedures that are part of Turbo Pascal's `Dos` unit. We include the `Dos` unit in our program by adding the statement

```
USES Dos;
```

at the top of the program as shown below. Once we do this all the procedures and functions in this unit are available as built-in functions and procedures.

```
{-----}
PROGRAM LabExer6; { MIS 387 1999}
{-----}
  USES Dos;
  VAR
    Year, Month, Day, DayOfWeek : WORD;
    Hour, Min, Sec, Sec100      : WORD;
  BEGIN
    GetDate(Year, Month, Day, DayOfWeek);
    Writeln('The date is : ', Year:5, Month:5, Day:5, DayOfWeek:5);

    GetTime(Hour, Min, Sec, Sec100);
    Writeln('The time is :', Hour:5, Min:5, Sec:5, Sec100:5);
    Readln;
  END. {PROGRAM LabExer6}
```

The program's date output would be something such as
1999 10 30 7

which stands for Sat 30 Oct 1999.

The program's time output would be something such as
21 35 23 69

which stands for 9:35:23pm, ignoring the hundredths part.

Using the procedures above write a procedure `WriteDateTime` that outputs the date and time in the following format :

```
DATE : dd - mmmm - yyyy
```

where `mmmm` is Jan, Feb, Mar, etc., and

```
TIME : hh:mm:ss am or pm
```

where `hh` is 1 – 12.

This procedure will have no parameters and will use (call) `GetDate` and `GetTime` to get the date and time in numeric format, which it then transforms into the format above.