

Chapter 4

□ ELEMENTARY STATEMENTS

In this chapter we cover *Expressions, Assignment statements, and Input-Output statements*. We will also look briefly at *Procedure and Function statements*.

4.1 Expressions

An expression is a rule for computing a value from variables and constants. There are three type of expression

1. Arithmetic
2. Boolean
3. String and Character

Here are some examples.

Arithmetic Expressions The expression on the right is the Pascal translation of the mathematical expression on the left. Given values for the variables in an expression a program *evaluates* the expression so that it has a value when the evaluation is complete.

$$\begin{array}{lcl} \frac{x + y}{a + b} & \implies & (x + y)/(a + b) \\ 2.5x + 95.7y & \implies & 2.5 * x + 95.7 * y \\ ax^2 + bx + c & \implies & a * x * x + b * x + c \\ \frac{b + \sqrt{b^2 - 4ac}}{2a} & \implies & ((b + \text{Sqrt}(b * b - 4.0 * a * c))/(2.0 * a)) \end{array}$$

The expression on the right must be formed from Pascal characters and cannot contain special mathematical symbols. Also, for the expression to have a value each variable in the expression must have a value. For example, in the first expression above the variables **x**, **y**, **a**, **b** must have values before the expression can be evaluated. If the values of these variables are 2.2, 1.4, 2.9, and 3.3 respectively then the expression evaluates to the value 0.5806451612903.

Although this may seem obvious, forgetting to give variables values is a common source of errors in programs.

Boolean Expressions These are expressions that evaluate to `true` or `false`. Some examples are

```
(q <= x)
(x <> y)
(4 < 5) AND ( 7.9 <= 32.1)
(m < n) OR (m < a) AND (q <= x)
test1 OR test2
```

We generally use Boolean expressions in selection and repetition control structures to test for some condition. Here are some examples :

In the last expression `test1` and `test2` must be boolean constants or variables, i.e., they must have the values `true` or `false`.

Character and String Expressions These combine strings and characters to form strings. Consider the following program :

```
program MoreStrings;
const
  Answer = 'Y';           { a CHAR constant }
  Year   = 'Third Commerce'; { a STRING constant }
  space  = ' ';
var
  s1 : STRING;
  s2 : STRING;
  s3 : STRING;
  s4 : STRING;
begin
  s1 := 'Janet';
  s2 := 'Johnson';
  s3 := s1 + space + s2;
  s4 := 'Is ' + s3 + ' in ' + Year + '??';

  Writeln(s4, ' --- ', Answer);
end.
```

The string expression to the right of `s3` evaluates to `Janet Johnson` the expression to the right of `s4` evaluates to `Is Janet Johnson in Third Commerce?` The program writes out the following on the screen

```
Is Janet Johnson in Third Commerce? --- Y
```

Note carefully that *blanks* in strings are significant, i.e., they are just as important as non-blank characters.

Before we go to the next section remember :

All constants and variables in expressions must be declared and have values before the expression can be evaluated

4.2 Assignment Statements

Assignment statements are used to give variables values, i.e., they place values in the memory boxes named by variables. The syntax of the assignment statement is as follows :

$$\langle \text{variable} \rangle := \langle \text{expression} \rangle;$$

The *semantics* or meaning is : evaluate the expression on the right of the := operator and assign (give) this value to the variable on the left of the := operator. In other words the assignment statement places the value of the expression in the memory box named or labelled by the variable. The value that was previously in the box is over-written by the value of the expression.

Here are some examples of assignment statements :

```

i := 3; { replaces the current value of i with 3}
i := k; { replaces the current value of i with the current value of k}
a := (b + c)/z
j := j + 1
Switch := (i < 0) AND (ch <> ' ')

```

Rule of Assignment In general, the *type* of the variable on the left of an assignment statement must be the same as the type of the value of the expression on the right. An exception is : a REAL variable may be assigned an INTEGER value.

Examples :

```

x := 5.6; { x must be REAL}
a := i*j; { a may be REAL or INTEGER}
k := k + 1; { k may be REAL or INTEGER}
Sw := (x<y) OR (y<z) OR (a='$'); {Sw must be BOOLEAN}
b := 'Bamboo'; { b must be STRING}

```

We now write a simple program that uses assignment statements to calculate the roots of a quadratic equation

$$ax^2 + bx + c = 0.$$

This equation has the roots defined by the expression

$$r_1, r_2 = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}.$$

```
program Quadratic;  
{ 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  
    a := 2.5;  
    b := 12.9;  
    c := 1.2;  
  
    r1 := (-b + Sqrt(b*b - 4.0*a*c))/(2.0*a);  
    r2 := (-b - Sqrt(b*b - 4.0*a*c))/(2.0*a);  
  
  end. { PROG Quadratic}
```

Note that this program is useless because

1. It has no output.
2. It works for only one set of values for a , b , and c .

We will see in the next section how to make this program more useful by putting *Input* and *Output* statements into it.

4.3 Input and Output Statements

The program in the preceding section, although valid, is useless because it has no input or output statements.

An *input statement* allows us to get data(values) from an input device and store them in variables (memory boxes). An *output statement* allows us to retrieve data(values) from variables or constants(memory boxes) and display them on an output device.

For example the following three statements *read* in data from the keyboard, perform a calculation, and *write* the result to the screen :

```

program ConvertF2C;
{ This program coverts Fahrenheit to Centigrade}
var
    Fahr, Cels : REAL;

begin
    Readln(Fahr);
    Cels := (5.0/9.0)*(Fahr - 32.0);
    Writeln(Cels)
end{Prog ConvertF2C}
    
```

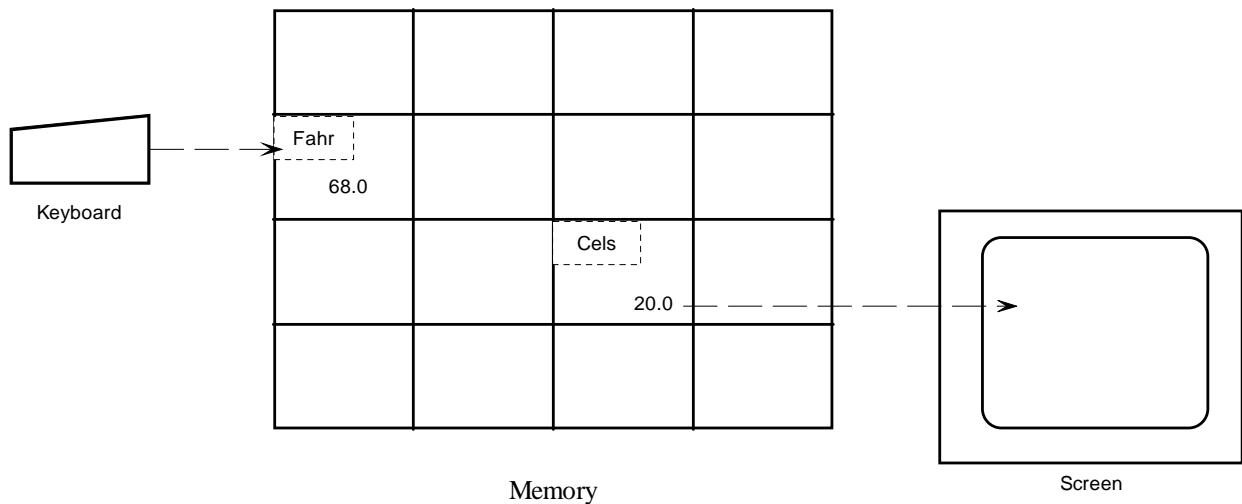


Figure 4.1. Input and Output.

We now modify the program *Quadratic* in the previous section so that we can input and output values.

```

program QuadraticI0;
{ 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('c has the value :', c);
    Writeln('The roots are   ', r1, ' and ', r2);
  end.

```

We note that the first 3 statements are input statements; the next 2 are process (calculation) statements; and the final 3 are output statements. In general, most programs have this form.

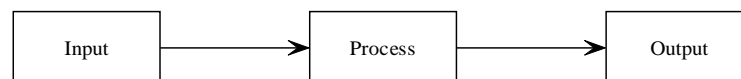


Figure 4.2. General Form of Programs.

Input Statements The syntax of an input statement is :

```

  Readln(<list of variables>);
  and
  Read(<list of variables>);

```

where the variables must not be of type `BOOLEAN`.

EXAMPLE : `Input Readln(a,b,c);` This means get the first number from the key board and store it in the variable `a`; get the next and store it in `b`; get the next and store it in `c`; skip to the next line of input, if any. \square

When typing in values at the keyboard we must separate these values by one or more blanks. For example, if we typed in the following on one line

123 49 657

then **a** would get the value 123, **b** would get the value 49, and **c** would get the value 657. If, on the other hand, we typed in the line

12349657

the **a** would get the value 12349657 and the program would stop and wait for you to type in values for **b** and **c**.

The statement above is equivalent to the following three statements

```

    Read(a);
    Read(b);
    Readln(c);
or
    Read(a, b); Readln(c);

```

The difference between the two types of input statement is this : the `Read(<list of variables>)` form reads in values for each variable and stops on the line that where these values appear; any subsequent input statement will start reading from this same line. The `Readln(<list of variables>)` form reads in values for all variables and then skips to the start of the next line; any subsequent input statement will start reading from this new line.

Consider the following partial program

```

program InputExample;
  var
    x, y    : REAL;
    z, a, b : REAL;
    .
    .
    .
    Readln(x, y, z);
    Read(a,b);
    .
    .
    .

```

If we type in the lines

12.3 5.67 0.07 51.3 72.9

29.7 56.7

then the variables get the following values :

$x = 12.3$, $y = 5.67$, $z = 0.07$, $a = 29.7$, $b = 56.7$

Notice that the values 51.3 and 72.9 are ignored because the `Readln(x,y,z)` skips to the next line after it has obtained values for **x**, **y**, and **z**.

If we alter the input statements above as follows

```
Read(x, y, z);  
Readln(a,b);
```

then the variables get the following values

$x = 12.3$, $y = 5.67$, $z = 0.07$, $a = 51.3$, $b = 72.9$

The statement `Readln`; with no variable list just skips to the next line without getting any input values (it doesn't ask for any). Hence

```
Readln(x, y, z);  
is equivalent to  
Read(x, y, z);  
Readln;
```

We will see in later chapters that input statements can be used to get data from input devices such as floppy and hard disks, cd-roms and communications ports (e.g. telephone lines).

Output Statements These are very similar to input statements except they send data out to output devices such as screen, printer, disks, and communications ports.

The syntax of the two output statements is :

```
Writeln(<list of variables, constants, literals, function values>;  
and  
Write(<list of variables, constants, literals, function values>;
```

The `Writeln(<list>)` form writes a list of values out to the screen and then skips to the next line; any subsequent output statement starts on this new line. The `Write(<list>)` form writes a list of values out to the screen and then stops; any subsequent output statement starts on the same line.

EXAMPLE : *Output Only.* Let us look at a program and see how the Write and Writeln statements work.

```

program OutputExample;
  const
    pi = 3.14159;
    e  = 2.17;
  var
    x, y    : REAL;
    z       : REAL;
    i, j, k : INTEGER;
  begin
    x := 12.345;
    y := 20.0;
    z := 4.0;
    i := 2;
    j := 10000;
    k := 222;

    Writeln('1234567890123456789012345678901234567890123');
    Writeln('The results of this program are as follows:');
    Writeln(x, y, z);

    Writeln(x:10:2, y:10:2, z:6:2);

    Write(i:5, j:10, k:4);
    Writeln(pi:10:5);

    Writeln(Sqrt(z):5:2, (y+z):5:1);
  end.

```

This program gives the following output:

```

1234567890123456789012345678901234567890123
The results of this program are as follows:
12.34520.04.0
   12.345   20.0   2.0
    2   10000 222   3.14159
 2.00 24.0

```

□

The Write or Writeln statements allow us to *format* the output. For example, the statement Writeln(x:10:2, y:10:2, z:6:2); above writes out the value of x and y using 10 spaces each with space for two digits after the decimal point. The general form of the formatted Write/Writeln statement is

```

Writeln(<variable>:<Total spaces>:<spaces after dec.point>

```

Note: The decimal point counts for one space. For integers and strings we do not specify `<spaces after dec.point>`

4.4 Procedure and Function Statements

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. They also allow us to re-use code that we have written because once a piece of code is written as a procedure we can use it anywhere in a program by *calling* the procedure by its name.

For example, the note at the end of Lab. Exercise 3 used two statements to pause the output of the program. Each time we wished to pause the program we included the two statements at the point where we wished to pause the program. Here is a better way, using a procedure.

```
PROGRAM PausesProg;
< CONSTANT definition part >; optional
< TYPE definition part >; optional
< VARIABLE declaration part >; optional

PROCEDURE Pause; { PROCEDURE definition part }
BEGIN
    Write('Press RETURN to continue...');
    Readln;
END; { PROC Pause }

BEGIN { statement part }
    .
    .
    .
    Pause;
    .
    .
    .
    Pause;
    .
    .
    .
    Pause;
END. { PROG PausesProg }
```

We have already used procedures in our programs : *Readln* and *Writeln* are built-in procedures to input and output data.

Functions are similar to procedures except that they *return* the value of some calculation. Some built-in functions are

1. $\sqrt{x} \rightarrow \text{Sqrt}(x)$
2. $x^2 \rightarrow \text{Sqr}(x)$
3. $\cos x \rightarrow \text{Cos}(x)$

We will cover procedures and functions in more detail in Chapter 6.

Laboratory Exercise No. 4: Compound Interest.

This is exercise in using procedures and functions that have been written by someone else.

IF $\mathcal{L}P$ is invested at interest i for n years the formula for the amount of money F in the account after n years is

$$F = P(1 + i)^n,$$

where i is in decimal form (not %) and all interest is compounded annually and remains in the account.

Write a Pascal program that

1. Prompts the user to input P , i , and n .
2. Calculates F using the supplied function `Power(x,n)` (this evaluates x^n)
3. Writes out the input data and the result F , along with appropriate descriptive headings.

Testing :

1. Test the program for various values of i and n , including the values 0.
2. What rate of interest doubles F in 10 years.
3. How many years does it take to double F at an interest rate of 10%.

Notes :

1. Use the `Pause` procedure below to pause the output at the end of the program.
2. Use the recursive function `Power(x,n)` below.
3. The procedure and function must appear in the `<PROC/FUNC part>`, before the `<STATEMENT part>`.

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

FUNCTION Power(x:REAL; n:INTEGER) : REAL;
BEGIN
    IF n=0 THEN Power := 1.0
    ELSE      Power := x*Power(x,n-1);
END; { FUNC Power }
```
