

# Chapter 5

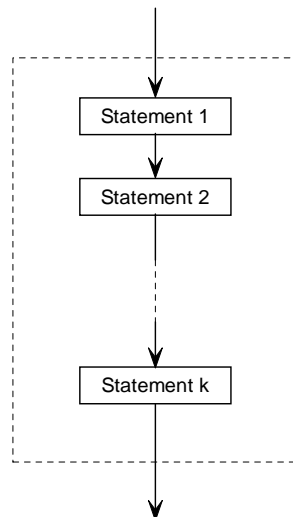
---

## □ CONTROL STATEMENTS

Control statements are used to change the order in which statements are executed in a program.

### 5.1 Sequence

So far we have used the *Sequence* control structure. This means that each statement is executed in the order in which it appears in the program. The flow diagram for this control structure is shown in Figure 5.1.



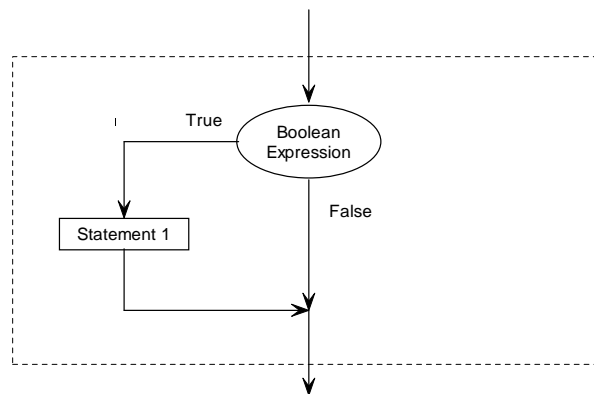
**Figure 5.1.** The SEQUENCE Control Structure.

## 5.2 Selection : IF Statements

We use these control structures if we wish to execute a statement depending on some condition. The first form of this control structure is called IF-THEN and has the following syntax :

```
if <BOOLEAN expression> then begin
    <statement sequence 1>;
end;
```

The flow diagram is in Figure 5.2



**Figure 5.2.** The IF-THEN Control Structure.

The following partial program uses this control structure :

```
Write('Type in your age :');
Readln(age);

if age >= 18 then begin
    Writeln('You are eligible to vote. ');
    Write('Type in your name :');
    Readln(Name);
end;
```

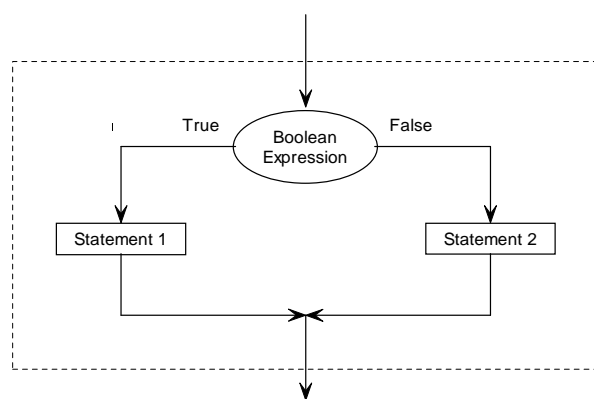
The second form is the IF-THEN-ELSE and allows us to select one of two possible statement sequences. It has the following syntax :

```

if <BOOLEAN expression> then begin
    <statement sequence 1>;
end
else begin
    <statement sequence 2>;
end;

```

The flow diagram is in Figure 5.3



**Figure 5.3.** The IF-THEN-ELSE Control Structure.

The following partial program uses this control structure :

```

Write('Type in your age :');
Readln(age);

if (age >= 18)then begin
    Writeln('You are eligible to vote. ');
    Write('Type in your name :');
    Readln(Name);
end
else BEGIN
    Writeln('You are not eligible to vote. ');
    Write('Type in your name :');
    Readln(Name);
    Writeln('We will contact you in ',
            (18-age):5, ' years');
end;

```

Note that there is no semi-colon after the first **end**.

The third form is the most general : it allows us to select 1 out of  $k$  alternative statement sequences. It has the following syntax :

```
if <BOOLEAN expression 1> then begin
    <statement sequence 1>;
end
else if <BOOLEAN expression 2> then begin
    <statement sequence 2>;
end
.
.
.
else if <BOOLEAN expression k-1> then begin
    <statement sequence k-1>;
end
else begin
    <statement sequence k>;
end;
```

The flow diagram is in Figure 5.4

The following partial program uses this control structure :

```
Write('Type in your mark :');
Readln(mark);

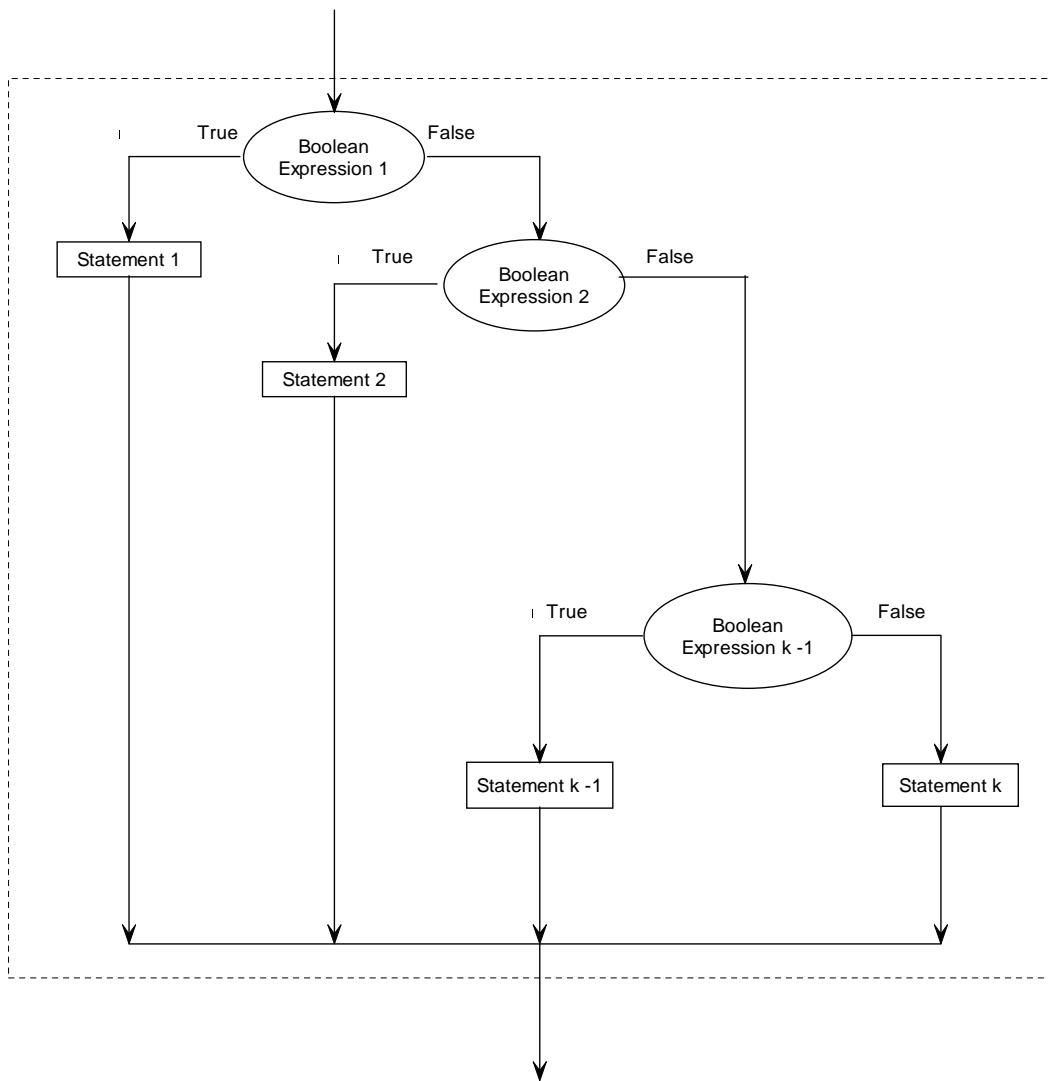
if (mark < 40) then begin
    Writeln('Fail');
end
else if (mark < 60) then begin
    Writeln('Pass');
end
else begin
    Writeln('Honours');
end;
```

Notice how the logic of the IF-THEN-ELSE statement above works : if the first boolean expression is *false* , i.e.,  $mark > 40$ , then we pass to the second boolean expression. If this expression is *true* then we must have  $40 \leq mark < 60$ . Likewise, if the second boolean expression is *false* we pass to the third statement sequence without a test for  $mark > 60$ – this must be *true* because the second test was *false* , i.e.,  $mark < 60$  is *false* , therefore  $mark > 60$  is *true* .

Let us look at a more complicated example. Suppose we wish to find out which of the following 5 ranges a mark falls into :

$[0 - 19]$ ,  $[20 - 39]$ ,  $[40 - 59]$ ,  $[60 - 79]$ ,  $[80 - 99]$ .

and assume that *mark* must be between 0 and 99, inclusive. The program below must perform 4 IF-tests to find out which of the 5 ranges the mark falls into.



**Figure 5.4.** The IF-THEN-ELSEIF Control Structure.

```
Write('Type in your mark :');
Readln(mark);

if (mark < 20) then begin
    Writeln('Range 1');
end
else if (mark < 40) then begin
    Writeln('Range 2');
end
else if (mark < 60) then begin
    Writeln('Range 3');
end
else if (mark < 80) then begin
    Writeln('Range 4');
end
else begin
    Writeln('Range 5');
end;
```

**Exercise 5.2.1 :** Is there a better way to solve this problem? Imagine if we had 100 different ranges – this would require a very long program. [HINT : a simple calculation on *mark* determines the range it falls into]. □

## 5.3 Repetition : WHILE and FOR Statements

The two forms of this control structure allow us to repeat a sequence of statements as often as we please.

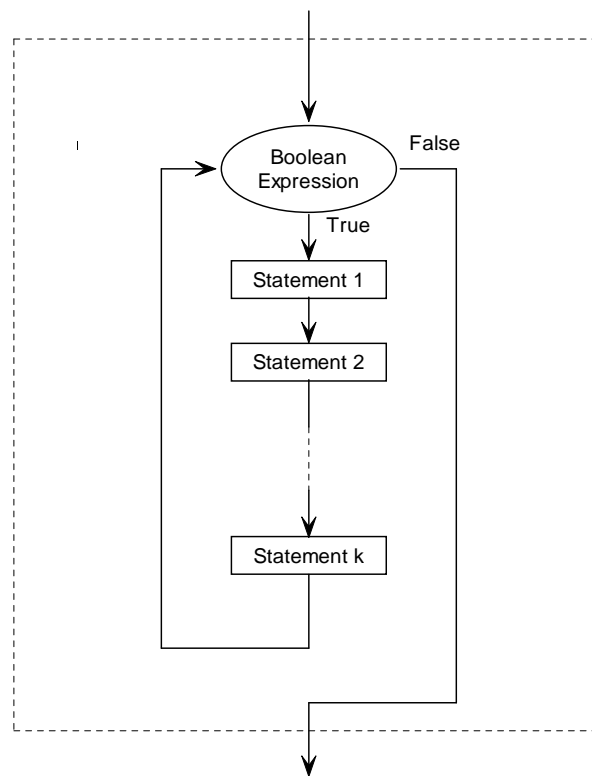
### 5.3.1 The WHILE Statement

The syntax of the WHILE form is as follows :

```
while <BOOLEAN expression> do begin
    <statement sequence>;
end;
```

The flow diagram for this control statement is shown in Figure 5.4

This control statement repeats the statement sequence as long as the boolean expression is true. When the boolean expression becomes false the control jumps to the first statement after the the **end** statement of the **while** -loop.



**Figure 5.5.** The WHILE Control Structure.

EXAMPLE : Write a program to find the sum and the sum of the squares of the first  $n$  natural numbers, i.e., we wish to calculate

$$S_1(n) = \sum_{i=1}^n i, \quad \text{and} \quad S_2(n) = \sum_{i=1}^n i^2.$$

The program should work for any  $n \geq 0$ .

The program must work for any  $n \geq 0$  so we don't know ahead of time how many natural numbers to add. This means we cannot use a big long assignment statement – we have to repeatedly add numbers until we have added  $n$  of them. This means we have to use a **while** - loop

```

PROGRAM Summation ;
{Calculates the sum and the sum of the squares}
{of the first n natural numbers. }
VAR
  S1 : INTEGER;
  S2 : INTEGER;
  n  : INTEGER;
  i  : INTEGER
BEGIN
  Write('Type in an integer number : ');
  Readln(n);
  S1 := 0;
  S2 := 0;
  i := 1;
  WHILE (i <= n) DO BEGIN
    S1 := S1 + i;
    S2 := S2 + i*i;
    i := i + 1;
  END; { while }
  Writeln('The sum of the first ', n,
    ' numbers is ', S1);
  Writeln('The sum of the squares',
    ' of the first ', n,
    ' numbers is ', S2);
END.{PROG Summation}

```

In this program we have used the mathematical fact that

$$S_1(i) = S_1(i-1) + i \quad \text{and} \quad S_2(i) = S_2(i-1) + i^2.$$

This translates to Pascal as

```
S1 := S1 + i; and S2 := S2 + i*i;
```

In the program **S1** on the left of the **:=** operator corresponds to  $S_1(i)$  and **S1** on the right of the **:=** operator corresponds to  $S_1(i-1)$ . Likewise for **S2**. □

**Exercise 5.3.1** : Work through the program above and make sure that it does what it claims to do in the comment line. Try **n = 4** and **6**. Does the program give the correct result for  $n = 0$ ? What happens if  $n < 0$ ? Check it out. □

EXAMPLE : The following program calculates the cost of an order where the manufacturer gives *price breaks*. That is, the price is 1.50/unit for 1–1000 units, 1.30/unit for 1001–5000 units, and 1.00/unit for more than 5000 units. This a program that uses all the control structures we have seen so far. □

```
PROGRAM OrderCost ;
CONST
    UnitC1   = 1.50;
    UnitC2   = 1.30;
    UnitC3   = 1.00;
    Break1   = 1000;
    Break2   = 5000;
VAR
    Name      : STRING;
    NoUnits   : INTEGER;
    Answer    : CHAR;
    Cost      : REAL;
BEGIN
    Answer := 'Y';
    WHILE (Answer = 'Y') DO BEGIN
        Write('Type in name           : ');
        ReadLn(Name);
        Write('Type in number of units : ');
        ReadLn(NoUnits);
        IF (NoUnits <= Break1) THEN BEGIN
            Cost := UnitC1*NoUnits;
        END IF
        ELSE IF (TaxPay <= Band2) THEN BEGIN
            Cost := UnitC2*NoUnits;
        END
        ELSE BEGIN
            Cost := UnitC3*NoUnits;
        END;
        WriteLn('Name           : ', Name);
        WriteLn('Number of Units : ', NoUnits:10);
        WriteLn('Cost           : ', Cost:10:2);
        Write('Do you want to continue (Y/N)?');
        ReadLn(Answer);
    END;{WHILE}
END.
```

EXAMPLE : The following program calculates the tax where the rate is 10% for the first £5000 of taxable pay, 30% for the next £20000 and 50% for the anything above £20000. This a program that uses all the control structures we have seen so far. □

```
PROGRAM TaxPayCalc ;
CONST
    TRate1  = 0.1;
    TRate2  = 0.3;
    TRate3  = 0.5;
    Band1   = 5000;
    Band2   = 20000;
VAR
    Name      : STRING;
    GrossPay  : REAL;
    NetPay    : REAL;
    TFAllow   : REAL;
    TaxPay    : REAL;
    Tax1      : REAL;
    Tax2      : REAL;
    Tax3      : REAL;
    Tax       : REAL;
    Answer    : CHAR;
BEGIN
    Answer := 'Y';
    WHILE (Answer = 'Y') DO BEGIN
        Write('Type in name           : ');
        Readln(Name);
        Write('Type in gross pay       : ');
        Readln(GrossPay);
        Write('Type in tax-free allowance : ');
        Readln(TFAllow);
        TaxPay := GrossPay - TFAllow;
        IF (TaxPay <= Band1) THEN BEGIN
            Tax := TRate1*TaxPay;
        END IF
        ELSE IF (TaxPay <= Band2) THEN BEGIN
            Tax1 := TRate1*Band1;
            Tax2 := TRate2*(TaxPay - Band1);
            Tax  := Tax1 + Tax2;
        END
        ELSE BEGIN
            Tax1 := TRate1*Band1;
            Tax2 := TRate2*(Band2 - Band1);
            Tax3 := TRate3*(TaxPay - Band2);
            Tax  := Tax1 + Tax2 +Tax3;
        END;
        NetPay := GrossPay - Tax;
        WriteLn('Name      : ', Name);
        WriteLn('Gross Pay : ', GrossPay:10:2);
        WriteLn('Tax       : ', Tax:10:2);
        WriteLn('Net Pay  : ', NetPay:10:2);
        Write('Do you want to continue (Y/N)?');
        Readln(Answer);
    END;{WHILE}
END.
```

### 5.3.2 The FOR Statement

The FOR statement is a useful repetition control statement if we know ahead of time how many times we need to repeat a statement sequence.

The syntax of the **for** statement has two forms and are as follows :

```
for <variable>:= <start> to <finish> do begin
    <statement sequence>;
end;
```

```
for <variable>:=<start> downto <finish> do begin
    <statement sequence>;
end;
```

Let us look at some examples to understand the semantics of the **for** statement.

```
program SummationFOR ;
{ Calculates the sum and the sum of the squares of the first n natural numbers}
var
    S1 : INTEGER;
    S2 : INTEGER;
    n : INTEGER;
    i : INTEGER
begin
    Write('Type in an integer number : ');
    Readln(n);

    S1 := 0;
    S2 := 0;

    for i := 1 to n do begin
        S1 := S1 + i;
        S2 := S2 + i*i;
    end; {FOR}

    Writeln('The sum of the first ', n, ' numbers is ', S1);
    Writeln('The sum of the squares of the first ',
            n, ' numbers is ', S2);

end.
```

This program does exactly the same calculation as the previous **Summation** program, except here we don't need to initialize **i** to 1, and we don't need to increment **i** by 1 each time around the loop. What happens is this : The **for** statement initializes **i** to the **<start>** value 1; it then checks to see if **i <= <finish>**; it then executes the statement sequence, at the end of which it increases (increments) **i** by 1 and goes back to the **for** statement to perform the check **i <= <finish>**. In other words the variable **i** gets the values 1, 2, ..., **n** and the statement sequence is executed for each of these values.

Why do we need two repetition control statements? We don't. The **for** statement is there for convenience when we know in advance how many times we need to repeat the statement sequence. We can always get rid of a **for** statement by translating it into a **while** statement

**Translating a FOR into a WHILE Statement** We repeat here the **for** syntax :

```
for <variable>:=<start> to <finish> do begin
    <statement sequence>;
end;
```

The equivalent **while** syntax is

```
<variable> := <start>;
while <variable> <= <finish> do begin
    <statement sequence>;
    <variable> := <variable> + 1;
end;
```

```
for j := -2 to n do begin
    Writeln(j);
end; { FOR }
```

```
j := -2;
while j <= n do begin
    Writeln(j);
    j := j + 1;
end; { FOR }
```

```
for j := i+1 to i+n do begin
    Writeln(j);
end; { FOR }
```

Notice here that **<start>** and **<finish>** are expressions. This **for** -loop translates to

```
j := i+1
while j <= i+n do begin
    Writeln(j);
    j := j + 1;
end; { FOR }
```

Notice above that  $j$  is incremented by 1. This is true of all **for** -loops.

We also have **for** -loops that go down by one. Here is an example and its translation into a **while** -loop.

```

for j := n downto 1 do begin
    Writeln(j);
end; { FOR }

```

```

j := n;
while j >= 1 do begin
    Writeln(j);
    j := j - 1;
end; { FOR }

```

### Rules for Loops

1. <variable> must be INTEGER, CHAR, or BOOLEAN.
2. <variable> is incremented by 1 in a **to** loop and decremented by 1 in a **downto** loop.
3. <start> may be any expression of the types in 1.
4. <finish> may be any expression of the types in 1.
5. The boolean expression in the **while** statement must be altered by the statements in the loop.

**Loops within Loops - Nesting** Many important programs need to have loops within loops. A good example is sorting : most sorting programs have loops within loops. Another example are matrix calculations where we need to work on each element  $a_{ij}$  of an  $n \times n$  matrix  $A$ . To see how nested loops work let us look at two examples

EXAMPLE : *Bubble Sort*. We saw this algorithm in the first chapter. Its basic operation is as follows : pass through an array of numbers and swap every adjacent pair that are out of order. Repeat this until no adjacent pairs are out of order.

The following program requires an array of numbers which we haven't formally studied yet. For the moment we define an array as a set of contiguous memory boxes labelled with a single identifier where individual boxes are identified by the array identifier and an index. For example, if  $A$  identifies the array then  $A[i]$  identifies the  $i$ th box of the array. Here is the program.

```

PROGRAM BubbleSort;
  {Sorts an array of integers x}
VAR
  i, n,
  pass : INTEGER;
  temp : INTEGER;
  x : ARRAY[1..1000] OF INTEGER;
BEGIN
  { Read values into n and the array x }
  FOR pass := 1 TO n
    FOR i := 1 to n - pass DO BEGIN
      IF x[i] > x[i + 1] THEN BEGIN
        temp := x[i];
        x[i] := x[i + 1];
        x[i + 1] := temp
      END; { IF }
    END; { FOR i }
  END: { FOR pass }
END. {PROG BUBBLESORT}

```

□

Nested loops are very important in processing matrices, called *two-dimensional arrays* in Pascal and other languages. EXAMPLE : *Matrix Addition*. The following program fragment adds two matrices, where  $C = A + B$  and

$$c_{ij} = a_{ij} + b_{ij}.$$

```

VAR
  i,j,n : INTEGER;
  a,b,c : ARRAY[1..100, 1..100] OF REAL;
BEGIN
  { Read in values for n, a[1..n, 1..n] and b[1..n, 1..n] }
  FOR i := 1 TO n DO BEGIN
    FOR j := 1 TO n DO BEGIN
      c[i,j] := a[i,j] + b[i,j];
    END; { FOR j }
  END; { FOR i }
END.

```

□

In the two examples above we used nested **for** -loops. We can also mix **while** and **for** loops.

EXAMPLE : *A Better Bubble Sort.* Here is a slicker version of `BubbleSort` that stops as soon as it finds that no adjacent pairs of numbers are out of order. To do this it sets a boolean variable `Swap` to the value `true` each time it swaps two adjacent values. If, at the start of any pass it finds `Swap = false` it knows that all boxes in the array are in order and stops.

```

PROGRAM BetterBubbleSort;
  {Sorts an array of integers x}
  VAR
    i, n,
    pass : INTEGER;
    temp : INTEGER;
    Swap : BOOLEAN;
    x : ARRAY[1..1000] OF INTEGER;
  BEGIN
    { Read values into n and the array x }
    Swap := true;
    pass := 0;
    WHILE Swap DO BEGIN
      pass := pass + 1;
      FOR i := 1 to n - pass DO BEGIN
        IF x[i] > x[i + 1] THEN BEGIN
          temp := x[i];
          x[i] := x[i + 1];
          x[i + 1] := temp;
          Swap := true;
        END
        ELSE BEGIN
          Swap := false;
        END { IF }
      END; { FOR i }
    END: { WHILE }
  END. {PROG BetterBubbleSort }

```

□

**Exercise 5.3.1 :** Go through the program above using the array of size 12 shown below. Make sure to keep a record of the value of the variable `Swap` each time you go through the loops. How many passes does the program make through the array. How many passes does it make through the following array of numbers.

10 12 13 20 23 34 28 42 42 67

1	2	3	4	5	6	7	8	9	10	11	12
22	14	11	6	74	24	12	81	31	10	53	7

**Figure 5.6.** An Array of Integers.

□

**Exercise 5.3.1 :** What does the following program do. Yes, it's obvious it writes out either **true** or **false**, but why is it writing out either of these values? [You have seen an algorithmic version of this program earlier].

```
PROGRAM WhatIsThis;
{Sorts an array of integers x}
VAR
  i, n : INTEGER;
  answer : BOOLEAN;
  x : ARRAY[1..1000] OF INTEGER;
BEGIN
  { Read values into n and the array x }
  FOR i := 1 to n-1 DO BEGIN
    IF x[i] > x[i+1] THEN BEGIN
      answer := false;
    END
    ELSE BEGIN
      answer := true;
    END { IF }
  END; { FOR i }
  Writeln('The answer is : ', answer);
END. {PROG WhatIsThis }
```

□

**Exercise 5.3.1 :** What does the following program fragment do? If you can't figure it out, write a program that incorporates this code and run it.

```
FOR i := 1 TO n DO BEGIN
  FOR j := 1 TO n DO BEGIN
    FOR k := 1 TO n DO BEGIN
      Writeln(i:5, j:5, k:5);
    END; { FOR k }
  END; { FOR j }
END; { FOR i }
```

□