Introduction

PASCAL is a programming language named after the 17th century mathematican Blaise Pascal. Pascal

- provides a teaching language that highlights concepts common to all computer languages
- standardises the language in such a way that it makes programs easy to write

Strict rules make it difficult for the programmer to write bad code!

Basic format of every Pascal program

Every Pascal program has the same essential format, which is illustrated below,

```
program TITLE ( input, output );
begin
  program statements;
  program statement
end.
```

program is the first word of all Pascal programs. It is a **keyword** (Keywords are reserved, ie, you cannot use keywords to describe variables).

TITLE is the name the programmer gives to the Pascal program being written.
It is an **identifier**. Identifiers begin with a letter, then followed by any digit, letter or the underscore character (_).

Question time. Which of the following are valid Pascal identifiers?

```
birthday      Too_hot?        First_Initial
grade         1stprogram      down.to.earth
see you       OldName         case
```

Answer:

Question time. Which of the following are valid Pascal identifiers?

```
birthday      Too_hot?        First_Initial
grade         1stprogram      down.to.earth
see you       OldName         case
```

birthday
First_Initial
grade
OldName

(input, output) states what the program will do, ie, input and/or output data. Data is inputted from the keyboard, and outputted to the console screen.

begin defines the starting point of the program, and provides a means of grouping statements together (ie all statements between a *begin* and *end* are considered part of the same group or block).

program statements are commands or instructions to the computer which perform various tasks.

end. This must always be the final statement of a Pascal program.

ALL PROGRAM STATEMENTS AND LINES ARE TERMINATED WITH A SEMI-COLON, EXCEPT THE BEGIN AND END KEYWORDS. PROGRAM STATEMENTS PRECEEDING AN END STATEMENT DO NOT REQUIRE A SEMI-COLON.

A SIMPLE Pascal Program

Write a program to print the words 'Hello. How are you?' on the console screen.

```
program MYFIRST (output);
begin
          writeln('Hello. How are you?')
end.
```

The keyword *writeln* writes text to the console screen. The text to be displayed is written inside **single quotes**. After printing the text inside the single quotes, the cursor is positioned to the beginning of the next line.

To print a single quote as part of the text, then use two quotes, eg,

```
program TWOQUOTES (output);
begin
          writeln('Hello there. I''m fine.')
end.
```

```
program output is;
Hello there. I'm fine.
_
```

```
Note the underscore character represents the position of the cursor
```

write versus *writeln*

The *write* statement leaves the cursor at the end of the current ouput, rather than going to a new line. By replacing the above program with a *write* statement, the result is,

```
program TWOQUOTES (output);
begin
          write('Hello there. I''m fine.')
end.
```

```
program output is;
Hello there. I'm fine._
```

```
Note the underscore character represents the position of the cursor
```

Write a program to print the the following words on the console screen.

```
         Hello. How are you?
         I'm just fine.
```

Answer:

```
Hello. How are you?
         I'm just fine.
```

```
program MYSECOND (output);
begin
        writeln('Hello. How are you?');
        writeln('I''m just fine.');
end.
```

Comments
are inserted into Pascal programs by enclosing the comment within { and } braces. Comments are ignored by the computer, but are helpful to explain how the program works to other programmers.

A Pascal program with comments

```
program DEMOPROG (output);
begin
       write('Hello there.');
       {the write statement does not set the cursor
        to the beginning of the next line. }
       writeln('This is getting boring.')
       { This is printed on the same line as Hello
         there, but now the cursor moves to the beginning
         of the next line, because this time we used writeln
         instead of write  }
end.
```

SELF TEST

1: Comments are opened with ___ and closed with ___

2: The end. statement signifies the ______ of a Pascal program.

3: The write statement sets the cursor ________________________________

____

4: Using the writeln statement, the cursor is positioned at __________

____

5: When printing text using write or writeln, the text is enclosed

using ____

6: To print a single quote using write or writeln, use ______________

7: Each program statement is terminated with a _______________________

8: If the program is required to input data from the keyboard, the first line of the program reads

____

9: If the program is required to input data from the keyboard, and also output data to the console screen, the first line of the program reads

____

Answers

SELF TEST

1: Comments are opened with { and closed with }

2: The end. statement signifies the **end** of a Pascal program.

3: The write statement sets the cursor **at the end of the current text**

4: Using the writeln statement, the cursor is positioned at **the beginning of the next line**

5: When printing text using write or writeln, the text is enclosed using **single quotes**

6: To print a single quote using write or writeln, use **two single quotes one after the after, ''**

7: Each program statement is terminated with a **semi-colon**

8: If the program is required to input data from the keyboard, the first line of the program reads

```
program name (input);
```

9: If the program is required to input data from the keyboard, and also output data to the console screen, the first line of the program reads

```
program name (input, output);
```

Pascal Variables

Variables store values and information. They allow programs to perform calculations and store data for later retrieval. Variables store numbers, names, text messages, etc.

Pascal supports **FOUR** standard variable types, which are

- integer
- char
- boolean
- real

integer

Integer variables store whole numbers, ie, no decimal places. Examples of integer variables are,

```
        34        6458     -90    0      1112
```

char

Character variables hold any valid character which is typed from the keyboard, ie digits, letters, punctuation, special symbols etc. Examples of characters are,

```
   XYZ    0ABC    SAM_SAID.GET;LOST  [ ] { } = + \ | %   ( ) * $
```

boolean

Boolean variables, also called logical variables, can only have one of two possible states, true or false.

real

Real variables are positive or negative numbers which include decimal places. Examples are,

```
     34.265   -3.55  0.0   35.997E+11
```

Here, the symbol **E** stands for 'times 10 to the power of'

Types integer, char and boolean are called ORDINAL types

Using Pascal VARIABLES in a program

The basic format for declaring variables is,

```
        var name : type;
```

where *name* is the name of the variable being declared, and *type* is one of the recognised data types for pascal.

Before any variables are used, they are declared (made known to the program). This occurs after the program heading, and before the keyword *begin*, eg,

```
        program VARIABLESINTRO (output);
           var  number1: integer;
                number2: integer;
                number3: integer;
        begin
                number1 := 34;  { this makes number1 equal to 34 }
                number2 := 43;  { this makes number2 equal to 43 }
                number3 := number1 + number2;
                writeln( number1, ' + ', number2, ' =  ', number3 )
        end.
```

The above program declares three integers, number1, number2 and number3.

To declare a variable, first write the variable name, followed by a colon, then the variable type (int real etc). Variables of the same type can be declared on the same line, ie, the declaration of the three integers in the previous program

```
var number1: integer;
    number2: integer;
    number3:integer;
```

could've been declared as follows,

```
var number1, number2, number3 : integer;
```

Each variable is seperated by a comma, the colon signifies there is no more variable names, then follows the data type to which the variables belong, and finally the trusty semi-colon to mark the end of the line.

Some example of variable declarations

```
program VARIABLESINTRO2 (output);
   var  number1: integer;
        letter : char;
        money  : real;
begin
        number1 := 34;
        letter  := 'Z';
        money   := 32.345;
        writeln( "number1 is ", number1 );
        writeln( "letter is  ", letter );
        writeln( "money is   ", money )
end.
```

SELF TEST
Are the following valid variable declarations?

```
var day, month : integer;
    time       : real;

var time : real;
    day  : integer;
    month: integer;
```

Classify each of the following according to the four basic data types.

```
34.276   ____________        -37     __________________

H        ____________        <       __________________

dd       ____________        5.09E+27 _______________

0        ____________        0.0     __________________
```

Answer:
SELF TEST
Are the following valid variable declarations?

```
var day, month : integer;
    time       : real;

var time : real;
    day  : integer;
    month: integer;
```

They are both identical and also valid!

Classify each of the following according to the four basic data types.

34.276	Real	-37	Integer
H	Character	<	Character
dd	Character	5.09E+27	Real
0	Integer	0.0	Real

VARIABLE NAMES

Variable names are a maximum of 32 alphanumeric characters. Some Pascal versions only recognise the first eight characters. The first letter of the data name must be ALPHABETIC (ie A to Z). Lowercase characters (a to z) are treated as uppercase. Examples of variable names are,

```
RATE_OF_PAY      HOURS_WORKED   B41   X   y   Home_score
```

Give variables meaningful names, which will help to make the program easier to read and follow. This simplifies the task of error correction.

ASSIGNING VALUES TO VARIABLES

Having declared a variable, you often want to make it equal to some value. In pascal, the special operator

```
:=
```

provides a means of assigning a value to a variable. The following portion of code, which appeared earlier, illustrates this.

```
var number1, number2, number3 : integer;
begin
       number1 := 43;   { make number1 equal to 43 decimal }
       number2 := 34;   { make number2 equal to 34 decimal }
       number3 := number1 + number2;  { number3 equals 77  }
```

SELF TEST

1. Declare an integer called sum

2. Declare a character called letter

3. Declare a variable called money which can be used to hold currency

4. Declare a variable called arctan which will hold scientific notation values (+e)

5. Declare an integer variable called total and initialise it to zero.

6. Declare a variable called loop, which can hold any integer value.

Answer:

SELF TEST: ANSWERS

1. Declare an integer called sum

```
        sum : integer;
```

2. Declare a character called letter

```
        letter : char;
```

3. Declare a variable called money which can be used to hold currency

```
        money : real;
```

4. Declare an integer variable called total and initialise it to zero

```
        total : integer;
        total := 0;
```

5. Declare a variable called loop, which can hold any integer value

```
        loop : integer;
```

ARITHMETIC STATEMENTS

The following symbols represent the arithmetic operators, ie, use them when you wish to perform calculations.

+	Addition
-	Subtraction
*	Multiplication
/	Division

Addition Example

```
program Add (output);
var number1, number2, result : integer;
begin
        number1 := 10;
        number2 := 20;
        result := number1 + number2;
        writeln(number1, " plus ", number2, " is ", result )
end.
```

Subtraction Example

```
program Subtract (output);
var number1, number2, result : integer;
begin
        number1 := 15;
        number2 := 2;
        result := number1 - number2;
        writeln(number1, " minus ", number2, " is ", result )
end.
```

Multiplication Example

```
program Multiply (output);
var number1, number2, result : integer;
begin
        number1 := 10;
        number2 := 20;
        result := number1 * number2;
        writeln(number1, " multiplied by ", number2, " is ", result )
end.
```

Division Example

```
program Divide (output);
var number1, number2, result : integer;
begin
        number1 := 20;
        number2 := 10;
        result := number1 / number2;
        writeln(number1, " divided by ", number2, " is ", result )
end.
```

SELF TEST

The following program contains a few errors. Identify each error (there are seven), and show the correct version on the right.

```
progam TEST (output)                      _________________________________

var    number1, number2; integer;         _________________________________

begin;                                    _________________________________

       number1 = 24;                      _________________________________

       number2 := number1 * 4;            _________________________________

       writeln('Help )                    _________________________________

end                                       _________________________________
```

Answers:

SELF TEST

The following program contains a few errors. Identify each error (there are seven), and show the correct version on the right.

```
progam TEST (output)
var    number1, number2; integer;
begin;
       number1 = 24;
       number2 := number1 * 4;
       writeln('Help )
end
```

```
program TEST (output);
var    number1, number2 : integer;
begin
       number1 := 24;
       number2 := number1 * 4;
       writeln('Help' )
end.
```

DISPLAYING THE VALUE OR CONTENTS OF VARIABLES

The *write* or *writeln* statement displays the value of variables on the console screen. To print text, enclose inside single quotes. To display the value of a variable, do NOT enclose using single quotes, eg, the following program displays the content of each of the variables declared.

```
program DISPLAYVARIABLES (output);
var     number1 : integer;
         letter  : char;
         money   : real;
begin
         number1 := 23;
         letter  := 'W';
         money   := 23.73;
         writeln('number1 = ', number1 );
         writeln('letter  = ', letter  );
         writeln('money   = ', money   )
end.
```

The display output from the above program will be,

```
number1 = 23
letter  = W
money   = 2.3730000000E+01
```

SELF TEST

Each of the following expressions is wrong. Rewrite each using correct Pascal, in the space provided.

```
Firstletter := A;                  ____________________________________

StartCount:= Initial := 0;         ____________________________________

Taxrate := 5%;                     ____________________________________

Total := 5 plus 7;                 ____________________________________

Effeciency := .35;                 ____________________________________
```

Answer:

SELF TEST

Each of the following expressions is wrong. Rewrite each using correct Pascal, in the space provided.

```
Firstletter := A;                      Firstletter := 'A';

StartCount := Initial := 0;            StartCount := 0;
                                       Initial := 0;

Taxrate := 5%;                         Taxrate := 0.05;

Total := 5 plus 7;                     Total := 5 + 7;

Effeciency := .35;                     Effeciency := 0.35;
```

CLASS EXERCISE

What is displayed by the following program.

```
program EXERCISE1 (output);
var     a, b : integer;
        c    : real;
begin
        a := 1;     b := 5;       c := 1.20;
        writeln('A = ', a + 3 );
        writeln('B = ', b - 2 );
        writeln('C = ', c / 2 )
end.
```

Answer:

CLASS EXERCISE

What is displayed by the following program.

```
program EXERCISE1 (output);
var     a, b : integer;
        c    : real;
begin
        a := 1;     b := 5;       c := 1.20;
        writeln('A = ', a + 3 );
        writeln('B = ', b - 2 );
        writeln('C = ', c / 2 )
end.
```

```
Class Exercise .. program display is
A = 4
B = 3
C = 6.0000000000E-01
```

PROGRAM ONE

You are to write a program which calculates and prints on the screen, the time required to travel 3000 miles at a speed of 500 mph.

PROGRAM TWO

Write a program to calculate the gross pay for a worker named FRED given that FRED worked 40 hours at $2.90 per hour.

Answer:

PROGRAM ONE

You are to write a program which calculates and prints on the screen, the time required to travel 3000 miles at a speed of 500 mph.

```
program PROG1 (output);
var  Time, Distance, Speed : real;
begin
     Distance := 3000;
     Speed    := 500;
     Time := Distance / Speed;
     writeln('It takes ',Time,' hours.')
end.
```

PROGRAM TWO

Write a program to calculate the gross pay for a worker named FRED given that FRED worked 40 hours at $2.90 per hour.

```
program PROG2 (output);
var  grosspay, hoursworked, hourlyrate : real;
begin
     hoursworked := 40;
     hourlyrate  := 2.90;
     grosspay    := hoursworked * hourlyrate;
     writeln('FRED''s gross pay is $', grosspay )
end.
```

GETTING INFORMATION/DATA FROM THE KEYBOARD INTO A PROGRAM

It is convenient to accept data whilst a program is running. The *read* and *readln* statements allow you to read values and characters from the keyboard, placing them directly into specified variables.

The program which follows reads two numbers from the keyboard, assigns them to the specified variables, then prints them to the console screen.

```
program READDEMO (input, output);
var     numb1, numb2 : integer;
begin
        writeln('Please enter two numbers separated by a space');
        read( numb1 );
        read( numb2 );
        writeln;
        writeln('Numb1 is ', numb1 , '   Numb2 is ', numb2 )
end.
```

When run, the program will display the message

```
Please enter two numbers separated by a space
```

then wait for you to enter in the two numbers. If you typed the two numbers, then pressed the return key, eg,

```
237  64
```

then the program will accept the two numbers, assign the value 237 to *numb1* and the value 64 to *numb2*, then continue and finally print

```
Numb1 is 237  Numb2 is 64
```

Differences between READ and READLN

The *readln* statement discards all other values on the same line, but *read* does not. In the previous program, replacing the *read* statements with *readln* and using the same input, the program would assign 237 to *numb1*, discard the value 64, and wait for the user to enter in another value which it would then assign to *numb2*.

The is read as a blank by *read*, and ignored by *readln*.

SELF TEST on READ

Assuming that we made the following declaration

```
var  C1, C2, C3, C4, C5, C6 : char;
```

and that the user types

```
ABCDE
```

then what would each of the following statements assign to the various variables,

```
read( C1 );                        C1 = __
read( C2 ); read( C3 );            C2 = __  C3 = __
read( C4, C5, C6 );                C4 = __  C5 = __  C6 = __
```

Answer:

SELF TEST on READ

Assuming that we made the following declaration

```
 var  C1, C2, C3, C4, C5, C6 : char;
```

and that the user types

```
ABCDE
```

then what would each of the following statements assign to the various variables,

```
read( C1 );                        C1 = __
read( C2 ); read( C3 );            C2 = __   C3 = __
read( C4, C5, C6 );                C4 = __   C5 = __   C6 = __

C1 = A
C2 = B   C3 = C
C4 = D   C5 = E    C6 =
```

SELF TEST on READLN

Assuming that we made the following declaration

```
 var  C1, C2, C3, C4, C5, C6 : char;
```

and that the user types

```
ABCDE
```

FOR EACH LINE, then what would each of the following statements assign to the various variables,

```
readln( C1 );                          C1 = __
readln( C2 ); readln( C3 );            C2 = __   C3 = __
readln( C4, C5, C6 );                  C4 = __   C5 = __   C6 = __
readln;                                _________________________
```

Answer:

SELF TEST on READLN

Assuming that we made the following declaration

```
 var  C1, C2, C3, C4, C5, C6 : char;
```

and that the user types

```
ABCDE
```

FOR EACH LINE, then what would each of the following statements assign to the various variables,

```
readln( C1 );                          C1 = __
readln( C2 ); readln( C3 );            C2 = __   C3 = __
readln( C4, C5, C6 );                  C4 = __   C5 = __   C6 = __
readln;                                _________________________


C1 = A
C2 = A   C3 = Waits for new input line
C4 = A   C5 = B    C6 = C
```

SELF TEST...Match the inputs and outputs for the following.....

```
program READCHARACTERS (input, output);
var     C1, C2, C3, C4, C5, C6 : char;
begin
        readln( C1, C2, C3, C4, C5, C6 );
        writeln(C1, C2, C3, C4, C5, C6 )
end.

Inputs                            Outputs ( spaces shown as _ )

a) Hi there                       a) A1_B2_

b) Hi                             b) 57_4_3
   there
c) 694                            c) Hi_the
   827
d) 57 4                           d) Hi_the
   329
e) A1                             e) 694_82
   B2 C3
```

Answer:

SELF TEST...Match the inputs and outputs for the following.....

```
program READCHARACTERS (input, output);
var     C1, C2, C3, C4, C5, C6 : char;
begin
        readln( C1, C2, C3, C4, C5, C6 );
        writeln(C1, C2, C3, C4, C5, C6 )
end.

Inputs                            Outputs ( spaces shown as _ )

a) Hi there                       a) A1_B2_

b) Hi                             b) 57_4_3
   there
c) 694                            c) Hi_the
   827
d) 57 4                           d) Hi_the
   329
e) A1                             e) 694_82
   B2 C3
```

```
a) Hi there              c) Hi_ther
b) Hi                    d) Hi_ther
   there
c) 694                   e) 694_82
   827
d) 57 4                  b) 57_4_3
   329
e) A1                    a) A1_B2_
   B2 C3
```

PROGRAM THREE

Ohm's law states that the voltage (V) in a circuit is equal to the current flowing in amperes (I) multiplied by the resistance in the ciruit (R) [ie, E = I * R]. Write a program to enter in the values of resistance and current, displaying the voltage which would exist.

Answer:

PROGRAM THREE

Ohm's law states that the voltage (V) in a circuit is equal to the current flowing in amperes (I) multiplied by the resistance in the ciruit (R) [ie, E = I * R]. Write a program to enter in the values of resistance and current, displaying the voltage which would exist.

```
program OHMSLAW (input, output);
var resistance, current, volts : real;
begin
    writeln('Please enter resistance value');
    readln( resistance );
    writeln('Please enter current value');
    readln( current );
    volts := current * resistance ;
    writeln('The voltage is ', volts )
end.
```

PROGRAM FOUR

Write a program which inputs two resistance values, and then displays their sum value when placed in series and then in parallel. [The total series resistance is R1 + R2, whilst the parallel resistance is (R1 * R2) / (R1 + R2)]

SPECIFYING THE DISPLAY FORMAT FOR THE OUTPUT OF VARIABLES

When variables are displayed, our version of Pascal assigns a specified number of character spaces (called a field width) to display them. The field widths for the various data types are,

INTEGER	-	Number of digits + 1 { or +2 if negative }
CHAR	-	1 for each character
REAL	-	12
BOOLEAN	-	4 if true, 5 if false

Often, the allotted field size is too big for the majority of display output. Pascal provides a way in which the programmer can specify the field size for each output.

```
writeln('WOW':10,'MOM!':10,'Hi there.');
```

The display output will be as follows,

```
WOW.......MOM!......Hi there.          ... indicates a space.
```

Note that to specify the field width of text or a particular variable, use a colon (:) followed by the field size.

```
'text string':fieldsize, variable:fieldsize
```

INTEGER DIVISION

There is a special operator, **DIV**, used when you wish to divide one integer by another (ie, you can't use /). The following program demonstrates this,

```
program INTEGER_DIVISION (output);
var     number1, number2, number3 : integer;
begin
       number1 := 4;
       number2 := 8;
       number3 := number2 DIV number1;
       writeln( number2:2,' divided by ',number1:2,' is ',number3:2)
end.

Sample Output
8 divided by  4 is  2
```

MODULUS

The MOD keyword means MODULUS, ie, it returns the remainder when one number is divided by another,

```
The modulus of 20 DIV 5 is 0
The modulus of 21 DIV 5 is 1
```

```
program MODULUS (output);
var     number1, number2, number3 : integer;
begin
       number1 := 3;
       number2 := 10;
       number3 := number2 MOD number1;
       writeln( number2:2,' modulus ',number1:2,' is ',number3:2)
end.

Sample Output
10 modulus  3 is  1
```

SELF TEST

1. Write a Pascal statement which sums the two integer variables *digit* and *value* into the variable *total*

2. Write a Pascal statement which subtracts the value 10 from the variable *loop*, leaving the result in the variable *sum*

3. Write a Pascal statement to display the value of the integer variable *total*

4. Write a Pascal statement to read in a character value into the variable *letter*

5. Write a Pascal statement to display the value of the real variable *small_value* using a field width of three places

Answers:

SELF TEST

1. Write a Pascal statement which sums the two integer variables *digit* and *value* into the variable *total*

```
	total := digit + value;
```

2. Write a Pascal statement which subtracts the value 10 from the variable *loop*, leaving the result in the variable *sum*

```
	sum := loop - 10;
```

3. Write a Pascal statement to display the value of the integer variable *total*

```
	writeln( total );
```

4. Write a Pascal statement to read in a character value into the variable *letter*

```
	readln( letter );
```

5. Write a Pascal statement to display the value of the real variable *small_value* using a field width of three places

```
	writeln( small_value:3 );
```

MAKING DECISIONS

Most programs need to make decisions. There are several statements available in the Pascal language for this. The **IF** statement is one of the them. The **RELATIONAL OPERATORS**, listed below, allow the programmer to test various variables against other variables or values.

```
	=   Equal to
	>   Greater than
	<   Less than
	<>  Not equal to
	<=  Less than or equal to
	>=  Greater than or equal to
```

The format for the **IF THEN** Pascal statement is,

```
	if  condition_is_true  then
	        execute_this_program_statement;
```

The condition (ie, A < 5) is evaluated to see if it's true. When the condition is true, the program statement will be executed. If the condition is not true, then the program statement following the keyword **then** will be ignored.

```
	program IF_DEMO (input, output);  {Program demonstrating IF THEN statement}
	var     number, guess : integer;
	begin
	        number := 2;
	        writeln('Guess a number between 1 and 10');
	        readln( guess );
	        if number = guess then  writeln('You guessed correctly. Good on you!');
	        if number <> guess  then  writeln('Sorry, you guessed wrong.')
	end.
```

Executing more than one statement as part of an IF

To execute more than one program statement when an **if** statement is true, the program statements are grouped using the *begin* and *end* keywords. Whether a semi-colon follows the *end* keyword depends upon what comes after it. When followed by another *end* or *end.* then it no semi-colon, eg,

```
program IF_GROUP1 (input, output);
var     number, guess : integer;
begin
        number := 2;
        writeln('Guess a number between 1 and 10');
        readln( guess );
        if number = guess then
        begin
            writeln('Lucky you. It was the correct answer.');
            writeln('You are just too smart.')
        end;
        if number <> guess then   writeln('Sorry, you guessed wrong.')
end.
```

```
program IF_GROUP2 (input, output);
var     number, guess : integer;
begin
        number := 2;
        writeln('Guess a number between 1 and 10');
        readln( guess );
        if number = guess then
        begin
           writeln('Lucky you. It was the correct answer.');
           writeln('You are just too smart.')
        end
end.
```

IF THEN ELSE

The **IF** statement can also include an **ELSE** statement, which specifies the statement (or block or group of statements) to be executed when the condition associated with the **IF** statement is false. Rewriting the previous program using an IF THEN ELSE statement,

```
{ Program example demonstrating IF THEN ELSE statement }
program IF_ELSE_DEMO (input, output);
var     number, guess : integer;
begin
        number := 2;
        writeln('Guess a number between 1 and 10');
        readln( guess );
        if number = guess  then
           writeln('You guessed correctly. Good on you!')
        else
           writeln('Sorry, you guessed wrong.')
end.
```

There are times when you want to execute more than one statement when a condition is true (or false for that matter). Pascal makes provison for this by allowing you to group blocks of code together by the use of the **begin** and **end** keywords. Consider the following portion of code,

```
                                               writeln('You guessed correctly. Good on
if number = guess  then                       you!');
 begin
 writeln('It may have been a lucky guess
though')
 end      {no semi-colon if followed by
an else }
 else
  begin
  writeln('Sorry, you guessed wrong.');
  writeln('Better luck next time')
 end;     {semi-colon depends on next
keyword }
```

CLASS EXERCISE

What is displayed when the following program is executed?

```
program IF_THEN_ELSE_TEST (output);
var     a, b, c, d : integer;
begin
    a := 5;  b := 3;  c := 99;  d := 5;
 if  a > 6  then  writeln('A');
 if  a > b  then  writeln('B');
                if  b = c  then
                begin
                  writeln('C');
                  writeln('D')
                end;
 if  b <> c  then writeln('E') else
writeln('F');
if  a >= c  then writeln('G') else
writeln('H');
                if  a <= d  then
                begin
                  writeln('I');
                  writeln('J')
                end
        end.
```

Answer:

CLASS EXERCISE

What is displayed when the following program is executed?

```
program IF_THEN_ELSE_TEST (output);
var     a, b, c, d : integer;
begin
   a := 5;  b := 3;  c := 99;  d := 5;
  if  a > 6  then  writeln('A');
 if  a > b  then  writeln('B');
                if  b = c  then
                begin
                  writeln('C');
                  writeln('D')
                end;
 if  b <> c  then writeln('E') else
writeln('F');
 if  a >= c  then writeln('G') else
writeln('H');
                if  a <= d  then
                begin
                  writeln('I');
                  writeln('J')
                end
        end.
```

```
Class Exercise .. Program output of
IF_THEN_ELSE_TEST is...
        B
        E
        H
        I
        J
```

CONSTANTS

When writing programs, it is desirable to use values which do not change during the programs execution. An example would be the value of PI, 3.141592654

In a program required to calculate the circumference of several circles, it would be simpler to write the words PI, instead of its value 3.14. Pascal provides CONSTANTS to implement this.

To declare a constant, the keyword **const** is used, followed by the name of the constant, an equals sign, the constants value, and then a semi-colon, eg,

```
        const PI = 3.141592654;
```

From now on, in the Pascal program, you use PI. When the program is compiled, the compiler replaces every occurrence of the word PI with its actual value.

Thus, constants provide a short hand means of writing values, and help to make programs easier to read. The following program demonstrates the use of constants.

```
        program CIRCUMFERENCE (input,output);
        const   PI = 3.141592654;
        var     Circumfer, Diameter : real;
```

```
begin
        writeln('Enter the diameter of the circle');
        readln(Diameter);
        Circumfer := PI * Diameter;
        writeln('The circles circumference is ',Circumfer)
end.
```

SPECIFYING THE NUMBER OF DECIMAL PLACES FOR DISPLAYING REALS

The following change to the above program will print out the circumference using a fieldwidth of ten, and two decimal places.

```
writeln('The circles circumference is ',Circumference:10:2);
```

PROGRAM EIGHT

Write a program which inputs the ordinary time and overtime worked, calculating the gross pay. The rate is 4.20 per hour, and overtime is time and a half.

Answer:

PROGRAM EIGHT

Write a program which inputs the ordinary time and overtime worked, calculating the gross pay. The rate is 4.20 per hour, and overtime is time and a half.

```
program PROG8 ( input, output );
var  grosspay, ordinary_time, hourlyrate, overtime, ot_rate: real;

begin
     hourlyrate := 4.20;
     ot_rate    := hourlyrate * 1.5;
     writeln('Please enter the number of hours worked');
     readln( ordinary_time );
     writeln('Please enter the number of overtime hours');
     readln( overtime );
     grosspay := (ordinary_time * hourlyrate) + (overtime * ot_rate);
     writeln('The gross pay is $', grosspay:5:2 )
end.
```

SELF TEST

1. Which of the following is an invalid Pascal relational operator

```
==
<>
<
>
```

2. Write a Pascal statement which compares the integer variable *sum* to the constant value 10, and if it is the same prints the string "Good guess"

3. Write a Pascal statement which compares the character variable *letter* to the character variable *chinput*, and if it is not the same, prints the value of *letter*

4. Write a Pascal statement to compare the character variable *letter* to the character constant 'A', and if less, prints the text string "Too low", otherwise print the text string "Too high"

5. Write a Pascal statement to display the text string "Valve open", if the variable *waterflow* is equal to 1, AND the variable *outputvalue* is equal to 0

6. Write a Pascal statement which declares a constant called *MAXSIZE* with a value of 80

7. Write a Pascal statement which will display the value of the real variable *degrees* using a fieldwidth of 5 with three decimal places

Answers:

SELF TEST

1. Which of the following is an invalid Pascal relational operator

```
        ==
        <>
        <
        >
```

2. Write a Pascal statement which compares the integer variable *sum* to the constant value 10, and if it is the same prints the string "Good guess"

```
        if  sum = 10  then  writeln('Good guess');
```

3. Write a Pascal statement which compares the character variable *letter* to the character variable *chinput*, and if it is not the same, prints the value of *letter*

```
        if  letter <> chinput  then  writeln( letter );
```

4. Write a Pascal statement to compare the character variable *letter* to the character constant 'A', and if less, prints the text string "Too low", otherwise print the text string "Too high"

```
        if  letter  <  'A' then  writeln('Too low')
        else  writeln('Too high');
```

5. Write a Pascal statement to display the text string "Valve open", if the variable *waterflow* is equal to 1, AND the variable *outputvalue* is equal to 0

```
        if  (waterflow = 1)  AND  (outputvalue = 0) then  writeln('Valve open');
```

6. Write a Pascal statement which declares a constant called *MAXSIZE* with a value of 80

```
        const  MAXSIZE  =  80;
```

7. Write a Pascal statement which will display the value of the real variable *degrees* using a fieldwidth of 5 with three decimal places

```
        writeln('Degrees = ', degrees:5:3 );
```

LOOPS

The most common loop in Pascal is the FOR loop. The statement inside the for block is executed a number of times depending on the control condition. The format's for the FOR command is,

```
FOR var_name := initial_value TO final_value DO  program_statement;


FOR var_name := initial_value TO final_value DO
begin
    program_statement; {to execute more than one statement in a for }
    program_statement; {loop, you group them using the begin and    }
    program_statement  {end statements                              }
end;                   {semi-colon here depends upon next keyword   }


FOR var_name := initial_value DOWNTO final_value DO  program_statement;
```

You must not change the value of the control variable (var_name) inside the loop. The following program illustrates the for statement.

```
program  CELCIUS_TABLE ( output );
var      celcius : integer; farenhiet : real;
begin
         writeln('Degree''s Celcius   Degree''s Farenhiet');
         for  celcius := 1  to  20  do
         begin
             farenhiet := ( 9 / 5 ) * celcius + 32;
             writeln( celcius:8, '      ',farenhiet:16:2 )
         end
end.
```

CLASS EXERCISE

What is the resultant output when this program is run.

```
program  FOR_TEST ( output );
var      s, j, k, i, l : integer;
begin
         s := 0;
         for  j:= 1 to 5 do
         begin
             write( j );
             s := s + j
         end;
         writeln( s );
         for  k := 0 to 1 do write( k );
         for  i := 10 downto 1 do writeln( i );
         j := 3; k := 8; l := 2;
         for  i := j to k do  writeln( i + l )
end.
```

Answer:

CLASS EXERCISE

What is the resultant output when this program is run.

```
program  FOR_TEST ( output );
var      s, j, k, i, l : integer;
begin
         s := 0;
         for  j:= 1 to 5 do
         begin
             write( j );
             s := s + j
         end;
         writeln( s );
         for  k := 0 to 1 do write( k );
         for  i := 10 downto 1 do writeln( i );
         j := 3; k := 8; l := 2;
         for  i := j to k do  writeln( i + l )
end.
```

```
Class Exercise .. Output of program FOR_TEST is,
1234515
0110
9
8
7
6
5
4
3
2
1
5
6
7
8
9
10
```

PROGRAM NINE

For the first twenty values of farenhiet, print out the equivalent degree in celcuis (Use a tabular format, with appropiate headings). [C = (5 / 9) * Farenhiet - 32]

Use the statement writeln('<14>'); to clear the screen.

Answer:

PROGRAM NINE

For the first twenty values of farenhiet, print out the equivalent degree in celcuis (Use a tabular format, with appropiate headings). [C = (5 / 9) * Farenhiet - 32]

Use the statement writeln('<14>'); to clear the screen.

```
PROGRAM NINE  Table of 1 to 20 Celcius
program PROG9 (output);
var   farenhiet : real;
      celcius   : integer;
begin
    writeln('<14>');     {clear screen on DG machine}
    writeln('Degree''s Celcius   Degree''s Farenhiet');
    for celcius := 1 to 20 do
    begin
       farenhiet := ( 9 / 5 ) * celcius + 32;
       writeln( celcius:8, '      ', farenhiet:16:2 )
    end
end.
```

NESTED LOOPS

A for loop can occur within another, so that the inner loop (which contains a block of statements) is repeated by the outer loop.

RULES RELATED TO NESTED FOR LOOPS

1. Each loop must use a seperate variable
2. The inner loop must begin and end entirely within the outer loop.

CLASS EXERCISE

Determine the output of the following program,

```
program  NESTED_FOR_LOOPS (output);
var      line, column : integer;
begin
         writeln('LINE');
         for  line := 1 to 6 do
         begin
             write( line:2 );
             for  column := 1 to 4 do
             begin
                 write('COLUMN':10);   write(column:2)
             end;
             writeln
         end
end.
```

Answer:

CLASS EXERCISE

Determine the output of the following program,

```
program  NESTED_FOR_LOOPS (output);
var      line, column : integer;
begin
         writeln('LINE');
         for  line := 1 to 6 do
         begin
             write( line:2 );
             for  column := 1 to 4 do
             begin
                 write('COLUMN':10);   write(column:2)
             end;
             writeln
         end
end.
```

```
Class exercise .. output of program NESTED_FOR_LOOPS is,
LINE
 1    COLUMN 1    COLUMN 2    COLUMN 3    COLUMN 4
 2    COLUMN 1    COLUMN 2    COLUMN 3    COLUMN 4
 3    COLUMN 1    COLUMN 2    COLUMN 3    COLUMN 4
 4    COLUMN 1    COLUMN 2    COLUMN 3    COLUMN 4
 5    COLUMN 1    COLUMN 2    COLUMN 3    COLUMN 4
 6    COLUMN 1    COLUMN 2    COLUMN 3    COLUMN 4
```

PROGRAM TEN
Given that the reactance (Xc) of a capacitor equals 1 / (2PIfC), where f is the frequency in hertz, C is the capacitance in farads, and PI is 3.14159, write a program that displays the reactance of five successive capacitor's (their value typed in from the keyboard), for the frequency range 100 to 1000 hertz in 10hz steps.

Answer:

PROGRAM TEN
Given that the reactance (Xc) of a capacitor equals 1 / (2PIfC), where f is the frequency in hertz, C is the capacitance in farads, and PI is 3.14159, write a program that displays the reactance of five successive capacitor's (their value typed in from the keyboard), for the frequency range 100 to 1000 hertz in 10hz steps.

```
program PROG10 (input, output);
const PI = 3.14159;
var  frequency , loopcount, innerloop : integer;
     capacitor,  Xc                   : real;
begin
   for loopcount := 1 to 5 do
   begin
     writeln;
     writeln('Enter capacitance farad value for capacitor #',
              loopcount);
     readln( capacitor );
     for innerloop := 1 to 10 do
     begin
        frequency := innerloop * 100;
        Xc := 1 / ( 2 * PI * frequency * capacitor );
        write('At ',frequency:4,'hz ');
        writeln('the reactance is ', Xc,' ohms.')
     end
   end
end.
```

PROGRAM ELEVEN
The factorial of an integer is the product of all integers up to and including that integer, except that the factorial of 0 is 1.

```
eg,  3! = 1 * 2 * 3    (answer=6)
```

Evaluate the factorial of an integer less than 20, for five numbers input successively via the keyboard.

Answer:

PROGRAM ELEVEN
The factorial of an integer is the product of all integers up to and including that integer, except that the factorial of 0 is 1.

```
eg,  3! = 1 * 2 * 3    (answer=6)
```

Evaluate the factorial of an integer less than 20, for five numbers input successively via the keyboard.

```
program PROG11 (input, output);
var  loopcount, innerloop, number, factorial : integer;
begin
   for loopcount := 1 to 5 do
   begin
      writeln;
      writeln('Enter number ',loopcount, ' for calculation');
      readln( number );
      if number = 0 then factorial := 0
      else
      begin
         factorial := 1;
         for innerloop := number downto 1 do
            factorial := factorial * innerloop
      end;
      writeln('The factorial of ',number,' is ',factorial)
   end
end.
```

THE WHILE LOOP

The while loop is similar to the for loop shown earlier, in that it allows a {group of} program statement(s) to be executed a number of times. The structure of the while statement is,

```
while   condition_is_true  do
begin
        program statement;
        program statement
end;         {semi-colon depends upon next keyword}
```

or, if only a single program statement is to be executed,

```
while   condition_is_true   do   program statement;
```

The program statement(s) are executed when the condition evaluates as true. Somewhere inside the loop the value of the variable which is controlling the loop (ie, being tested in the condition) must change so that the loop can finally exit.

SELF TEST

Determine the output of the following program

```
program WHILE_DEMO (output);
const   PI = 3.14;
var     XL, Frequency, Inductance : real
begin
        Inductance := 1.0;
        Frequency  := 100.00;
        while  Frequency < 1000.00 do
        begin
            XL := 2 * PI * Frequency * Inductance;
            writeln('XL at ',Frequency:4:0,' hertz = ', XL:8:2 );
            Frequency := Frequency + 100.00
        end
 end.
```

Answer:

SELF TEST

Determine the output of the following program

```
program WHILE_DEMO (output);
const   PI = 3.14;
var     XL, Frequency, Inductance : real
begin
        Inductance := 1.0;
        Frequency  := 100.00;
        while  Frequency < 1000.00 do
        begin
            XL := 2 * PI * Frequency * Inductance;
            writeln('XL at ',Frequency:4:0,' hertz = ', XL:8:2 );
            Frequency := Frequency + 100.00
```

```
          end
   end.
```

```
Self test .. Output of program WHILE_DEMO is..
XL at  100 hertz = .........
XL at  200 hertz = .........
............................
XL at 1000 hertz = .........
```

REPEAT

The REPEAT statement is similar to the while loop, how-ever, with the repeat statement, the conditional test occurs after the loop. The program statement(s) which constitute the loop body will be executed at least once. The format is,

```
repeat
     program statement;
until  condition_is_true;   {semi-colon depends on next keyword}
```

There is no need to use the begin/end keywords to group more than one program statement, as all statements between repeat and until are treated as a block.

The CASE statement

The case statement allows you to rewrite code which uses a lot of if else statements, making the program logic much easier to read. Consider the following code portion written using if else statements,

```
if operator = '*' then result := number1 * number2
   else if operator = '/' then result := number1 / number2
       else if operator = '+' then result := number1 + number2
           else if operator = '-' then result := number1 - number2
               else  invalid_operator = 1;
```

Rewriting this using case statements,

```
case  operator  of
      '*' : result:= number1 * number2;
      '/' : result:= number1 / number2;
      '+' : result:= number1 + number2;
      '-' : result:= number1 - number2;
otherwise    invalid_operator := 1
end;
```

The value of *operator* is compared against each of the values specified. If a match occurs, then the program statement(s) associated with that match are executed.

If *operator* does not match, it is compared against the next value. The purpose of the *otherwise* clause ensures that appropiate action is taken when *operator* does not match against any of the specified cases.

You must compare the variable against a constant, how-ever, it is possible to group cases as shown below,

```
case  user_request  of
      'A' :
      'a' :  call_addition_subprogram;
      's' :
      'S' :  call_subtraction_subprogram;
end;
```

PROGRAM TWELVE

Convert the following program, using appropiate case statements.

```
program  PROG_TWELVE (input, output);
var      invalid_operator : boolean;
         operator : char;
         number1, number2, result : real;
begin
         invalid_operator := FALSE;
         writeln('Enter two numbers and an operator in the format');
         writeln(' number1 operator number2');
         readln(number1); readln(operator); readln(number2);
         if operator = '*' then result := number1 * number2
         else if operator = '/' then result := number1 / number2
         else if operator = '+' then result := number1 + number2
         else if operator = '-' then result := number1 - number2
         else invalid_operator := TRUE;

         if invalid_operator then
            writeln('Invalid operator')
         else
            writeln(number1:4:2,' ', operator,' ', number2:4:2,' is '
                    ,result:5:2)
end.
```

Answer:

PROGRAM TWELVE

Convert the following program, using appropiate case statements.

```
program  PROG_TWELVE (input, output);
var      invalid_operator : boolean;
         operator : char;
         number1, number2, result : real;
begin
         invalid_operator := FALSE;
         writeln('Enter two numbers and an operator in the format');
         writeln(' number1 operator number2');
         readln(number1); readln(operator); readln(number2);
         if operator = '*' then result := number1 * number2
         else if operator = '/' then result := number1 / number2
         else if operator = '+' then result := number1 + number2
         else if operator = '-' then result := number1 - number2
         else invalid_operator := TRUE;

         if invalid_operator then
            writeln('Invalid operator')
         else
            writeln(number1:4:2,' ', operator,' ', number2:4:2,' is '
                    ,result:5:2)
end.
```

```
Conversion of PROG_TWELVE using case operator
program  PROG_TWELVE (input, output);          {Data General Version}
var      invalid_operator : boolean;
         operator : char;
         number1, number2, result : real;
begin
         invalid_operator := FALSE;
         writeln('Enter two numbers and an operator in the format');
         writeln(' number1 operator number2');
         readln(number1); readln(operator); readln(number2);
         case operator of
             '*': result := number1 * number2;
             '/': result := number1 / number2;
             '+': result := number1 + number2;
             '-': result := number1 - number2;
         otherwise invalid_operator := TRUE
         end;
         if invalid_operator then
            writeln('Invalid operator')
         else
            writeln(number1:4:2,' ', operator,' ', number2:4:2,' is '
                    ,result:5:2)
end.
{Note that turbo pascal does not support use of otherwise}
{Special changes for Turbo are                           }

         case operator of
             '*': result := number1 * number2;
             '/': result := number1 / number2;
             '+': result := number1 + number2;
             '-': result := number1 - number2;
         else invalid_operator := TRUE
         end;
```

ENUMERATED DATA TYPES

Enumerated variables are defined by the programmer. It allows you to create your own data types, which consist of a set of symbols. You first create the set of symbols, and assign to them a new data type variable name.

Having done this, the next step is to create working variables to be of the same type. The following portions of code describe how to create enumerated variables.

```
        type  civil_servant = ( clerk, police_officer, teacher, mayor );
        var   job, office : civil_servant;
```

The new data type created is *civil_servant*. It is a set of values, enclosed by the ( ) parenthesis. These set of values are the only ones which variables of type *civil_servant* can assume or be assigned.

The next line declares two working variables, *job* and *office*, to be of the new data type *civil_servant*.

The following assignments are valid,

```
        job := mayor;
        office := teacher;

        if office = mayor then writeln('Hello mayor!');
```

The list of values or symbols between the parenthesis is an ordered set of values. The first symbol in the set has an ordinal value of zero, and each successive symbol has a value of one greater than its predecessor.

```
        police_officer < teacher
```

evaluates as true, because *police_officer* occurs before *teacher* in the set.

MORE EXAMPLES ON ENUMERATED DATA TYPES

```
        type beverage = ( coffee, tea, cola, soda, milk, water );
             color    = ( green, red, yellow, blue, black, white );
        var  drink : beverage;
             chair : color;

        drink := coffee;
        chair := green;

        if chair = yellow then drink := tea;
```

ADDITIONAL OPERATIONS WITH USER DEFINED VARIABLE TYPES

Consider the following code,

```
        type Weekday = ( Monday, Tuesday, Wednesday, Thursday, Friday );
        var  Workday : Weekday;
```

The first symbol of the set has the value of 0, and each symbol which follows is one greater. Pascal provides three additional operations which are performed on user defined variables. The three operations are,

```
 ord( symbol )   returns the value of the symbol, thus ord(Tuesday)
                 will give a value of 1
 pred( symbol )  obtains the previous symbol, thus
                 pred(Wednesday) will give Tuesday

 succ( symbol )  obtains the next symbol, thus succ(Monday)
                 gives Tuesday
```

Enumerated values can be used to set the limits of a *for* statement, or as a constant in a *case* statement, eg,

```
        for Workday := Monday to Friday
             .........

        case Workday of
              Monday : writeln('Mondays always get me down.');
              Friday : writeln('Get ready for partytime!')
        end;
```

Enumerated type values cannot be input from the keyboard or outputted to the screen, so the following statements are illegal,

```
        writeln( drink );
        readln( chair );
```

SELF TEST ON ENUMERATED DATA TYPES
Whats wrong with?

```
        type Day = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
                    Sunday);
        var  Today : Day;

        for Today := Sunday to Monday do
        begin
           writeln( Today );
           Today := succ( Today )
        end;
```

Whats wrong with

```
        type COLOR = ( Red, Blue, Green, Yellow );
        var  Green, Red : COLOR;
```

SUBRANGES
Just as you can create your own set of pre-defined data types, you can also create a smaller subset or subrange of an existing set which has been previously defined. Each subrange consists of a defined lower and upper limit. Consider the following,

```
        type DAY = (Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday);
             Weekday = Monday..Friday;     {subrange of DAY}
             Weekend = Saturday..Sunday;   {subrange of DAY}
             Hours   = 0..24;              {subrange of integers}
             Capitals= 'A'..'Z';           {subrange of characters)

        NOTE: You cannot have subranges of type real.
```

Which of the following are legal

```
        type  Gradepoints = 0.0..4.0;
              Numbers = integer;
              Alphabet = 'Z'..'A';
```

Answer:

Which of the following are legal

```
        type  Gradepoints = 0.0..4.0;
              Numbers = integer;
              Alphabet = 'Z'..'A';


        Which of the following are legal....NONE ARE!
        Cannot have subranges of real type
        Cannot do this, must be  Numbers = 1..500;
        Cannot do this, must be  Alphabet = 'A'..'Z'   as 'A' comes before 'Z'
```

SELF TEST

1. Write a for loop to display the following output

```
        1 2 3 4 5 6 7 8 9 10
```

2. Write a for loop to display the following output

```
        1
        22
        333
        4444
        55555
```

3. Write a while loop to display the following output

```
        A B C D E F
```

4. Rewrite the following if statements as a **Case** statement

```
        if flag = 1 then number := 10
                else if flag = 2 then number := 20
                        else if flag = 3 then number := 40;
```

5. Define an enumerated data type called *chair*, which has the set of values *lounge, deck, executive*

6. Write pascal statements to define a new working variable *mychair*, of type *chair*, and assign the value *deck* to this new variable.

7. Define a new subrange called *minutes*, which has a set of ranges from 0 to 60.

Answer:

SELF TEST: ANSWERS

1. Write a for loop to display the following output

```
        1 2 3 4 5 6 7 8 9 10

        for loop := 1 to 10 do write( loop, ' ' );
```

2. Write a for loop to display the following output

```
        1
        22
        333
        4444
        55555


        for loop := 1 to 5 do
        begin
                for loop1 := 1 to loop do write( loop );
                writeln
        end
```

3. Write a while loop to display the following output

```
        A B C D E F

        loop := 'A';
        while loop <= 'F' do
        begin
                write( loop, ' ' );
                loop := loop + 1
        end;
```

4. Rewrite the following if statements as a **Case** statement

```
        if flag = 1 then number := 10
                else if flag = 2 then number := 20
                        else if flag = 3 then number := 40;

        case flag of
                1 : number := 10;
                2 : number := 20;
                3 : number := 40
        end;
```

5. Define an enumerated data type called *chair*, which has the set of values *lounge, deck, executive*

```
        type chair = ( lounge, deck, executive );
```

6. Write pascal statements to define a new working variable *mychair*, of type *chair*, and assign the value *deck* to this new variable.

```
        var  mychair : chair;

        mychair := deck;
```

7. Define a new subrange called *minutes*, which has a set of ranges from 0 to 60.

```
        type minutes = 0..60;
```

ARRAYS

An array is a structure which holds many variables, all of the same data type. The array consists of so many elements, each element of the array capable of storing one piece of data (ie, a variable).

An array is defined as follows,

```
        type  array_name = ARRAY [lower..upper] of data_type;
```

Lower and *Upper* define the boundaries for the array. *Data_type* is the type of variable which the array will store, eg, type int, char etc. A typical declaration follows,

```
        type  intarray = ARRAY [1..20] of integer;
```

This creates a definition for an array of integers called *intarray*, which has 20 separate locations numbered from 1 to 20. Each of these positions (called an **element**), holds a single integer. The next step is to create a working variable to be of the same type, eg,

```
        var  numbers : intarray;
```

Each element of the *numbers* array is individually accessed and updated as desired.

To assign a value to an element of an array, use

```
        numbers[2] := 10;
```

This assigns the integer value 10 to element 2 of the *numbers* array. The value or element number (actually its called an index) is placed inside the square brackets.

To assign the value stored in an element of an array to a variable, use

```
        number1 := numbers[2];
```

This takes the integer stored in element 2 of the array *numbers*, and makes the integer *number1* equal to it.

Consider the following array declarations

```
const  size = 10;
       last = 100;
type   sub = 'a'..'z';
       color = (green, yellow, red, orange, blue );
var    chararray : ARRAY [1..size] of char;
       {an array of 10 characters. First element is chararray[1],
        last element is chararry[10] }

       intarray  : ARRAY [sub] of integer;
       {an array of 26 integers. First element is intarray['a']
        last element is intarray['z']  }

       realarray : ARRAY [5..last] of real;
       {an array of 95 real numbers. First element is realarray[5]
        last element is realarray[100]   }

       artstick  : ARRAY [-3..2] of color;
       {an array of 6 colors. First element is artstick[-3]
        last element is artstick[2]  }

       huearray  : ARRAY [color] of char;
       {an array of 6 characters. First element is huearray[green]
        last element is huearray[blue]   }
```

CHARACTER ARRAYS

You can have arrays of characters. Text strings from the keyboard may be placed directly into the array elements. You can print out the entire character array contents. The following program illustrates how to do this,

```
program CHARRAY (input,output );
type word = PACKED ARRAY [1..10] of char;
var  word1 : word;
     loop  : integer;
begin
  writeln('Please enter in up to ten characters.');
  readln( word1 );  { this reads ten characters directly from the
                      standard input device, placing each character
                      read into subsequent elements of word1 array }

  writeln('The contents of word1 array is ');

  for loop := 1 to 10 do                      {print out each element}
     writeln('word1[',loop,'] is ',word1[loop] );

  writeln('Word1 array contains ', word1 )   {print out entire array}
end.
```

Note the declaration of **PACKED ARRAY**, and the use of just the array name in conjuction with the *readln* statement. If the user typed in

```
        Hello there
```

then the contents of the array *word1* will be,

```
         word1[1] = H
         word1[2] = e
         word1[3] = l
         word1[4] = l
         word1[5] = o
         word1[6] =     { a space }
         word1[7] = t
         word1[8] = h
         word1[9] = e
         word1[10]= r
```

The entire contents of a packed array of type char can also be outputted to the screen simply the using the array name without an index value, ie, the statement

```
         writeln('Word1 array contains ', word1 );
```

will print out all elements of the array *word1*, displaying

```
         Hello ther
```

INTEGER ARRAYS

Arrays can hold any of the valid data types, including integers. Integer arrays cannot be read or written as an entire unit, only packed character arrays can. The following program demonstrates an integer array, where ten successive numbers are inputted, stored in separate elements of the array numbers, then finally outputted to the screen one at a time.

```
        program INT_ARRAY (input,output );
        type int_array = ARRAY [1..10] of integer;
        var  numbers : int_array;
             loop  : integer;
        begin
          writeln('Please enter in up to ten integers.');
          for loop := 1 to 10 do
             readln( numbers[loop] );

          writeln('The contents of numbers array is ');
          { print out each element }
          for loop := 1 to 10 do
             writeln('numbers[',loop:2,'] is ',numbers[loop] )
        end.
```

SELF TEST

What does the following program display on the screen.

```
        program ARRAY_TEST (output);
        var     numbers : ARRAY [1..5] of integer;
        begin
             numbers[1] := 7;
             numbers[2] := 13;
             numbers[3] := numbers[2] - 1;
             numbers[4] := numbers[3] DIV 3;
             numbers[5] := numbers[3] DIV numbers[4];
             for loop := 1 to 5 do
                writeln('Numbers[',loop,'] is', numbers[loop] )
        end.
```

Answer:

SELF TEST

What does the following program display on the screen.

```
program ARRAY_TEST (output);
var     numbers : ARRAY [1..5] of integer;
begin
     numbers[1] := 7;
     numbers[2] := 13;
     numbers[3] := numbers[2] - 1;
     numbers[4] := numbers[3] DIV 3;
     numbers[5] := numbers[3] DIV numbers[4];
     for loop := 1 to 5 do
        writeln('Numbers[',loop,'] is', numbers[loop] )
end.
```

```
Self Test .. Output of ARRAY_TEST is..
Numbers[1] is 7
Numbers[2] is 13
Numbers[3] is 12
Numbers[4] is 4
Numbers[5] is 3
```

HOW CHARACTERS ARE INTERNALLY REPRESENTED

Internally, most computers store characters according to the ASCII format. ASCII stands for American Standard Code for Information Interchange. Characters are stored according to a numbered sequence, whereby A has a value of 64 decimal, B a value of 65 etc. Several functions which manipulate characters follow.

- CHR
  The **chr** or character position function returns the character associated with the ASCII value being asked, eg,
-
- `chr( 65 )   will return the character   A`
- ORD
  The **ord** or ordinal function returns the ASCII value of a requested character. In essence, it works backwards to the **chr** function. Ordinal data types are those which have a predefined, known set of values.
  Each value which follows in the set is one greater than the previous. Characters and integers are thus ordinal data types.
-
- `ord( 'C' )   will return the value   67`
- SUCC
  The **succ**essor function determines the next value or symbol in the set, thus
-
- `succ( 'd' )  will return  e`
- PRED
  The **pred**ecessor function determines the previous value or symbol in the set, thus
-
- `pred( 'd' )  will return  c`

COMPARISON OF CHARACTER VARIABLES

Character variables, when compared against each other, is done using the ASCII value of the character. Consider the following portion of code,

```
var  letter1, letter2 : char;
begin
     letter1 := 'A';  letter2 := 'C';
     if letter1 < letter2   then
        writeln( letter1, ' is less than ',letter2 )
     else
        writeln( letter2, ' is less than ',letter1 )
end.
```

STRING ARRAYS, COMPARISON OF

Packed character arrays of the same length are comparable. There follows a short program illustrating this,

```
program PACKED_CHAR_COMPARISON (output);
type  string1 = packed array [1..6] of char;
var   letter1, letter2 : string1;
begin
      letter1 := 'Hello ';
      letter2 := 'HellO ';

      if letter1 < letter2 then
         writeln( letter1,' is less than ',letter2)
      else
         writeln( letter2,' is less than ',letter1)
end.
```

SELF TEST

1. Write a Pascal statement to define an array called *numbers*, which is an integer array with elements ranging from 1 to 20

2. Write a Pascal statement to create an array called *mynumbers*, of type *numbers*, which was defined in 1. above

3. Write a Pascal statement which assigns the integer value 20 to element 4 of the array *mynumbers*, which was declared in 2. above

4. Write Pascal statements which define a packed array of characters (15 elements), called *word*, create a working variable of type *word* called ***myword***, and then reads input from the keyboard into the array *myword*

5. Write Pascal statements which will sum the contents of an integer array called *mynumbers*, which has 20 elements numbered 1 to 20

6. Write a Pascal statement which will initialise a packed character array called *message* to the string 'Hello there!'. The array has thirteen elements

7. Write a Pascal statement to display the ASCII value of the letter 'A'

8. Write a Pascal statement to display the character represented by the ASCII value 52

9. Write a Pascal statement to display the character which follows 'F'

10. Write a Pascal statement to display the character which comes before 'Z'

Answer:

SELF TEST: ANSWERS

1. Write a Pascal statement to define an array called *numbers*, which is an integer array with elements ranging from 1 to 20

```
type  numbers = ARRAY[1..20] of integer;
```

2. Write a Pascal statement to create an array called *mynumbers*, of type *numbers*, which was defined in 1. above

```
var  mynumbers : numbers;
```

3. Write a Pascal statement which assigns the integer value 20 to element 4 of the array *mynumbers*, which was declared in 2. above

```
mynumbers[4] := 20;
```

4. Write Pascal statements which define a packed array of characters (15 elements), called *word*, create a working variable of type *word* called ***myword***, and then reads input from the keyboard into the array *myword*

```
type  word = PACKED ARRAY[1..15] of char;
var   myword : word;

begin
       readln( myword );
```

5. Write Pascal statements which will sum the contents of an integer array called *mynumbers*, which has 20 elements numbered 1 to 20

```
total := 0;
for loop := 1 to 20 do
       total := total + mynumbers[loop];
```

6. Write a Pascal statement which will initialise a packed character array called *message* to the string 'Hello there!'. The array has thirteen elements

```
message := 'Hello there! ';
```

7. Write a Pascal statement to display the ASCII value of the letter 'A'

```
writeln( ord('A') );
```

8. Write a Pascal statement to display the character represented by the ASCII value 52

```
writeln( chr(52) );
```

9. Write a Pascal statement to display the character which follows 'F'

```
writeln( succ('F') );
```

10. Write a Pascal statement to display the character which comes before 'Z'

```
writeln( pred('Z') );
```

COMMON FUNCTIONS

The Pascal language provides a range of functions to perform data transformation and calculations. The following section provides an explanation of the commonly provided functions,

- ABS
  The ABSolute function returns the absolute value of either an integer or real, eg,
-
- *`ABS( -21 )    returns   21`*
- *`ABS( -3.5)    returns   3.5000000000E+00`*
- COS
  The COSine function returns the cosine value, in radians, of an argument, eg,
-
- *`COS( 0 )       returns  1.0`*
- EXP
  The exponential function calculates e raised to the power of a number, eg,
-
- *`EXP(10 )      returns e to the power of 10`*

  There is no function in Pascal to calculate expressions such as an, ie,

```
      23  is 2*2*2  = 8
```

  These are calculated by using the formula

```
     an = exp( n * ln( a ) )
```

- LN
  The logarithm function calculates the natural log of a number greater than zero.
- ODD
  The odd function determines when a specified number is odd or even, returning true when the number is odd, false when it is not.
- ROUND
  The round function rounds its number (argument) to the nearest integer. If the argument is positive
- `rounding is up for fractions greater than or equal to .5`
- `rounding is down for fractions less than .5`

  If the number is negative

```
     rounding is down (away from zero) for fractions >= .5
     rounding is up (towards zero) for fractions < .5
```

- SIN
  The sine function returns the sine of its argument, eg,
-
- *`SIN( PI / 2 )    returns  1.0`*
- SQR
  The square function returns the square (ie the argument multiplied by itself) of its supplied argument,
-
- *`SQR( 2 )         returns  4`*
- SQRT
  This function returns {always returns a real} the square root of its argument, eg,
-
- *`SQRT( 4 )        returns  2.0000000000E+00`*

- TRUNC
  This function returns the whole part (no decimal places) of a real number.
-
- `TRUNC(4.87)      returns 4`
- `TRUNC(-3.4)      returns 3`

PROGRAM FOURTEEN
Given the following list of wages stored in an array,

```
        210.33      119.78      191.05      222.94
```

calculate the total breakdown of required coins (ignore dollars) into 50c, 20c, 10c, 5c, 2c, and 1c pieces.

Answer:

PROGRAM FOURTEEN
Given the following list of wages stored in an array,

```
        210.33      119.78      191.05      222.94
```

calculate the total breakdown of required coins (ignore dollars) into 50c, 20c, 10c, 5c, 2c, and 1c pieces.

```
program PROG14 (output);     {coin program}
var     wages : array[1..6] of real;
        cents : real;
        loop, fiftys, twentys, tens, fives, twos, ones : integer;
begin
    {initialise wages}
    wages[1] := 210.33;    wages[2] := 119.78;
    wages[3] := 191.05;    wages[4] := 222.94;
    wages[5] := 0.0;       { end of wage terminator }
    loop := 1;

    fiftys := 0; twentys := 0; tens := 0; fives := 0; twos := 0;
    ones := 0;

    while (wages[loop] <> 0.0 ) do
    begin
        cents := wages[loop] - trunc( wages[loop] ); {get cents}
        while cents >= 0.4999 do
        begin
            fiftys := fiftys + 1;
            cents := cents - 0.50
        end;
        while cents >= 0.1999 do
        begin
            twentys := twentys + 1;
            cents := cents - 0.20
        end;
        while cents >= 0.0999 do
        begin
            tens := tens + 1;
            cents := cents - 0.10
        end;
        while cents >= 0.0499 do
        begin
            fives := fives + 1;
            cents := cents - 0.05
        end;
        while cents >= 0.0199 do
        begin
            twos := twos + 1;
            cents := cents - 0.02
        end;
        while cents >= 0.00999 do
        begin
```

```
            ones := ones + 1;
            cents := cents - 0.01
        end;
        loop := loop + 1
    end;
    writeln;
    writeln('The total breakdown of coins required is');
    writeln('     50c    20c    10c    5c     2c     1c');
    writeln(fiftys:7,twentys:7,tens:7,fives:7,twos:7,ones:7)
end.
```

OPERATOR PRECEDENCE

Pascal, when determining how to perform calculations, works according to pre-defined rules. These rules may be overridden by the use of parenthesis ().

The priority given to the various operators, from highest to lowest, are

```
    NOT                 Negation
    * / DIV MOD AND
    + - OR
    =  <>  <  <=  >  >=  IN

    The operators are always evaluated left to right
```

Class Exercise on Operator precedence

Given that

```
  A := 1;      B := 2;      C := 4;
```

What does X equal after each of the following statements,

```
  X := A / B / C;               ________________
  X := A + B / C;               ________________
  X := A * B * C;               ________________
  X := A * B - C;               ________________
  X := A + B + C;               ________________
  X := A / B * C;               ________________
  X := A * B / C;               ________________
  X := A + B - C;               ________________
```

Click here for answer

Parenthesis are used to override the order of precedence. Consider the expression

```
             A + B
        X = -------
             C + D
```

becomes in Pascal

```
        X := ( A + B ) / ( C + D )
```

and the expression

```
                   B
        X = A  +  ---  +  D
                   C
```

becomes in Pascal

```
        X := A + ( B / C ) + D
```

Self Test on Operator Precedence

Write statements in Pascal which correctly express each of the following mathematical expressions.

```
             2                                2
 1. Z = X + Y                 2.  Z = ( X + Y )

         A + B + E                         B
 3. Z = -----------           4.  Z = A + ---
```

```
          D + E                          C

        A + B                              B
 5. Z = -------                6.  Z = A + -------
          C                              D - C
```

Answer:

Self Test on Operator Precedence

Write statements in Pascal which correctly express each of the following mathematical expressions.

```
             2                                2
 1. Z = X + Y                  2.  Z = ( X + Y )

        A + B + E                        B
 3. Z = -----------            4.  Z = A + ---
          D + E                          C

        A + B                              B
 5. Z = -------                6.  Z = A + -------
          C                              D - C


             2                                2
 1. Z = X + Y                  2.  Z = ( X + Y )

    Z := X + (Y * Y);              Z := (X + Y) * (X + Y);

        A + B + E                        B
 3. Z = -----------            4.  Z = A + ---
          D + E                          C

    Z := (A+B+E) / (D+E);          Z := A + (B / C);

        A + B                              B
 5. Z = -------                6.  Z = A + -------
          C                              D - C

    Z := (A + B) / C;              Z := A + ( B / ( D - C ) );
```

MODULAR PROGRAMMING USING PROCEDURES AND FUNCTIONS

Modular programming is a technique used for writing large programs. The program is subdivided into small sections. Each section is called a **module**, and performs a single task.

Examples of tasks a module might perform are,

- displaying an option menu
- printing results
- calculating average marks
- sorting data into groups

A module is known by its name, and consists of a set of program statements grouped using the begin and end keywords. The module (group of statements) is executed when you type the module name.

Pascal uses three types of modules. The first two are called **PROCEDURES**, the other a **FUNCTION**.

- Simple procedures do not accept any arguments (values or data) when the procedure is executed (called).
- Complex procedures accept values to work with when they are executed.
- Functions, when executed, return a value (ie, calculate an answer which is made available to the module which wants the answer)

Procedures help support structured program design, by allowing the independant development of modules. Procedures are essentially sub-programs.

SIMPLE PROCEDURES

Procedures are used to perform tasks such as displaying menu choices to a user. The procedure (module) consists of a set of program statements, grouped by the *begin* and *end* keywords. Each procedure is given a **name**, similar to the title that is given to the main module.

Any variables used by the procedure are declared before the keyword *begin*.

```
PROCEDURE  DISPLAY_MENU;
begin
     writeln('<14>Menu choices are');
     writeln(' 1: Edit text file');
     writeln(' 2: Load text file');
     writeln(' 3: Save text file');
     writeln(' 4: Copy text file');
     writeln(' 5: Print text file')
end;
```

The above procedure called **DISPLAY_MENU**, simply executes each of the statements in turn. To use this in a program, we write the name of the procedure, eg,

```
program PROC1 (output);

PROCEDURE  DISPLAY_MENU;
begin
     writeln('<14>Menu choices are');
     writeln(' 1: Edit text file');
     writeln(' 2: Load text file');
     writeln(' 3: Save text file');
     writeln(' 4: Copy text file');
     writeln(' 5: Print text file')
end;

begin
        writeln('About to call the procedure');
        DISPLAY_MENU;
        writeln('Now back from the procedure')
end.
```

In the main portion of the program, it executes the statement

```
        writeln('About to call the procedure');
```

then calls the procedure DISPLAY_MENU. All the statements in this procedure are executed, at which point we go back to the statement which follows the call to the procedure in the main section, which is,

```
        writeln('Now back from the procedure')
```

The sample output of the program is

```
About to call the procedure
Menu choices are
 1: Edit text file
 2: Load text file
 3: Save text file
 4: Copy text file
 5: Print text file
Now back from the procedure
```

SELF TEST ON SIMPLE PROCEDURES

What does this program display?

```
program  SIMPLE_PROCEDURES (input,output);
var      time, distance, speed : real;

procedure display_title;
begin
     writeln('This program calculates the distance travelled based');
     writeln('on two variables entered from the keyboard, speed and');
     writeln('time.')
end;

procedure get_choice;
begin
      writeln('Please enter the speed in MPH');
      readln( speed );
      writeln('Please enter the time in hours');
      readln( time )
end;

procedure calculate_distance;
begin
      distance := speed * time
end;

procedure display_answer;
begin
      writeln('The distance travelled is ', distance:5:2,' miles.')
end;

begin      {This is the actual start of the program}
      display_title;
      get_choice;
      calculate_distance;
      display_answer
end.
```

{Note that the three variables, time, speed and distance, are available to all procedures. They may be updated by any procedure, and are known as **GLOBAL** variables}.

Variables which are declared external (outside of) to any procedure are accessible anywhere in the program. The use of global variables is limited. In a large program, it is difficult to determine which procedure updates the value of a global variable.

PROGRAM FIFTEEN
Convert the calculator program (program 12), using simple procedures, to perform the various calculations. Use global variables for *number1*, *operator* and *number2*.

Answer:

PROGRAM FIFTEEN
Convert the calculator program (program 12), using simple procedures, to perform the various calculations. Use global variables for *number1*, *operator* and *number2*.

```
program PROG15 (input,output);
var      invalid_operator : boolean;
         operator : char;
         number1, number2, result : real;

procedure MULTIPLY;
begin
         result := number1 * number2
end;

procedure DIVIDE;
begin
         result := number1 / number2
end;

procedure ADD;
begin
         result := number1 + number2
end;

procedure SUBTRACT;
begin
         result := number1 - number2
end;

procedure GET_INPUT;
begin
         writeln('Enter two numbers and an operator in the format');
         writeln(' number1 operator number2');
         readln(number1); readln(operator); readln(number2)
end;

begin
         invalid_operator := FALSE;
         GET_INPUT;
         case operator of
             '*': MULTIPLY;
             '/': DIVIDE;
             '+': ADD;
             '-': SUBTRACT;
         otherwise invalid_operator := TRUE
         end;
         if invalid_operator then
            writeln('Invalid operator')
         else
            writeln(number1:4:2,' ', operator,' ', number2:4:2,' is '
                    ,result:5:2)
end.

{Special changes for Turbo are
         case operator of
             '*': result := MULTIPLY;
```

```
    '/': result := DIVIDE;
    '+': result := ADD;
    '-': result := SUBTRACT;
else invalid_operator := TRUE
end;                                            }
```

PROCEDURES AND LOCAL VARIABLES

A procedure can declare it's own variables to work with. These variables belong to the procedure in which they are declared. Variables declared inside a procedure are known as **local**.

Local variables can be accessed anywhere between the *begin* and matching *end* keywords of the procedure. The following program illustrates the use and scope (where variables are visible or known) of local variables.

```
program LOCAL_VARIABLES (input, output);
var   number1, number2 : integer;  {these are accessible by all}

procedure add_numbers;
var   result : integer;            {result belongs to add_numbers}
begin
      result := number1 + number2;
      writeln('Answer is ',result)
end;

begin                              {program starts here}
      writeln('Please enter two numbers to add together');
      readln( number1, number2 );
      add_numbers
end.
```

SELF TEST ON LOCAL VARIABLES

Determine this programs output.

```
program  MUSIC (output);
const SCALE = 'The note is ';
var   JohnnyOneNote : char;

procedure Tune;
const SCALE = 'The note now is ';
var   JohnnyOneNote : char;
begin
      JohnnyOneNote := 'A';
      writeln(SCALE, JohnnyOneNote )
end;

begin
      JohnnyOneNote := 'D';
      writeln(SCALE, JohnnyOneNote );
      Tune;
      writeln(SCALE, JohnnyOneNote )
end.
```

Answer:

SELF TEST ON LOCAL VARIABLES

Determine this programs output.

```
program  MUSIC (output);
const SCALE = 'The note is ';
var   JohnnyOneNote : char;

procedure Tune;
const SCALE = 'The note now is ';
var   JohnnyOneNote : char;
begin
      JohnnyOneNote := 'A';
      writeln(SCALE, JohnnyOneNote )
end;

begin
      JohnnyOneNote := 'D';
      writeln(SCALE, JohnnyOneNote );
      Tune;
      writeln(SCALE, JohnnyOneNote )
end.
```

```
Self Test on Local variables, output of program MUSIC is,
The note is D
The note now is A
The note is D
```

PROCEDURES WHICH ACCEPT ARGUMENTS

Procedures may also accept variables (data) to work with when they are called.

Declaring the variables within the procedure

- The variables accepted by the procedure are enclosed using parenthesis.
- The declaration of the accepted variables occurs between the procedure name and the terminating semi-colon.

Calling the procedure and Passing variables (or values) to it

- When the procedure is invoked, the procedure name is followed by a set of parenthesis.
- The variables to be passed are written inside the parenthesis.
- The variables are written in the same order as specified in the procedure.

Consider the following program example,

```
program ADD_NUMBERS (input, output);

procedure CALC_ANSWER ( first, second : integer );
var   result : integer;
begin
      result := first + second;
      writeln('Answer is ', result )
end;
```

```
var   number1, number2 : integer;
begin
      writeln('Please enter two numbers to add together');
      readln( number1, number2 );
      CALC_ANSWER( number1, number2)
end.
```

SELF TEST ON PROCEDURES WHICH ACCEPT PARAMETERS

The output is?

```
program  TestValue (output);
var  x, y : integer;

procedure  NoEffect ( x, y : integer );
begin
    x := y;  y := 0;
    writeln( x, y )
end;

begin
    x := 1;  y := 2;
    writeln( x, y );
    NoEffect( x, y );
    writeln( x, y )
end.
```

Answer:

SELF TEST ON PROCEDURES WHICH ACCEPT PARAMETERS

The output is?

```
program  TestValue (output);
var  x, y : integer;

procedure  NoEffect ( x, y : integer );
begin
    x := y;  y := 0;
    writeln( x, y )
end;

begin
    x := 1;  y := 2;
    writeln( x, y );
    NoEffect( x, y );
    writeln( x, y )
end.
```

```
Self test on procedures which accept arguments, output of Testvalue is
1  2
2  0
1  2
```

Value Parameters

In the previous programs, when variables are passed to procedures, the procedures work with a **copy** of the original variable. The value of the original variables which are passed to the procedure are not changed.

The copy that the procedure makes can be altered by the procedure, but this does not alter the value of the original. When procedures work with copies of variables, they are known as **value parameters**.

Consider the following code example,

```
program  Value_Parameters (output);

procedure  Nochange ( letter : char; number : integer );
begin
     writeln( letter );
     writeln( number );
     letter := 'A';        {this does not alter mainletter}
     number := 32;         {this does not alter mainnumber}
     writeln( letter );
     writeln( number )
end;

var  mainletter : char;    {these variables known only from here on}
     mainnumber : integer;
begin
     mainletter := 'B';
     mainnumber := 12;
     writeln( mainletter );
     writeln( mainnumber );
     Nochange( mainletter, mainnumber );
     writeln( mainletter );
     writeln( mainnumber )
end.
```

PROGRAM SIXTEEN

Write a program, using procedures which accept value parameters, to implement the calculator program as derived in program fifteen. Each procedure will print out its own result. No global variables must be used.

Variable parameters

Procedures can also be implemented to change the value of original variables which are accepted by the procedure. To illustrate this, we will develop a little procedure called **swap**. This procedure accepts two integer values, swapping them over.

Previous procedures which accept value parameters cannot do this, as they only work with a copy of the original values. To force the procedure to use variable parameters, preceed the declaration of the variables (inside the parenthesis after the function name) with the keyword **var**.

This has the effect of using the original variables, rather than a copy of them.

```
        program  Variable_Parameters (output);

        procedure  SWAP ( var value1, value2 : integer );
        var   temp : integer;
        begin
              temp := value1;
              value1 := value2;  {value1 is actually number1}
              value2 := temp     {value2 is actually number2}
        end;

        var   number1, number2 : integer;
        begin
              number1 := 10;
              number2 := 33;
              writeln( 'Number1 = ', number1,' Number2 = ', number2 );
              SWAP( number1, number2 );
              writeln( 'Number1 = ', number1,' Number2 = ', number2 )
        end.
```

When this program is run, it prints out

```
        Number1 = 10  Number2 = 33
        Number1 = 33  Number2 = 10
```

SELF TEST
Why is the following procedure declaration incorrect?

```
        procedure Wrong ( A : integer; var B : integer );
        var   A : integer; B : real;
```

Answer:
….
….
….
….

FUNCTIONS - A SPECIAL TYPE OF PROCEDURE WHICH RETURNS A VALUE
Procedures accept data or variables when they are executed. Functions also accept data, but have the ability to return a value to the procedure or program which requests it. Functions are used to perform mathematical tasks like factorial calculations.

A function

- begins with the keyword *function*
- is similar in structure to a procedure
- somewhere inside the code associated with the function, a value is assigned to the function name
- a function is used on the righthand side of an expression
- can only return a simple data type

The actual heading of a function differs slightly than that of a procedure. Its format is,

```
      function  Function_name (variable declarations) : return_data_type;
```

After the parenthesis which declare those variables accepted by the function, the return data type (preceeded by a colon) is declared.

```
function  ADD_TWO ( value1, value2 : integer ) : integer;
begin
     ADD_TWO := value1 + value2
end;
```

The following line demonstrates how to call the function,

```
result := ADD_TWO( 10, 20 );
```

thus, when ADD_TWO is executed, it equates to the value assigned to its name (in this case 30), which is then assigned to result.

SELF TEST
Determine the output of the following program

```
program  function_time (input, output);
const    maxsize = 80;
type     line = packed array[1..maxsize] of char;

function COUNTLETTERS ( words : line) : integer; {returns an integer}
var      loop_count : integer;                   {local variable}
begin
      loop_count := 1;
      while (words[loop_count] <> '.') and (loop_count <= maxsize) do
              loop_count := loop_count + 1;
      COUNTLETTERS := loop_count - 1
end;

var   oneline : line;
      letters : integer;
begin
      writeln('Please enter in a sentence terminated with a .');
      readln( oneline );
      letters := COUNTLETTERS( oneline );
      writeln('There are ',letters,' letters in that sentence.')
end.
```

Answer:

SELF TEST
Determine the output of the following program

```
program  function_time (input, output);
const    maxsize = 80;
type     line = packed array[1..maxsize] of char;

function COUNTLETTERS ( words : line) : integer; {returns an integer}
var      loop_count : integer;                   {local variable}
begin
      loop_count := 1;
      while (words[loop_count] <> '.') and (loop_count <= maxsize) do
              loop_count := loop_count + 1;
      COUNTLETTERS := loop_count - 1
end;

var   oneline : line;
      letters : integer;
begin
      writeln('Please enter in a sentence terminated with a .');
      readln( oneline );
      letters := COUNTLETTERS( oneline );
      writeln('There are ',letters,' letters in that sentence.')
```

```
        end.


        Please enter in a sentence terminated with a .
        Hello there.
        There are 11 letters in that sentence.
```

PROGRAM SEVENTEEN

Write a program to calculate the cube of a given number (answer = number*number*number). Use a function to calculate the cube.

Answer:

PROGRAM SEVENTEEN

Write a program to calculate the cube of a given number (answer = number*number*number). Use a function to calculate the cube.

```
        program PROG17 (input,output);   {cube program using a function}

        function CUBE( x : integer ) : integer;
        begin
             CUBE := x * x * x
        end;

        var number, answer : integer;
        begin
            writeln('Enter integer to be cubed.');
            readln( number );
            answer := CUBE ( number );
            writeln('The cube of ',number,' is ', answer)
        end.
```

SELF TEST

1. Which of the following Pascal functions which change the value 6.6 to an integer value of 7

```
        odd
        round
        trunc
        abs
```

2. Which of the following Pascal operators has the least priority

```
        =
        +
        /
        NOT
```

3. Write a simple Pascal procedure called *Welcome* which prints the text string "Welcome to Pascal"

4. Write a Pascal procedure called *Multiply*, which accepts two integers, *number1* and *number2*, and prints the result of multiplying the two integers together

5. What is the output of the following Pascal program

```
program Sample( output );
var  x, y : integer;

procedure godoit( x, y : integer );
begin
        x := y;  y := 0;
        writeln( x, y );
end;

begin
        x := 1; x := 2;
        godoit( x, y );
        writeln( x, y )
end.
```

6. Write a Pascal function called *Multiply2* which returns an integer result. The function accepts two integer parameters, *number1* and *number2* and returns the value of multiplying the two parameters

Answer:

SELF TEST: ANSWERS

1. Which of the following Pascal functions which change the value 6.6 to an integer value of 7

```
odd
round
trunc
abs
```

2. Which of the following Pascal operators has the least priority

```
=
+
/
NOT
```

3. Write a simple Pascal procedure called *Welcome* which prints the text string "Welcome to Pascal"

```
procedure Welcome;
begin
        writeln('Welcome to Pascal')
end;
```

4. Write a Pascal procedure called *Multiply*, which accepts two integers, *number1* and *number2*, and prints the result of multiplying the two integers together.

```
procedure Multiply( number1, number2 : integer );
var  Result : integer;
begin
       Result := number1 * number2;
       writeln( Result )
end;
```

5. What is the output of the following Pascal program

```
program Sample( output );
var  x, y : integer;

procedure godoit( x, y : integer );
begin
       x := y;  y := 0;
       writeln( x, y );
end;

begin
       x := 1; y := 2;
       godoit( x, y );
       writeln( x, y )
end.
```

```
  Program Output
    2  0
    1  2
```

6. Write a Pascal function called *Multiply2* which returns an integer result. The function accepts two integer parameters, *number1* and *number2* and returns the value of multiplying the two parameters

```
function Multiply2( number1, number2 : integer ) : integer;
var Result : integer;
begin
       Result := number1 * number2;
       Multiply2 := Result
end;
```

RECORDS

A record is a user defined data type suitable for grouping data elements together. All elements of an array must contain the same data type.

A record overcomes this by allowing us to combine different data types together. Suppose we want to create a data record which holds a student name and mark. The student name is a packed array of characters, and the mark is an integer.

We could use two seperate arrays for this, but a record is easier. The method to do this is,

- define or declare what the new data group (record) looks like
- create a working variable to be of that type

The following portion of code shows how to define a record, then create a working variable to be of the same type.

```
        TYPE   studentname = packed array[1..20] of char;
               studentinfo = RECORD
                               name : studentname;
                               mark : integer
                            END;

        VAR    student1 : studentinfo;
```

The first portion defines the composition of the record identified as *studentinfo*. It consists of two parts (called **fields**).

The first part of the record is a packed character array identified as *name*. The second part of *studentinfo* consists of an integer, identified as *mark.*

The declaration of a record begins with the keyword **record**, and ends with the keyword **end;**

The next line declares a working variable called *student1* to be of the same type (ie composition) as *studentinfo.*

Each of the individual fields of a record are accessed by using the format,

```
      recordname.fieldname := value or variable;
```

An example follows,

```
      student1.name := 'JOE BLOGGS          ';  {20 characters}
      student1.mark := 57;
```

Lets create a new data record suitable for storing the date

```
        type  date = RECORD
                        day   : integer;
                        month : integer;
                        year  : integer
                     END;
```

This declares a **NEW data type** called *date*. This *date* record consists of three basic data elements, all integers. Now declare working variables to use in the program. These variables will have the same composition as the *date* record.

```
        var   todays_date : date;
```

defines a variable called *todays_date* to be of the same data type as that of the newly defined record *date*.

ASSIGNING VALUES TO RECORD ELEMENTS

These statements assign values to the individual elements of the record *todays_date*,

```
        todays_date.day   := 21;
        todays_date.month := 07;
        todays_date.year  := 1985;
```

NOTE the use of the **.fieldname** to reference the individual fields within *todays_date*.

SELF TEST

What does this statement do?

```
readln( todays_date.day, todays_date.month, todays_date.year );
```

Answer:

SELF TEST

What does this statement do?

```
readln( todays_date.day, todays_date.month, todays_date.year );
```

```
Self Test ..
The program statement reads three values from the keyboard,
into each of the individual fields of the record todays_date.
```

Sample program illustrating records

```
program   RECORD_INTRO (output);
type  date = record
                month, day, year : integer
             end;
var   today : date;

begin
      today.day    :=   25;
      today.month  :=   09;
      today.year   := 1983;
      writeln('Todays date is ',today.day,':',today.month,':',
               today.year)
end.
```

Records of the same type are assignable.

```
var  todays_date, tomorrows_date : date;
begin
     todays_date.day   := 9;
     todays_date.month := 7;
     todays_date.year  := 1976;
     tomorrows_date    := todays_date;
```

The last statement copies all the elements of *todays_date* into the elements of *tomorrows_date*.

This statement adds one to the value stored in the field *day* of the record *tomorrows_date*.

```
tomorrows_date.day := tomorrows_date.day + 1;
```

PROGRAM EIGHTEEN

Write a program that prompts the user for todays date, a procedure using variable parameters which calculates tomorrows date, and the main program displaying tommorrows date.

Use records for todays date, tomorrows date, An array can be used to hold the days for each month of the year.

```
Jan to Dec = 31,28,31,30,31,30,31,31,30,31,30,31
```

Remember to change the month or year as necessary.

Answer:

PROGRAM EIGHTEEN
Write a program that prompts the user for todays date, a procedure using variable parameters which calculates tomorrows date, and the main program displaying tommorrows date.

Use records for todays date, tomorrows date, An array can be used to hold the days for each month of the year.

```
        Jan to Dec = 31,28,31,30,31,30,31,31,30,31,30,31
```

Remember to change the month or year as necessary.

```
        program PROG18 (input,output);   {date calculation program}
        type    date = record
                         day, month, year : integer;
                       end;
                datename = array[1..12] of integer;

        procedure update( var tomorrow : date; days_in_month : datename );
        begin
             tomorrow.day := tomorrow.day + 1;                 {increment day}
             if tomorrow.day > days_in_month[tomorrow.month] then
             begin
                tomorrow.day := 1;
                tomorrow.month := tomorrow.month + 1;          {adjust month }
                if tomorrow.month > 12 then                       {adjust year  }
                begin
                  tomorrow.month := 1;
                  tomorrow.year := tomorrow.year + 1
                end
            end
        end;

        var  todays_date : date;
             days : datename;
        begin
             days[1] := 31;  days[2] := 28;  days[3] := 31;  days[4] := 30;
             days[5] := 31;  days[6] := 30;  days[7] := 31;  days[8] := 31;
             days[9] := 30; days[10] := 31; days[11] := 30; days[12] := 31;

             writeln('Enter todays date dd mm yy ');
             readln( todays_date.day, todays_date.month, todays_date.year);
             update( todays_date, days );
             writeln('Tomorrows date will be ', todays_date.day,'-',
                      todays_date.month,'-',todays_date.year)
        end.
```

RECORDS AND PROCEDURES
The following program demonstrates passing a record to a procedure, which updates the record, then prints the updated time.

```
        program  TIME (input,output);
        type     time = record
                          seconds, minutes, hours : integer
                        end;
        var  current, next : time;

        { function to update time by one second }
```

```
procedure  timeupdate( var now : time); {variable parameter}
var  newtime : time;                    {local variable}
begin
     newtime := now;                    {use local instead of orginal}
     newtime.seconds := newtime.seconds + 1;

     if  newtime.seconds = 60  then
     begin
        newtime.seconds := 0;
        newtime.minutes := newtime.minutes + 1;
        if  newtime.minutes = 60  then
        begin
             newtime.minutes := 0;
             newtime.hours := newtime.hours + 1;
             if  newtime.hours = 24  then
                newtime.hours := 0
        end
     end;
     writeln('The updated time is ',newtime.hours,':',newtime.minutes,
         ':',newtime.seconds)
end;

begin
      writeln('Please enter in the time using hh mm ss');
      readln( current.hours, current.minutes, current.seconds );
      timeupdate( current )
end.
```

ARRAYS OF RECORDS

can also be created, in the same way as arrays of any of the four basic data types. The following statement declares a record called *date*.

```
type  date = record
         month, day, year : integer
      end;
```

Lets now create an array of these records, called birthdays.

```
var   birthdays : array[1..10] of date;
```

This creates an array of 10 elements. Each element consists of a record of type date, ie, each element consists of three integers, called month, day and year. Pictorially it looks like,

```
|----------------|
|     month      | <----<----------------
|----------------|      |                |
|      day       |      |--Element 1     |
|----------------|      |                |
|     year       | <----                 |
|----------------|                       |
|     month      | <----                 |
|----------------|      |                |
|      day       |      |--Element 2     |
|----------------|      |                |--< birthdays
|     year       | <----                 |
|----------------|                       |
                                         |
|----------------|                       |
|     month      | <----                 |
|----------------|      |                |
|      day       |      |--Element 10    |
|----------------|      |                |
|     year       | <----<----------------
|----------------|
```

Consider the following assignment statements.

```
birthdays[1].month :=    2;
birthdays[1].day   :=   12;
birthdays[1].year  := 1983;
birthdays[1].year  := birthdays[2].year;
```

which assign various values to the array elements.

RECORDS CONTAINING ARRAYS

Records can also contain arrays as a field. Consider the following example, which shows a record called *month*, whose element *name* is actually an array.

```
type  monthname = packed array[1..4] of char;
      month = RECORD
                 days : integer;
                 name : monthname
              END;
var  this_month : month;
     this_month.days    := 31;  this_month.name[0] := 'J';
     this_month.name[1] := 'a'; this_month.name[2] := 'n';
     this_month.name := 'Feb ';
```

CLASS EXERCISE

- Determine the program output
- Draw a table illustrating the memory contents of array *test_times* after initilisation.

```
program  RECORD_TEST (output);
type     time = RECORD
                    hours, minutes, seconds : integer
                END;

procedure timeupdate ( var newtime : time );
begin
     newtime.seconds := newtime.seconds + 1;
     if  newtime.seconds = 60  then
     begin
        newtime.seconds := 0;
        newtime.minutes := newtime.minutes + 1;
        if  newtime.minutes = 60  then
        begin
            newtime.minutes := 0;
            newtime.hours := newtime.hours + 1;
            if  newtime.hours = 24  then
                newtime.hours := 0
         end
     end
end;

var  test_times : array [1..3] of time;
     loop : integer;
begin
     test_times[1].hours   := 11;
     test_times[1].minutes := 59;
     test_times[1].seconds := 59;
     test_times[2].hours   := 12;
     test_times[2].minutes :=  0;
     test_times[2].seconds :=  0;
     test_times[3].hours   :=  1;
     test_times[3].minutes := 29;
     test_times[3].seconds := 59;
     for loop := 1 to 3 do
     begin
        writeln('Time is ',test_times[loop].hours,':',
                 test_times[loop].minutes,':',test_times[loop].seconds);
        timeupdate(test_times[loop]);
        write('One second later its ');
        writeln(test_times[loop].hour,s':',test_times[loop].minutes,
                ':',test_times[loop].seconds)
     end
end.
```

Answer:

CLASS EXERCISE

- Determine the program output
- Draw a table illustrating the memory contents of array *test_times* after initilisation.

```
program  RECORD_TEST (output);
type     time = RECORD
                    hours, minutes, seconds : integer
                END;

procedure timeupdate ( var newtime : time );
begin
     newtime.seconds := newtime.seconds + 1;
     if  newtime.seconds = 60  then
     begin
        newtime.seconds := 0;
        newtime.minutes := newtime.minutes + 1;
        if  newtime.minutes = 60  then
        begin
            newtime.minutes := 0;
            newtime.hours := newtime.hours + 1;
            if  newtime.hours = 24  then
                newtime.hours := 0
         end
     end
end;

var  test_times : array [1..3] of time;
     loop : integer;
begin
     test_times[1].hours   := 11;
     test_times[1].minutes := 59;
     test_times[1].seconds := 59;
     test_times[2].hours   := 12;
     test_times[2].minutes :=  0;
     test_times[2].seconds :=  0;
     test_times[3].hours   :=  1;
     test_times[3].minutes := 29;
     test_times[3].seconds := 59;
     for loop := 1 to 3 do
     begin
        writeln('Time is ',test_times[loop].hours,':',
                 test_times[loop].minutes,':',test_times[loop].seconds);
```

```
          timeupdate(test_times[loop]);
          write('One second later its ');
          writeln(test_times[loop].hour,s':',test_times[loop].minutes,
                  ':',test_times[loop].seconds)
       end
end.
```

```
Class Exercise..Program output is,
Time is 11:59:59
One second later its 12:0:0
Time is 12:0:0
One second later its 12:0:1
Time is 1:29:59
One second later its 1:30:0
```

Table illustrating array test_times contents after initialisation,

```
|---------------|<----------  ------------
|      11       | hours     |              |
|---------------|           |              |
|      59       | minutes   |- Element 1   |
|---------------|           |              |
|      59       | seconds   |              |
|---------------|<----------               |
|      12       | hours     |              |
|---------------|           |              |
|      00       | minutes   |- Element 2   |-- test_times
|---------------|           |              |
|      00       | seconds   |              |
|---------------|<----------               |
|      01       | hours     |              |
|---------------|           |              |
|      29       | minutes   |- Element 3   |
|---------------|           |              |
|      59       | seconds   |              |
|---------------|<----------  ------------
```

RECORDS WITHIN RECORDS

Records can also contain other records as a field. Consider where both a *date* and *time* record are combined into a single record called *date_time*, eg,

```
type  date = RECORD
                day, month, year : integer
             END;
      time = RECORD
                hours, minutes, seconds : integer
             END;
      date_time = RECORD
                     sdate : date;
                     stime : time
                  END;
```

This defines a record whose elements consist of two other previously declared records. The statement

```
var   today  : date_time;
```

declares a working variable called *today*, which has the same composition as the record *date_time*. The statements

```
        today.sdate.day     :=   11;
        today.sdate.month   :=    2;
        today.sdate.year    := 1985;
        today.stime.hours   :=    3;
        today.stime.minutes :=    3;
        today.stime.seconds :=   33;
```

sets the *sdate* element of the record *today* to the eleventh of february, 1985. The *stime* element of the record is initialised to three hours, three minutes, thirty-three seconds.

with RECORDS

The *with* statement, in association with records, allows a quick and easy way of accessing each of the records members without using the dot notation.

Consider the following program example, where the variable *student* record is initialised. Note how the name of the record is associated with each of the initialised parts. Then look at the code that follows, and note the difference being the absence of the record name.

```
        program withRecords( output );

        type    Gender = (Male, Female);
                Person = Record
                        Age : Integer;
                        Sex : Gender
                end;

        var Student : Person;

        begin
                Student.Age := 23;
                Student.Sex := Male;

                with Student do begin
                        Age := 19;
                        Sex := Female
                end;

                with Student do begin
                        Writeln( 'Age := ', Age );
                        case Sex of
                                Male   : Writeln( 'Sex := Male' );
                                Female : Writeln( 'Sex := Female' )
                        end
                end
        end.
```

SETS

Sets exist in every day life. They are a way of classifying common types into groups. In Pascal, we think of sets as containing a range of limited values, from an initial value through to an ending value.

Consider the following set of integer values,

```
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

This is a set of numbers (integers) whose set value ranges from 1 to 10. To define this as a set type in Pascal, we would use the following syntax.

```
        program SetsOne( output );

        type  numberset = set of 1..10;

        var  mynumbers : numberset;

        begin
        end.
```

The statement

```
        type  numberset = set of 1..10;
```

declares a new type called *numberset*, which represents a set of integer values ranging from 1 as the lowest value, to 10 as the highest value. The value *1..10* means the numbers 1 to 10 inclusive. We call this the **base set**, that is, the set of values from which the set is taken.

The base set is a range of limited values. For example, we can have a *set of char*, but not a *set of integers*, because the set of integers has too many possible values, whereas the set of characters is very limited in possible values.

The statement

```
        var  mynumbers : numberset;
```

makes a working variable in our program called *mynumbers*, which is a set and can hold any value from the range defined in *numberset*.

SET OPERATIONS

The typical operations associated with sets are,

- assign values to a set
- determine if a value is in one or more sets
- set addition (UNION)
- set subtraction (DIFFERENCE)
- set commonality (INTERSECTION)

Assigning Values to a set: UNION
Set union is essentially the addition of sets, which also includes the initialisation or assigning of values to a set.

Consider the following statement which assigns values to a set

```
program SetsTWO( output );

type  numberset = set of 1..10;

var  mynumbers : numberset;

begin
        mynumbers := [];
        mynumbers := [2..6]
end.
```

The statement

```
mynumbers := [];
```

assigns an empty set to *mynumbers*. The statement

```
mynumbers := [2..6];
```

assigns a **subset** of values (integer 2 to 6 inclusive) from the range given for the set type *numberset*. Please note that assigning values outside the range of the set type from which *mynumbers* is derived will generate an error, thus the statement

```
mynumbers := [6..32];
```

is illegal, because *mynumbers* is derived from the base type *numberset*, which is a set of integer values ranging from 1 to 10. Any values outside this range are considered illegal.

Determining if a value is in a set
Lets expand the above program example to demonstrate how we check to see if a value resides in a set. Consider the following program, which reads an integer from the keyboard and checks to see if its in the set.

```
program SetsTHREE( input, output );

type  numberset = set of 1..10;

var  mynumbers : numberset;
     value : integer;

begin
        mynumbers := [2..6];
        value := 1;
        while( value <> 0 ) do
        begin
                writeln('Please enter an integer value, (0 to exit)');
                readln( value );
                if value <> 0 then
                begin
                        if value IN mynumbers then
                                writeln('Its in the set')
                        else
                                writeln('Its not in the set')
                end
        end
end.
```

More on set UNION, combining sets

Lets now look at combining some sets together. Consider the following program, which creates two sets, then joins the sets together to create another.

```
program SetsUNION( input, output );

type  numberset = set of 1..40;

var  mynumbers, othernumbers, unionnumbers : numberset;
     value : integer;

begin
        mynumbers := [2..6];
        othernumbers := [4..10];
        unionnumbers := mynumbers + othernumbers + [14..20];
        value := 1;
        while( value <> 0 ) do
        begin
                writeln('Please enter an integer value, (0 to exit)');
                readln( value );
                if value <> 0 then
                begin
                        if value IN unionnumbers then
                                writeln('Its in the set')
                        else
                                writeln('Its not in the set')
                end
        end
end.
```

The statement

```
        var  mynumbers, othernumbers, unionnumbers : numberset;
```

declares three sets of type *numberset*.

The statement

```
                mynumbers := [2..6];
```

assigns a **subset** of values (integer 2 to 6 inclusive) from the range given for the set type *numberset*.

The statement

```
                othernumbers := [4..10];
```

assigns a **subset** of values (integer 4 to 10 inclusive) from the range given for the set type *numberset*.

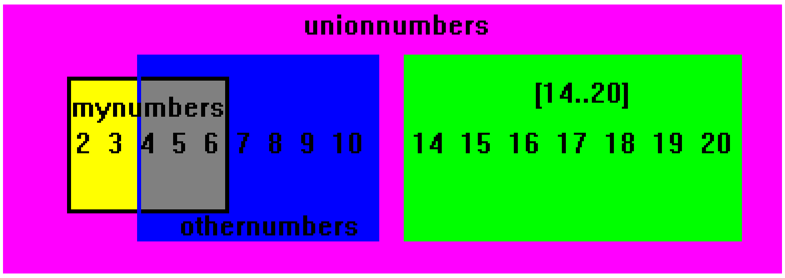
The statement

```
                unionnumbers := mynumbers + othernumbers + [14..20];
```

assigns the set of values in *mynumbers*, *othernumbers* and the set of values of 14 to 20 to *unionnumbers*.

If a specific value occurs in more than one set (as is the case of 4, 5, and 6, which are in *mynumbers* and *othernumbers*), then the other duplicate value is ignored (ie, only one instance of the value is copied to the new set.

This means that *unionnumbers* contains the values

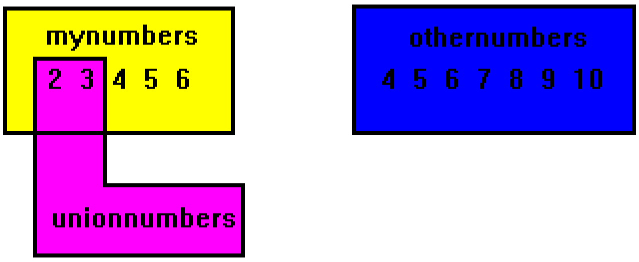


Set Subtraction, DIFFERENCE

In this operation, the new set will contain the values of the first set that are NOT also in the second set.

```
program SetsDIFFERENCE( input, output );

type  numberset = set of 1..40;

var  mynumbers, othernumbers, unionnumbers : numberset;
     value : integer;

begin
        mynumbers := [2..6];
        othernumbers := [4..10];
        unionnumbers := mynumbers - othernumbers;
        value := 1;
        while( value <> 0 ) do
        begin
                writeln('Please enter an integer value, (0 to exit)');
                readln( value );
                if value <> 0 then
                begin
                        if value IN unionnumbers then
                                writeln('Its in the set')
                        else
                                writeln('Its not in the set')
                end
        end
end.
```

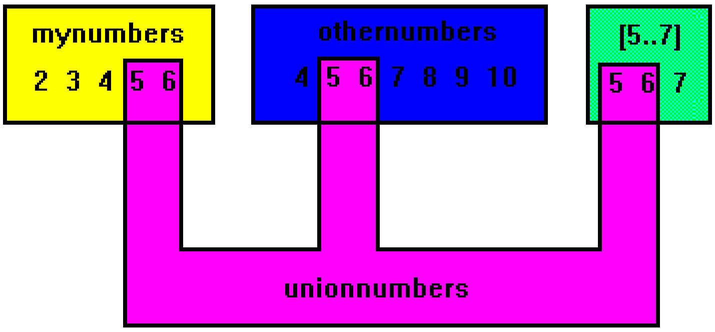
unionnumbers contains the values



Set Commonality, INTERSECTION
In this operation, the new set will contain the values which are common (appear as members) of the specified sets.

```
program SetsINTERSECTION( input, output );

type  numberset = set of 1..40;

var  mynumbers, othernumbers, unionnumbers : numberset;
     value : integer;

begin
        mynumbers := [2..6];
        othernumbers := [4..10];
        unionnumbers := mynumbers * othernumbers * [5..7];
        value := 1;
        while( value <> 0 ) do
        begin
                writeln('Please enter an integer value, (0 to exit)');
                readln( value );
                if value <> 0 then
                begin
                        if value IN unionnumbers then
                                writeln('Its in the set')
                        else
                                writeln('Its not in the set')
                end
        end
end.
```

unionnumbers contains the values



FILE HANDLING
So far, data has been inputted from the keyboard, and outputted to the console screen.

The keyboard is known as the standard input device, and the console screen is the standard output device. Pascal names these as INPUT and OUTPUT respectively.

Occasions arise where data must be derived from another source other than the keyboard. This data will exist external to the program, either stored on diskette, or derived from some hardware device.

In a lot of cases, hardcopy (a printout) of program results is needed, thus the program will send the output to either the printer or the disk instead of the screen.

A program which either reads information from, or writes information to, a place on a disk, is performing

FILE Input/Output (I/O).

A File is a collection of information. In Pascal, this information may be arranged as text (ie a sequence of characters), as numbers (a sequence of integers or reals), or as records. The information is collectively known by a sequence of characters, called a **FILENAME**.

You have already used filenames to identify the source programs written and used in this tutorial.

USING A FILE IN PASCAL

Files are referred to in Pascal programs by the use of filenames. You have already used two default filenames, input and output. These are associated with the keyboard and console screen. To derive data from another source, it must be specified in the program heading, eg,

```
        program  FILE_OUTPUT( input, fdata );
```

This informs Pascal that you will be using a file called *fdata*. Within the variable declaration section, the file type is declared, eg

```
        var  fdata : file of char;
```

This declares the file *fdata* as consisting of a sequence of characters. Pascal provides a standard definition called **TEXT** for this, so the following statement is identical,

```
        var   fdata : TEXT;
```

BASIC FILE OPERATIONS

Once the file is known to the program, the operations which may be performed are,

1. The file is prepared for use by RESET or REWRITE
2. Information is read or written using READ or WRITE
3. The file is then closed by using CLOSE

PREPARING A FILE READY FOR USE

The two commands for preparing a file ready for use in a program are **RESET** and **REWRITE**. Both procedures use the name of the file variable you want to work with. They also accept a string which is then associated with the file variable, eg

```
        var  filename : string[15];

             readln( filename );
```

- **RESET (fdata, filename);**
  This prepares the file specified by filename for reading. All reading operations are performed using fdata.
- **REWRITE (fdata, filename);**
  This prepares the file specified by filename for writing. All write operations are performed using fdata. If the file already exists, it is re-created, and all existing information lost!

READING AND WRITING TO A FILE OF TYPE TEXT

The procedures READ and WRITE can be used. These procedures also accept the name of the file, eg,

```
        writeln( fdata, 'Hello there. How are you?');
```

writes the text string to the file *fdata* rather than the standard output device.

Turbo Pascal users must use the **assign** statement, as only one parameter may be supplied to either reset or rewrite.

```
assign( fdata, filename );
reset(  fdata );
rewrite( fdata );
```

CLOSING A FILE

When all operations are finished, the file is closed. This is necessary, as it informs the program that you have finished with the file. The program releases any memory associated with the file, ensuring its (the files) integrity.

```
CLOSE( fdata );         {closes file associated with fdata}
```

Once a file has been closed, no further file operations on that file are possible (unless you prepare it again).

SAMPLE FILE OUTPUT PROGRAM TO WRITE DATA TO A TEXT FILE

```
program  WRITETEXT (input, output, fdata );
var  fdata : TEXT;
     ch    : char;
     fname : packed array [1..15] of char;
begin
     writeln('Enter a filename for storage of text.');
     readln( fname );
     rewrite( fdata, fname );        {create a new fdata          }
     readln;                         {clear input buffer          }
     read( ch );                     {read character from keyboard}
     while  ch <> '*' do             {stop when an * is typed     }
     begin
        write( fdata, ch );          {write character to fdata    }
        read( ch )                   {read next character         }
     end;
     write( fdata, '*');             {write an * for end of file  }
     close( fdata )                  {close file fdata            }
end.
```

SELF TEST

Determine what the following code statements do

```
writeln( output, 'Hello there. It''s me again');
writeln('The time has come, the Walrus said,');
readln( input, ch );
readln( ch );
```

Answer:

SELF TEST

Determine what the following code statements do

```
writeln( output, 'Hello there. It''s me again');
writeln('The time has come, the Walrus said,');
readln( input, ch );
readln( ch );


Self Test .. File statements
Both writeln statements display info on screen
Both readln statements accept info from keyboard
```

THE COMPOSITION OF TEXT FILES

Text files are arranged as a sequence of variable length lines.

- Each line consists of a sequence of characters.
- Each line is terminated with a special character, called
  END-OF-LINE (EOLN)
- The last character is another special character, called
  END-OF-FILE (EOF)

THIS IS WHAT A TEXT FILE LOOKS LIKE

```
He was not quite as old as people estimated. In fact, the furrowedEOLN
brow that swept many a street was only fourty-five.EOLN
Life had not been easy for the hunchback, it's difficult to playEOLN
any game when all you can see are your feet. In spite of theEOLN
hardships, he was as gentle as a roaring elephant going overEOLN
Niagara falls.EOF
```

End of File and End of Line

EOF

Accepts the name of the input file, and returns true if there is no more data to be read.

EOLN

Accepts the name of the input file, and is true if there are no more characters on the current line.

When reading information from a text file, the character which is read can be compared against EOLN or EOF. Consider the following program which displays the contents of a text file on the console screen.

```
program  SHOWTEXT ( infile, input, output );
var   ch : char;
    fname : packed array [1..15] of char;
    infile: TEXT;
begin
    writeln('Please enter name of text file to display.');
    readln( fname );

    reset( infile, fname );     {open a file using filename stored in}
                                {array fname                         }
    while not eof( infile ) do
    begin
       while not eoln( infile ) do
       begin
           read( infile, ch );
           write( ch )
       end;
       readln( infile );       {read  eoln character}
       writeln                 {write eoln character}
    end;
    close( infile )             {close filename specified by fname}
end.
```

PROGRAM TWENTY-ONE

Write a program to count the number of characters in a text file. The valid characters are 'A' to 'Z', and 'a' to 'z'.

Answer:

PROGRAM TWENTY-ONE

Write a program to count the number of characters in a text file. The valid characters are 'A' to 'Z', and 'a' to 'z'.

```
        program PROG21 (input,output, infile);   {count characters in file}
        type    legal1 = 'A'..'Z';
                legal2 = 'a'..'z';
        var     infile : TEXT;
                fname : string[15];
                ch : char;
                count : integer;
        begin
            count := 0;
            writeln('Please enter name of text file to count.');
            readln( fname );
            { for turbo pascal
              assign( infile, fname );
              reset( infile );
            }

            reset( infile, fname );     {open a file using filename stored in}
                                        {array fname                         }
            while not eof( infile ) do
            begin
                while not eoln( infile ) do
                begin
                    read( infile, ch );
                    if ((ch>='A')and(ch<='Z'))or((ch >='a')and(ch<='z')) then
                       count := count + 1
                end;
                readln( infile )         {read  eof character}
            end;
            close( infile );             {close filename specified by fname}
            writeln('The number of characters in ',fname,' is ',count)
        end.
```

PROGRAM TWENTY-TWO

Write a program to count the number of words in a text file.

Answer:

PROGRAM TWENTY-TWO

Write a program to count the number of words in a text file.

```
{ Program to count words in a text file. Adapted from C program found}
{ in  Programming in C : S Kochan, pg 174 -                          }
program PROG22 (input, output, infile );
type    oneline = packed array[1..81] of char;

{ a function to determine if a character is alphabetic }
function alphabetic ( ch : char ) : boolean;
begin
    if ( ((ch >= 'a') AND (ch <= 'z')) OR ((ch >= 'A') AND (ch <= 'Z')) ) then
       alphabetic := TRUE
    else
       alphabetic := FALSE
end;

{ a function to count the number of words in a string }
function count_words ( var line : oneline ) : integer;
var  i, word_count : integer;
     looking_for_word : boolean;
begin
    looking_for_word := TRUE;
    word_count := 0;
    for i := 1 to 81 do
    begin
        if alphabetic( line[i] ) then
        begin
           if looking_for_word then
           begin
               word_count := word_count + 1;
               looking_for_word := FALSE
           end
        end
        else
           looking_for_word := TRUE
    end;
    count_words := word_count
end;

var  infile : text;
     tline  : oneline;
     fname  : string[15];
     total,count  : integer;
     ch           : char;
begin
     total := 0;
     writeln('Please enter name of input file to count');
     readln (fname);
     assign (infile, fname);
     reset( infile);
     while not eof(infile) do
     begin
         for count := 1 to 81 do
             tline[count] := ' ';
         count := 1;
         while not eoln(infile) do
         begin
             read(infile, ch );
             tline[count] := ch;
             count := count + 1
         end;
         total := total + count_words( tline );
         readln(infile)      { read eoln character }
     end;
     writeln('There are ',total,' words in the text file.')
end.
```

FILES OF NUMBERS

Files may also consist of integers or reals. The procedures *read* and *write* can be used to transfer one value at a time.

The procedures *readln* and *writeln* cannot be used with file types other than text.

PROGRAM TWENTY-THREE

Write a program which adds up a list of numbers from a file. Create a sample file to test your program.
Answer:

PROGRAM TWENTY-THREE
Write a program which adds up a list of numbers from a file. Create a sample file to test your program.

```
{developed from a routine in OH PASCAL, pg 444                      }
program PROG23A (input,output,outfile ); {create a file of integers }
var outfile : file of integer;
    current, total : integer;
    fname : string[15];
begin
    total := 0;
    writeln('Enter name of file to contain numbers');
    readln (fname);
    assign( outfile, fname );
    rewrite( outfile );
    writeln('Enter in integers, a value of 0 stops');
    read( current );
    while current <> 0 do
    begin
       write( outfile, current);
       read( current )
    end;
    close( outfile )
end.
```

```
{developed from a routine in OH PASCAL, pg 444                       }
program PROG23 (input,output,infile );     {sum of integers in a file}
var infile : file of integer;
    current, total : integer;
    fname : string[15];
begin
    total := 0;
    writeln('Enter name of file containing numbers');
    readln (fname);
    assign( infile, fname );
    reset( infile );
    while not eof( infile ) do
    begin
       read( infile, current );
       total := total + current
    end;
    writeln('The sum of all numbers is ', total)
end.
```

FILES OF RECORDS
Files can also contain records. Using *read* or *write*, it is possible to transfer a record at a time.

PROGRAM TWENTY_FIVE
Implement a Pascal program which allows the recalling of a group of student marks. The program is to output the highest and lowest marks, as well as the mean.

Use an array of records to store the names and marks. Using an output file, sort the student names, marks into ascending order, so that the student with the highest mark will be written first.

The details are,

```
Student 1      Joe Bloggs      56
        2      Bill Anderson   24
        3      William Tell    78
        4      Bob Crane       23
        5      Peter Hall      57
        6      Charles French  76
        7      Bryan Goldwater 65
        8      Stewart Phelps  89
        9      Dave Stevens    78
       10      Ted Rosse       64
```

The student name consists of 16 characters, and the student mark is an integer in the range 0 to 100. Our example has a maximum of ten students.

Answer:

PROGRAM TWENTY_FIVE
Implement a Pascal program which allows the recalling of a group of student marks. The program is to output the highest and lowest marks, as well as the mean.

Use an array of records to store the names and marks. Using an output file, sort the student names, marks into ascending order, so that the student with the highest mark will be written first.

The details are,

```
Student 1      Joe Bloggs      56
        2      Bill Anderson   24
        3      William Tell    78
        4      Bob Crane       23
        5      Peter Hall      57
        6      Charles French  76
        7      Bryan Goldwater 65
        8      Stewart Phelps  89
        9      Dave Stevens    78
       10      Ted Rosse       64
```

The student name consists of 16 characters, and the student mark is an integer in the range 0 to 100. Our example has a maximum of ten students.

```
program prog25A (input,output,outfile);   {create student file}
const outname = 'STUDENT.DAT';
type  student = record
                 name : string[16];
                 mark : integer;
                end;
var class : array [1..10] of student;
    loopcount : integer;
    outfile : file of student;
begin
    class[1].name := 'Joe Bloggs      '; class[1].mark := 56;
    class[2].name := 'Bill Anderson   '; class[2].mark := 24;
    class[3].name := 'William Tell    '; class[3].mark := 78;
    class[4].name := 'Bob Crane       '; class[4].mark := 23;
    class[5].name := 'Peter Hall      '; class[5].mark := 57;
    class[6].name := 'Charles French  '; class[6].mark := 76;
    class[7].name := 'Bryan Goldwater '; class[7].mark := 65;
    class[8].name := 'Stewart Phelps  '; class[8].mark := 89;
    class[9].name := 'Dave Stevens    '; class[9].mark := 78;
    class[10].name := 'Ted Rosse       '; class[10].mark := 64;

{ for turbo pascal  assign( outfile, outname ); rewrite( outfile ); }
    rewrite( outfile, outname );
    for loopcount := 1 to 10 do
       write( outfile, class[loopcount] );
    writeln('Student.dat created and written.');
    close( outfile )
end.
```

```
program prog25B (input,output,infile);   {read back in student file}
const inname = 'STUDENT.DAT';
type  student = record
                 name : string[16];
                 mark : integer;
                end;
var class : array [1..10] of student;
    loopcount, classsize : integer;
    infile : file of student;
begin
{ for turbo pascal  assign( infile, inname );  reset( infile );   }
    rewrite( infile, inname );
    classsize := 1;
    while not eof(infile) do
    begin
       read( infile, class[classsize] );
       classsize := classsize + 1
    end;

    for loopcount := 1 to ( classsize - 1 )  do
    begin
      write('Student ',loopcount:2,'   is   ' );
      writeln(class[loopcount].name,'   ',class[loopcount].mark)
    end;
    close( infile )
end.
```

```
{read back, sort, write, student file}
program prog25C (input, output, infile, outfile);
const inname  = 'STUDENT.DAT';
      outname = 'STUDENT.SRT';
type  student = record
                 name : string[16];
                 mark : integer;
                end;
      class = array [1..10] of student;

{ find highest mark }
function gethighest(studclass : class; sizeclass : integer) : integer;
var temp, count : integer;
begin
    temp := studclass[1].mark;
    count := 2;
    while count <= sizeclass do
    begin
       if studclass[count].mark > temp then
          temp := studclass[count].mark;
       count := count + 1
    end;
    gethighest := temp;
end;

{ find lowest mark }
function getlowest(studclass : class; sizeclass : integer) : integer;
var temp, count : integer;
begin
    temp := studclass[1].mark;
    count := 2;
    while count <= sizeclass do
    begin
       if studclass[count].mark < temp then
          temp := studclass[count].mark;
       count := count + 1
    end;
    getlowest := temp;
end;

{ find mean }
function getmean ( studclass : class; sizeclass : integer ) : real;
var total, loop : integer;
begin
    total := 0;
    for loop := 1 to sizeclass do
        total := total + studclass[loop].mark;

    getmean := total / sizeclass;
end;

{ sort into ascending order, standard sequential sort used here }
procedure sort( var studclass : class; sizeclass : integer );
var temp : student;
    loop, base, index : integer;
begin
    base := 1;
    while  base < sizeclass   do
    begin
       index := base + 1;
       while  index <= sizeclass   do
       begin
          if studclass[base].mark < studclass[index].mark  then
          begin
             temp.mark := studclass[base].mark;
             temp.name := studclass[base].name;
             studclass[base].name := studclass[index].name;
             studclass[base].mark := studclass[index].mark;
             studclass[index].name := temp.name;
             studclass[index].mark := temp.mark
          end;
          index := index + 1
       end;
       base := base + 1
    end;
end;

var mainclass : class;
    loopcount, classsize, highest, lowest : integer;
    mean : real;
    infile, outfile : file of student;
begin
{ for turbo pascal    assign( infile, inname );
                      reset ( infile );
                      assign( outfile, outname);
                      rewrite(outfile);
}
    reset( infile, inname );
    rewrite(outfile, outname);
    classsize := 1;
    while not eof(infile) do
    begin
       read( infile, mainclass[classsize] );
       classsize := classsize + 1
    end;
    close( infile );

    { find highest, lowest and average marks }
    highest := gethighest( mainclass, classsize - 1 );
    lowest  := getlowest ( mainclass, classsize - 1 );
    mean    := getmean   ( mainclass, classsize - 1 );

    { now sort into ascending order }
    sort( mainclass, classsize - 1 );

    { now write out sorted class to outfile }
    for loopcount := 1 to ( classsize - 1 )  do
      write(outfile,mainclass[loopcount]);

    writeln('The highest mark was ', highest );
    writeln('The lowest  mark was ', lowest  );
    writeln('The mean    mark was ', mean:3:2);
    close( outfile )
end.
```

STRINGS

The following program illustrates using STRINGS (a sequence of characters) in a DG Pascal program. **STRING** is type defined as a packed array of type *char*.

Message is then declared as the same type as STRING, ie, a packed array of characters, elements numbered one to eight.

```
PROGRAM DGSTRING (INPUT, OUTPUT);
TYPE STRING = PACKED ARRAY [1..8] OF CHAR;
VAR  MESSAGE : STRING;
BEGIN
        WRITELN('HELLO BRIAN.');
        MESSAGE := '12345678';
        WRITELN('THE MESSAGE IS ', MESSAGE)
END.
```

Turbo Pascal, how-ever, allows an easier use of character strings by providing a new keyword called STRING. Using STRING, you can add a parameter (how many characters) specifying the string length. Consider the above program re-written for turbo pascal.

```
PROGRAM TPSTRING (INPUT, OUTPUT);
VAR  MESSAGE : STRING[8];
BEGIN
        WRITELN('HELLO BRIAN.');
        MESSAGE := '12345678';
        WRITELN('THE MESSAGE IS ', MESSAGE)
END.
```

Obviously, the turbo pascal version is easier to use. BUT, the following program shows a similar implementation for use on the DG.

```
PROGRAM DGSTRING2 (INPUT, OUTPUT);
CONST   $STRINGMAXLENGTH = 8;        {defines maxlength of a string}
%INCLUDE 'PASSTRINGS.IN';            {include code to handle strings}
VAR  MESSAGE : $STRING_BODY;
BEGIN
        WRITELN('HELLO BRIAN.');
        MESSAGE := '12345678';
        WRITELN('THE MESSAGE IS ', MESSAGE)
END.
```

Strings

DG Pascal also provides the following functions for handling and manipulating strings.

- APPEND
  concatenate two strings. calling format is
- `APPEND( string1, string2 );`
-
- where *string2* is added onto the end of *string1*.
-
- LENGTH
  returns a short_integer which represents the length (number of characters) of the string.
- `LENGTH( stringname );`

- SETSUBSTR
  replaces a substring in a target string with a substring from
  a source string.
- SETSUBSTR(Targetstr, tstart, tlen, Sourcestr, sstart);
-
- where
- *Targetstr* is the target string
-
- *tstart* is an integer representing the start position
- (within Targetstr) of the substring that is to be replaced
-
- *tlen* is an integer representing the length of the substring
- that you are replacing in *Targetstr*
-
- *Sourcestr* is the source string which contains the substring
-
- *sstart* is an integer which specifies the starting position
- of the substring within *Sourcestr*

POINTERS

Pointers enable us to effectively represent complex data structures, to change values as arguments to functions, to work with memory which has been dynamically allocated, and to store data in complex ways.

A pointer provides an indirect means of accessing the value of a particular data item. Lets see how pointers actually work with a simple example,

```
program pointers1( output );
type    int_pointer = ^integer;

var     iptr : int_pointer;
begin
        new( iptr );
        iptr^ := 10;
        writeln('the value is ', iptr^);
        dispose( iptr )
end.
```

The line

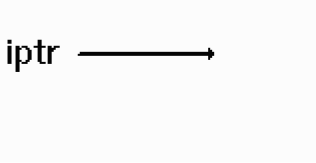
```
type    int_pointer = ^integer;
```

declares a new type of variable called *int_pointer*, which is a pointer (denoted by ^) to an integer.

The line

```
var     iptr : int_pointer;
```

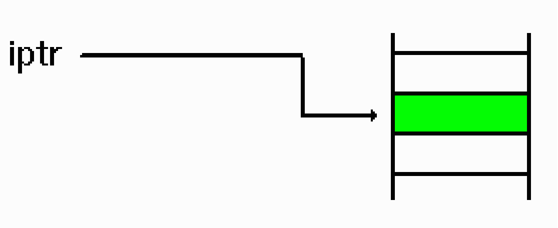
declares a working variable called *iptr* of type *int_pointer*. The variable *iptr* will not contain numeric values, but will contain the address in memory of a dynamically created variable (by using **new**). Currently, there is no storage space allocated with *iptr*, which means you cannot use it till you associate some storage space to it. Pictorially, it looks like,



The line

```
                new( iptr );
```

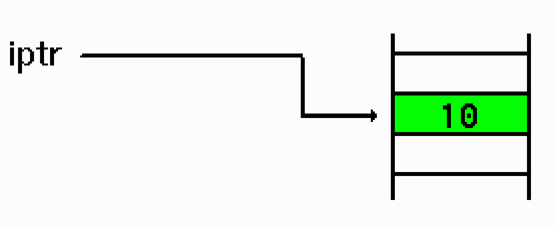
creates a new dynamic variable (ie, its created when the program actually runs on the computer). The pointer variable *iptr* points to the location/address in memory of the storage space used to hold an integer value. Pictorially, it looks like,



The line

```
                iptr^ := 10;
```

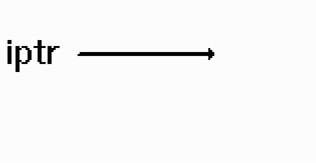
means go to the storage space allocated/associated with *iptr*, and write in that storage space the integer value 10. Pictorially, it looks like,



The line

```
                dispose( iptr )
```

means deallocate (free up) the storage space allocated/associated with *iptr*, and return it to the computer system. This means that *iptr* cannot be used again unless it is associated with another *new()* statement first. Pictorially, it looks like,



POINTERS

Pointers which do not reference any memory location should be assigned the value **nil**. Consider the following program, which expands on the previous program.

```
program pointers2( output );
type    int_pointer = ^integer;

var     iptr : int_pointer;
begin
        new( iptr );
        iptr^ := 10;
        writeln('the value is ', iptr^);
        dispose( iptr );
        iptr := nil;
        if iptr = nil
           then writeln('iptr does not reference any variable')
        else
           writeln('The value of the reference for iptr is ', iptr^)
end.
```

The line

```
iptr := nil;
```

assigns the value **nil** to the pointer variable *iptr*. This means that the pointer is valid and stil exists, but it does not point to any memory location or dynamic variable.

The line

```
if iptr = nil
```

tests *iptr* to see if its a *nil pointer*, ie, that it is not pointing to a valid reference. This test is very useful and will come in use later on when we want to construct more complex data types like linked lists.

POINTERS

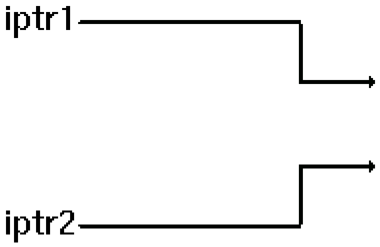
Pointers of the same type may be equated and assigned to each other. Consider the following program

```
program pointers3( output );
type    int_pointer = ^integer;

var     iptr1, iptr2 : int_pointer;
begin
        new( iptr1 );
        new( iptr2 );
        iptr1^ := 10;
        iptr2^ := 25;
        writeln('the value of iptr1 is ', iptr1^);
        writeln('the value of iptr2 is ', iptr2^);
        dispose( iptr1 );
        iptr1 := iptr2;
        iptr1^ := 3;
        writeln('the value of iptr1 is ', iptr1^);
        writeln('the value of iptr2 is ', iptr2^);
        dispose( iptr2 );
end.
```

The lines

```
                    new( iptr1 );
                    new( iptr2 );
```

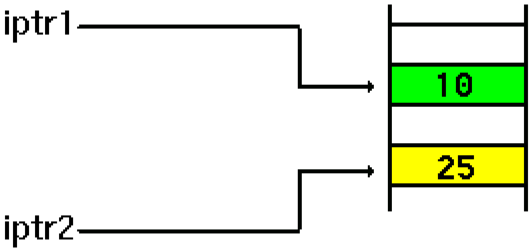
creates two integer pointers named *iptr1* and *iptr2*. They are not associated with any dynamic variables yet, so pictorially, it looks like,



The lines

```
                    iptr1^ := 10;
                    iptr2^ := 25;
```

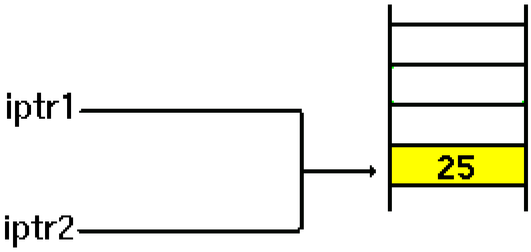
assigns dynamic variables to each of the integer variables. Pictorially, it now looks like,



The lines

```
                    dispose( iptr1 );
                    iptr1 := iptr2;
```

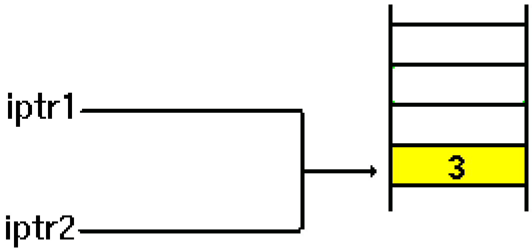
remove the association of *iptr1* from the dynamic variable whose value was 10, and the next line makes *iptr1* point to the same dynamic variable that *iptr2* points to. Pictorially, it looks like,



The line

```
                    iptr1^ := 3;
```

assigns the integer value 3 to the dynamic variable associated with *iptr1*. In effect, this also changes *iptr2^*. Pictorially, it now looks like,



The programs output is

```
        the value of iptr1 is 10
        the value of iptr2 is 25
        the value of iptr1 is 3
        the value of iptr2 is 3
```

SUMMARY OF POINTERS

- A pointer can point to a location of any data type, including records. Its basic syntax is,
- `type  Pointertype = ^datatype;`
- `var  NameofPointerVariable : Pointertype;`
- The procedure *new* allocates storage space for the pointer to use
- The procedure *dispose* deallocates the storage space associated with a pointer
- A pointer can be assigned storage space using *new*, or assigning it the value from a pointer of the same type (eg, iptr1 := iptr2;)
- A pointer can be assigned the value **nil**, to indicate that it is not pointing to any storage space
- The value at the storage space associated with a pointer may be read or altered using the syntax
- `NameofPointerVariable^`
- A pointer may reference a type which has not yet been created (this will be covered next)

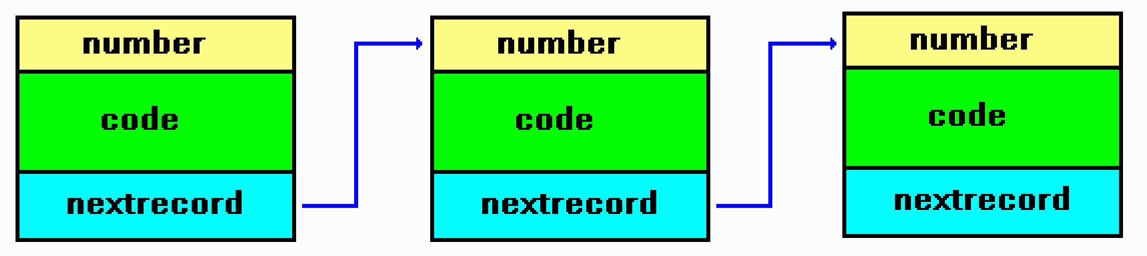
POINTERS: Referencing data types that do not yet exist

The most common use of pointers is to reference structured types like records. Often, the record definition will contain a reference to the pointer,

```
        type  rptr = ^recdata;
                recdata = record
                        number : integer;
                        code   : string;
                        nextrecord : rptr
                end;

        var  currentrecord : rptr;
```

In this example, the definition for the field *nextrecord* of *recdata* includes a reference to the pointer of type *iptr*. As you can see, *rptr* is defined as a pointer of type *recdata*, **which is defined on the next lines**. This is allowed in Pascal, for pointer types.

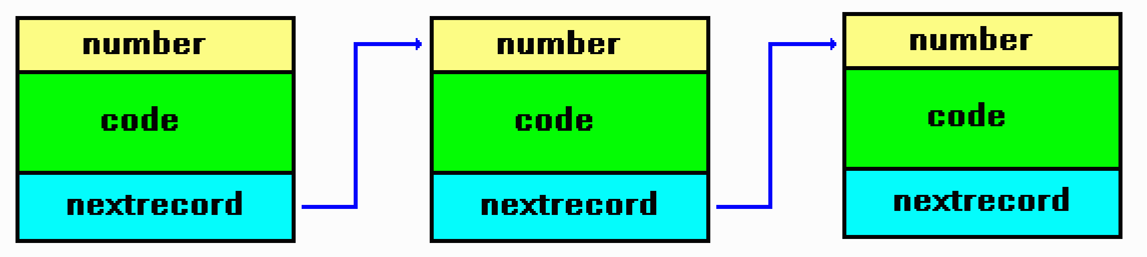
Using a definition of *recdata*, this will allow us to create a list of records, as illustrated by the following picture.



In this case, a list is simply of number of records (all of the same type), linked together by the use of pointers.

POINTERS

Lets construct the actual list as shown below, as an example.



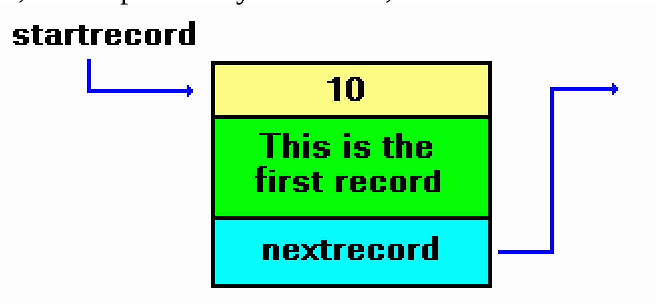
```
program PointerRecordExample( output );

type  rptr = ^recdata;
        recdata = record
                number : integer;
                code   : string;
                nextrecord : rptr
        end;

var  startrecord : rptr;

begin
        new( startrecord );
        if startrecord = nil then
        begin
                writeln('1: unable to allocate storage space');
                exit
        end;
        startrecord^.number := 10;
        startrecord^.code := 'This is the first record';
        new( startrecord^.nextrecord );
        if startrecord^.nextrecord = nil then
        begin
                writeln('2: unable to allocate storage space');
                exit
        end;
        startrecord^.nextrecord^.number := 20;
        startrecord^.nextrecord^.code := 'This is the second record';
        new( startrecord^.nextrecord^.nextrecord );
        if startrecord^.nextrecord^.nextrecord = nil then
        begin
                writeln('3: unable to allocate storage space');
                exit
        end;
```

```
                startrecord^.nextrecord^.nextrecord^.number := 30;
                startrecord^.nextrecord^.nextrecord^.code := 'This is the third
record';
                startrecord^.nextrecord^.nextrecord^.nextrecord := nil;
                writeln( startrecord^.number );
                writeln( startrecord^.code );
                writeln( startrecord^.nextrecord^.number );
                writeln( startrecord^.nextrecord^.code );
                writeln( startrecord^.nextrecord^.nextrecord^.number );
                writeln( startrecord^.nextrecord^.nextrecord^.code );
                dispose( startrecord^.nextrecord^.nextrecord );
                dispose( startrecord^.nextrecord );
                dispose( startrecord )
        end.
```

The lines of code

```
                new( startrecord );
                if startrecord = nil then
                begin
                        writeln('1: unable to allocate storage space');
                        exit
                end;
                startrecord^.number := 10;
                startrecord^.code := 'This is the first record';
```

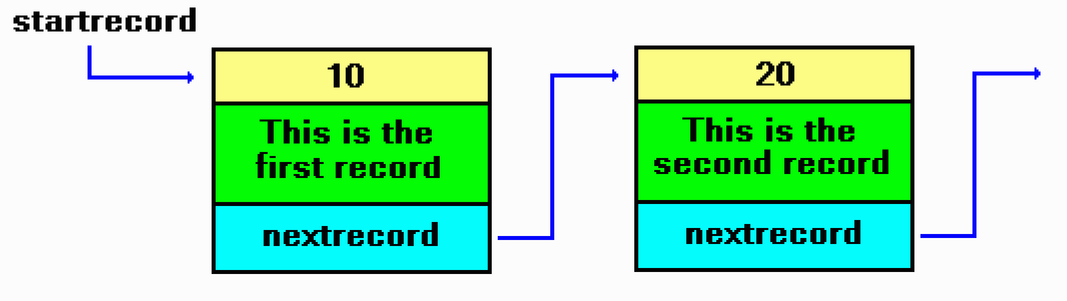
create the beginning of the list, which pictorially looks like,



The lines of code

```
                new( startrecord^.nextrecord );
                if startrecord^.nextrecord = nil then
                begin
                        writeln('2: unable to allocate storage space');
                        exit
                end;
                startrecord^.nextrecord^.number := 20;
                startrecord^.nextrecord^.code := 'This is the second record';
```

link in the next record, which now looks like,



The lines of code

```
                new( startrecord^.nextrecord^.nextrecord );
                if startrecord^.nextrecord^.nextrecord = nil then
                begin
```

```
                        writeln('3: unable to allocate storage space');
                        exit
                end;
                startrecord^.nextrecord^.nextrecord^.number := 30;
                startrecord^.nextrecord^.nextrecord^.code := 'This is the third
record';
                startrecord^.nextrecord^.nextrecord^.nextrecord := nil;
```

link in the third and final record, also setting the last *nextrecord* field to **nil**. Pictorially, the list now looks like,

The remaining lines of code print out the fields of each record.

POINTERS

The previous program can be rewritten to make it easier to read, understand and maintain. To do this, we will use a dedicated pointer to maintain and initialise the list, rather than get into the long notation that we used in the previous program, eg,

```
        startrecord^.nextrecord^.nextrecord^.number := 30;
```

The modified program now looks like,

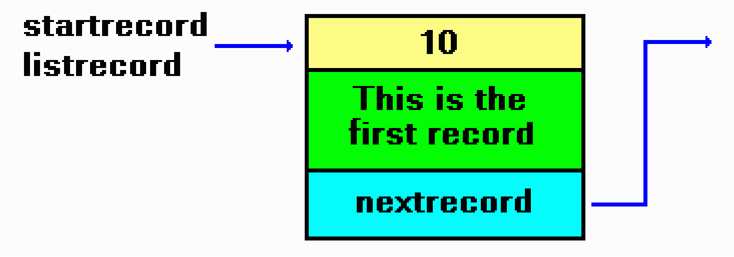
```
        program PointerRecordExample2( output );

        type  rptr = ^recdata;
                recdata = record
                        number : integer;
                        code   : string;
                        nextrecord : rptr
                end;

        var  startrecord, listrecord : rptr;

        begin
                new( listrecord );
                if listrecord = nil then
                begin
                        writeln('1: unable to allocate storage space');
                        exit
                end;
                startrecord := listrecord;
                listrecord^.number := 10;
                listrecord^.code := 'This is the first record';
                new( listrecord^.nextrecord );
                if listrecord^.nextrecord = nil then
                begin
                        writeln('2: unable to allocate storage space');
                        exit
                end;
                listrecord := listrecord^.nextrecord;
                listrecord^.number := 20;
                listrecord^.code := 'This is the second record';
                new( listrecord^.nextrecord );
                if listrecord^.nextrecord = nil then
                begin
                        writeln('3: unable to allocate storage space');
                        exit
                end;
                listrecord := listrecord^.nextrecord;
                listrecord^.number := 30;
                listrecord^.code := 'This is the third record';
                listrecord^.nextrecord := nil;
                while startrecord <> nil do
                begin
                        listrecord := startrecord;
                        writeln( startrecord^.number );
                        writeln( startrecord^.code );
                        startrecord := startrecord^.nextrecord;
                        dispose( listrecord )
                end
        end.
```

In this example, the pointer *listrecord* is used to create and initialise the list. After creation of the first record, it is saved in the pointer *startrecord*.

The lines of code

```
        new( listrecord );
        if listrecord = nil then
        begin
                writeln('1: unable to allocate storage space');
                exit
        end;
        startrecord := listrecord;
        listrecord^.number := 10;
        listrecord^.code := 'This is the first record';
```

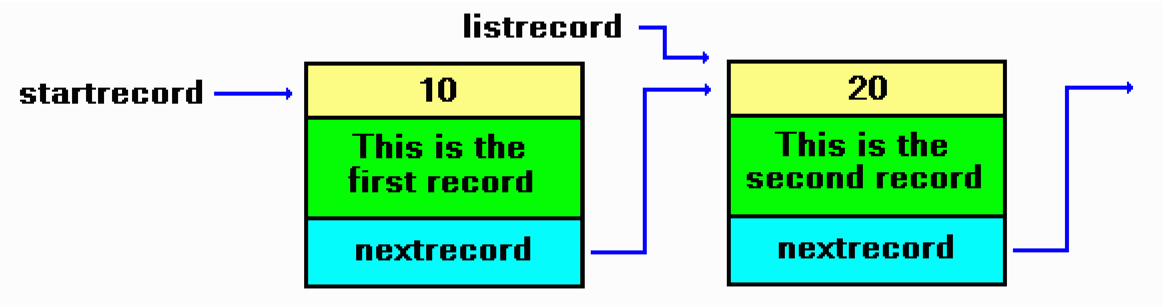
creates the first record and initialises it, then remembers where it is by saving it into *startrecord*. Pictorially, it looks like,



The lines of code

```
        new( listrecord^.nextrecord );
        if listrecord^.nextrecord = nil then
        begin
                writeln('2: unable to allocate storage space');
                exit
        end;
        listrecord := listrecord^.nextrecord;
        listrecord^.number := 20;
        listrecord^.code := 'This is the second record';
```

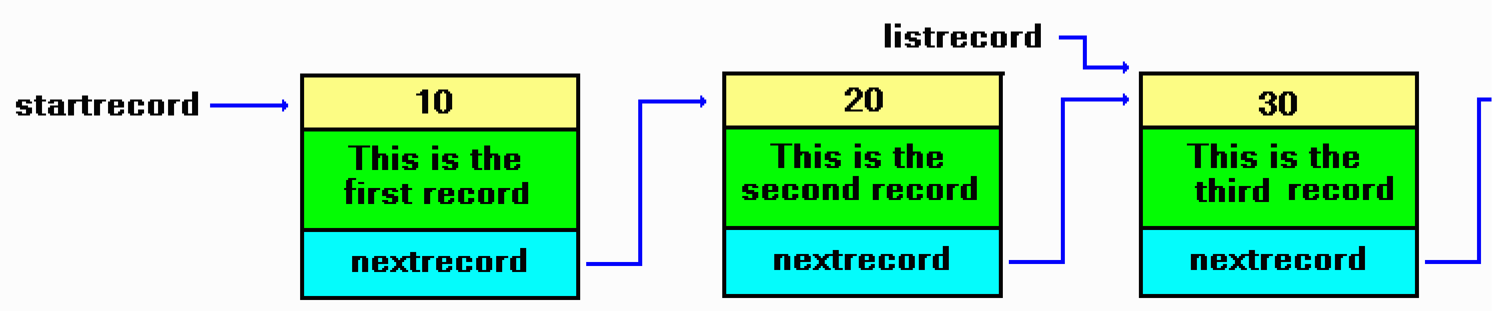
add a new record to the first by linking it into *listrecord^.nextrecord*, then moving *listrecord* to the new record. Pictorially, it looks like,



The lines of code

```
        new( listrecord^.nextrecord );
        if listrecord^.nextrecord = nil then
        begin
                writeln('3: unable to allocate storage space');
                exit
        end;
        listrecord := listrecord^.nextrecord;
        listrecord^.number := 30;
        listrecord^.code := 'This is the third record';
        listrecord^.nextrecord := nil;
```

add the last record to the previous by linking it into *listrecord^.nextrecord*, then moving *listrecord* to the new record. Pictorially, it looks like,



Note how much easier the code looks than the previous example.

POINTERS

Lets modify the previous program which introduced a separate pointer for tranversing the links of records. This time, rather than statically creating three records, we will allow the use to enter the details as the list is created.

The modified program appears below.

```
program PointerRecordExample3( input, output );

type   rptr = ^recdata;
        recdata = record
                number : integer;
                code : string;
                nextrecord : rptr
        end;

var     startrecord, listrecord, insertptr : rptr;
        digitcode : integer;
        textstring : string;
        exitflag, first : boolean;
```

```
begin
        exitflag := false;
        first := true;
        while exitflag = false do
        begin
                writeln('Enter in a digit [-1 to end]');
                readln( digitcode );
                if digitcode = -1 then
                        exitflag := true
                else
                begin
                        writeln('Enter in a small text string');
                        readln( textstring );
                        new( insertptr );
                        if insertptr = nil then
                        begin
                                writeln('1: unable to allocate storage space');
                                exit
                        end;
                        if first = true then begin
                                startrecord := insertptr;
                                listrecord := insertptr;
                                first := false
                        end
                        else begin
                                listrecord^.nextrecord := insertptr;
                                listrecord := insertptr
                        end;
                        insertptr^.number := digitcode;
                        insertptr^.code := textstring;
                        insertptr^.nextrecord := nil
                end
        end;
        while startrecord <> nil do
        begin
             listrecord := startrecord;
             writeln( startrecord^.number );
             writeln( startrecord^.code );
             startrecord := startrecord^.nextrecord;
             dispose( listrecord )
        end
end.
```

The program uses three pointers. ***startrecord*** remembers the start or head of the list, ***listrecord*** is used to link between the previous record and the next/current one, and ***insertptr*** is used to create a new record which is then linked into the chain.

POINTERS

An example of constructing a list of words and line numbers

The following program illustrates a buggy method of reading a small file and generating a list of words and associated line numbers. It does this using a linked list.

Its been ported from a C equivalent example in the C programming module. It fails on large text files (generates a heap overflow error). Proper handling of error situations is minimised so as to concentrate primarily on code execution.

Use it at your own peril.

```
program findwords( input, output );

{ $M 32000, 65536 }

const TRUE = 1;
      FALSE = 0;
      BS = 8;
      TAB = 9;
      LF = 10;
      VT = 11;
      FF = 12;
      CR = 13;


{  this holds the line numbers for each word. Its double linked for
   ease of freeing memory later on }
type listptr = ^list;
     list = record
              line : integer;       { line number of occurrence     }
              nextline : listptr;   { link to next line number      }
              prevline : listptr    { link to previous line number  }
     end;

{ this holds the word with a link to a struct list holding line
   numbers. Double linking to simplify freeing of memory later on   }
    wordptr = ^words;
    words = record
              word : string;        { pointer to word                 }
              lines : listptr;      { pointer to list of line numbers }
              nextword : wordptr;   { pointer to next word in list    }
              prevword : wordptr;   { pointer to previous word in list}
     end;

var
   head, tail : wordptr;   { beginning and end of list       }
   fin : file of char;     { input file handle               }
   filename : string;      { name of input file              }
   thisisfirstword : integer;  { to handle start of list words=0 }

{ customised exit routine to provide orderly shutdown }
procedure myexit( exitcode : integer );
var
   word_ptr, tempw : wordptr;
   line_ptr, templ : listptr;
begin
   { close input file }
   close( fin );

   { free any allocated memory }
   writeln('Deallocating memory:');
   word_ptr := head;
   while  word_ptr <> nil do
   begin
      tempw := word_ptr;              { remember where we are        }
      line_ptr := word_ptr^.lines;    { go through line storage list }
      while line_ptr <> nil do
      begin
         templ := line_ptr;               { remember where we are    }
         line_ptr := line_ptr^.nextline;  { point to next list       }
         dispose( templ )                 { free current list        }
      end;
      word_ptr := word_ptr^.nextword;     { point to next word node  }
      dispose( tempw )                    { free current word node   }
   end;

   { return to OS }
   halt( exitcode )
end;

{ check to see if word already in list, 1=found, 0=not present }
function checkforword( word : string ) : integer;
var ptr : wordptr;
begin
   ptr := head;                     { start at first word in list }
   while ptr <> nil do
   begin
      if  ptr^.word = word then   { found the word?                  }
         checkforword := TRUE;     { yes, return found                }
      ptr := ptr^.nextword        { else cycle to next word in list  }
   end;
   checkforword := FALSE           { word has not been found in list  }
end;

{ enter word and occurrence into list }
procedure makeword( word : string; line : integer );
var
   newword, word_ptr : wordptr;
   newline, line_ptr : listptr;
begin
   if checkforword( word ) = FALSE then
   begin
      { insert word into list }
      newword := new( wordptr );
      if newword = nil then
      begin
         writeln('Error allocating word node for new word: ', word );
         myexit( 1 )
      end;
      { add newnode to the list, update tail pointer }
      if thisisfirstword = TRUE then
      begin
        head := newword;
        tail := nil;
        thisisfirstword := FALSE;
        head^.prevword := nil
      end;
      newword^.nextword := nil;    { node is signified as last in list }
      newword^.prevword := tail;   { link back to previous node in list }
      tail^.nextword := newword;   { tail updated to last node in list }
      tail := newword;
      { allocate storage for the word including end of string NULL }
      tail^.word := word;

      { allocate a line storage for the new word }
      newline := new( listptr );
      if newline = nil then
      begin
         writeln('Error allocating line memory for new word: ', word);
         myexit( 3 )
      end;
      newline^.line := line;
      newline^.nextline := nil;
      newline^.prevline := nil;
      tail^.lines := newline
   end
   else
   begin
       { word is in list, add on line number }
      newline := new( listptr );
      if newline = nil then
      begin
         writeln('Error allocating line memory for existing word: ', word);
         myexit( 4 )
      end;
      { cycle through list to get to the word }
      word_ptr := head;
      while  word_ptr <> nil do
      begin
         if  word_ptr^.word = word then
             break;
         word_ptr := word_ptr^.nextword;
      end;
      if word_ptr = nil then
      begin
         writeln('ERROR - SHOULD NOT OCCUR ');
         myexit( 5 )
      end;
      { cycle through the line pointers }
      line_ptr := word_ptr^.lines;
      while  line_ptr^.nextline <> nil do
         line_ptr := line_ptr^.nextline;

      { add next line entry }
      line_ptr^.nextline := newline;
      newline^.line := line;
      newline^.nextline := nil;
      newline^.prevline := line_ptr  { create back link to previous line number }
   end
end;

{ read in file and scan for words }
procedure processfile;
var
   ch : char;
   loop, in_word, linenumber : integer;
   buffer : string;
begin
   in_word := 0;       { not currently in a word }
   linenumber := 1;    { start at line number 1  }
   loop := 0;          { index character pointer for buffer[] }
   buffer := '';

   read( fin, ch );
   while not Eof( fin ) do
   begin
       case ch of
         chr(CR) : begin
                    if in_word = 1 then begin
                      in_word := 0;
                      makeword( buffer, linenumber );
                      buffer := '';
                    end;
                    linenumber := linenumber + 1
                   end;
        ' ', chr(LF), chr(TAB), chr(VT), chr(FF), ',' , '.'  :
                   begin
                    if in_word = 1 then begin
                       in_word := 0;
                       makeword( buffer, linenumber );
                       buffer := '';
                    end
                   end;
        else
                   begin
                    if in_word = 0 then begin
                       in_word := 1;
                       buffer := buffer + ch
                     end
                     else begin
                       buffer := buffer + ch
                     end
                    end;
      end; { end of switch }
      read( fin, ch )
   end  { end of while }
end;

{ print out all words found and the line numbers }
procedure printlist;
var
   word_ptr : wordptr;
   line_ptr : listptr;
begin
   writeln('Word list follows:');
   word_ptr := head;
   while  word_ptr <> nil do
   begin
      write( word_ptr^.word, ': ' );
      line_ptr := word_ptr^.lines;
      while line_ptr <> nil  do
      begin
         write( line_ptr^.line, ' ' );
         line_ptr := line_ptr^.nextline
      end;
      writeln;
      word_ptr := word_ptr^.nextword
   end
end;

procedure initvars;
begin
   head := nil;
   tail := nil;
   thisisfirstword := TRUE
end;

begin
   writeln('Enter filename of text file: ');
   readln( filename );
   assign( fin, filename );
   reset( fin );
   {  if  fin = nil then
      begin
         writeln('Unable to open ',filename,' for reading');
         myexit( 1 )
      end;
   }
   initvars;
   processfile;
   printlist;
   myexit(0)
end.
```

COMMAND LINE ARGUMENTS

When a program is invoked, it may accept arguments from the command line such as the name of a data file to process.

In TurboC, the two functions **ParamCount** and **ParamStr** are used to retrieve these values.

ParamCount

This function returns the number of arguments of the command line which follow the name of the program. In this example below,

```
        test  file1.c  file2.pas
```

the program **test** is invoked with two parameters.

ParamStr

This function returns a string representing the value of the command-line parameter.

```
program commandline( output );

var arguments : integer;

begin
    if ParamCount = 0 then
    begin
         writeln( 'No parameters supplied' );
         halt(1)
    end
    else begin
         writeln('There are ', ParamCount, ' parameters' );
         for arguments := 1 to ParamCount do
             Writeln( 'Parameter ',arguments,' = ',ParamStr(arguments) );
    end
end.
```