STRUCTURED PROGRAMMING IN PASCAL

Second Edition

HALIJAH OMAR

Second edition 1995

Genetic Computer School, 1 Selegie Road, #03-04, Paradiz Centre, Singapore 0718

CHAPTER 1

1 INTRODUCTION TO PASCAL PROGRAMMING

1.1 WORKING IN THE TURBO PASCAL ENVIRONMENT

One of the reasons Turbo Pascal is so enjoyable to use is its integrated development environment (IDE). Once you are in the Turbo Pascal IDE, you can edit, compile, run, and debug your programs without having to go back to the DOS prompt.

1.1.1 GETTING STARTED

To start Turbo Pascal, type **TURBO** at the DOS prompt and press ENTER. Turbo Pascal enters the integrated development environment. At the top of the screen is the main menu, which contains seven choices: **File, Edit, Run, Compile, Options, Debug** and **Break/Watch**. To select one of these options, you can either highlight the option using F10 and the arrow keys and then press ENTER or type the first letter of the option (for example, press F for File). In the Turbo Pascal integrated development environment, the screen is divided into two sections - the Edit window and the Watch window. Program source code is entered and edited in the Edit window, which activates the Turbo Pascal editor. The editor is like a simple word processor that you can use to write your programs. Once you have typed your program into the Edit window, you can use the Watch window to examine your program as it executes.

1.1.2 THE FILE MENU

The File menu is the first selection on the main menu. From this menu you can load source files, change the logged disk and directory, activate the DOS shell and more. The File menu contains nine choices: **Load, New, Pick, Save, Write To, Directory, Change dir, OS Shell** and **Quit.**

1.1.3 THE EDIT SELECTION

Selecting Edit brings you into Edit window and activates the Turbo Pascal editor. The highlight on the menu will disappear and a cursor will appear in the Edit window. To return to the main menu simply press the 1`10 key.

1.1.4 THE RUN MENU

From the run menu you can execute a program just as you would from the DOS prompt, or step through a program one line at a time. The Run menu includes **Run, Program reset, Go to cursor, Trace into**, **Step over** and **User screen**.

1.1.5 THE COMPILE MENU

You can use the Compile menu units and complete programs. The choices on this menu include **Compile, Make, Build, Destination, Find error, Primary rile** and **Get info**.

1.1.6 THE FILE MENU

The features in the File menu let you load a file into the Turbo editor, change directory, or execute a DOS command without leaving the Turbo Pascal integrated development environment.

The Load option is used to read a file from the disk and place it in the Turbo editor. You can call the Load function directly by pressing the F3 key. To load a file form the Directory window, use the arrow keys to highlight the name you want, and then press ENTER. The New selection on the file menu tells Turbo Pascal to employ the editor and set the file name to NONAME.PAS.

When you save the file, Turbo Pascal will ask you for a new filename. The Save selection stores the editor contents to disk. You can call this function directly by pressing the F2 key. The Write to selection writes the current contents of the editor to a filename that you specify. When you want to exit the Turbo Pascal integrated development environment, select Quit or press ALT-X to return to DOS.

1.1.7 THE RUN MENU

The options on the Run menu are used to execute a program from within the Turbo integrated development environment. Most of the selections are used for debugging programs by executing portions and then stopping. The **Run** selection on the Run menu is used to execute a program from the Turbo integrated development environment. You can also select this option by pressing **CTRL-F9**. When you select **Run**, Turbo Pascal will execute the program currently in the editor (or the file currently specified as the primary file). The program will run normally until it encounters a run-time error or a break point that you have set. When you are debugging in Turbo Pascal you cannot see what your program is displaying on the screen. If you want to see what your program is displaying (from the user's point of view), use the **User Screen** selection or press **ALT-F5**.

1.1.8 THE COMPILE MENU

The compile menu contains the commands you need to create linkable object modules or complete executable programs. You specify a primary file, get information about a compiled program, and more. When you select the **Compile** feature from the Compile Menu, Turbo Pascal compiles the source file that is

currently loaded in the editor. If the source file in the editor uses any other units, those units must be compiled first. You can call the **Compile** feature by pressing **ALT-F9**.

1.2 UNDERSTANDING OF PROGRAM STRUCTURE

Pascal has been designed to encourage modular programming. Thus, each step or logical group of steps within the algorithm can generally be translated into Pascal module. Pascal modules are called *blocks, functions* or *procedures,* depending upon the way they are used.

In addition, the syntax of Pascal requires that all of the declarations and destinations must appear at the beginning of a program. The resulting overall organization of Pascal program as below:

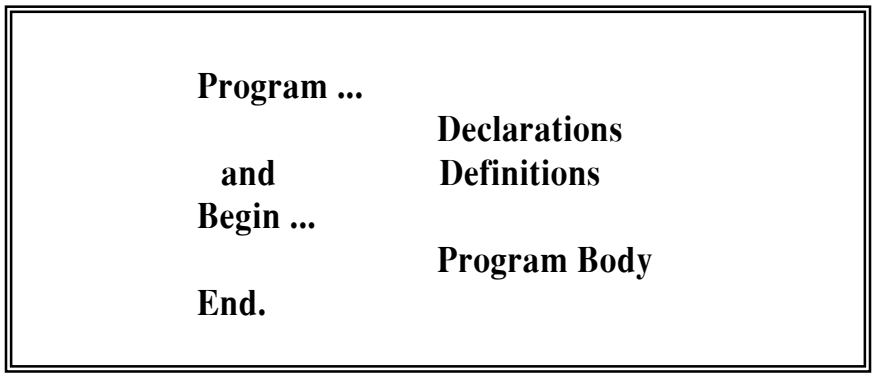


Figure 1. Overall Organization of a Pascal Program

As shown in Figure 1, all declarations appear at the beginning of a program. The declarations are followed by the main block, which is bracketed by the words **BEGIN** and **END**. Let us now consider the organization of each of these modules in more detail. A detailed of a Pascal program is shown in figure 2.

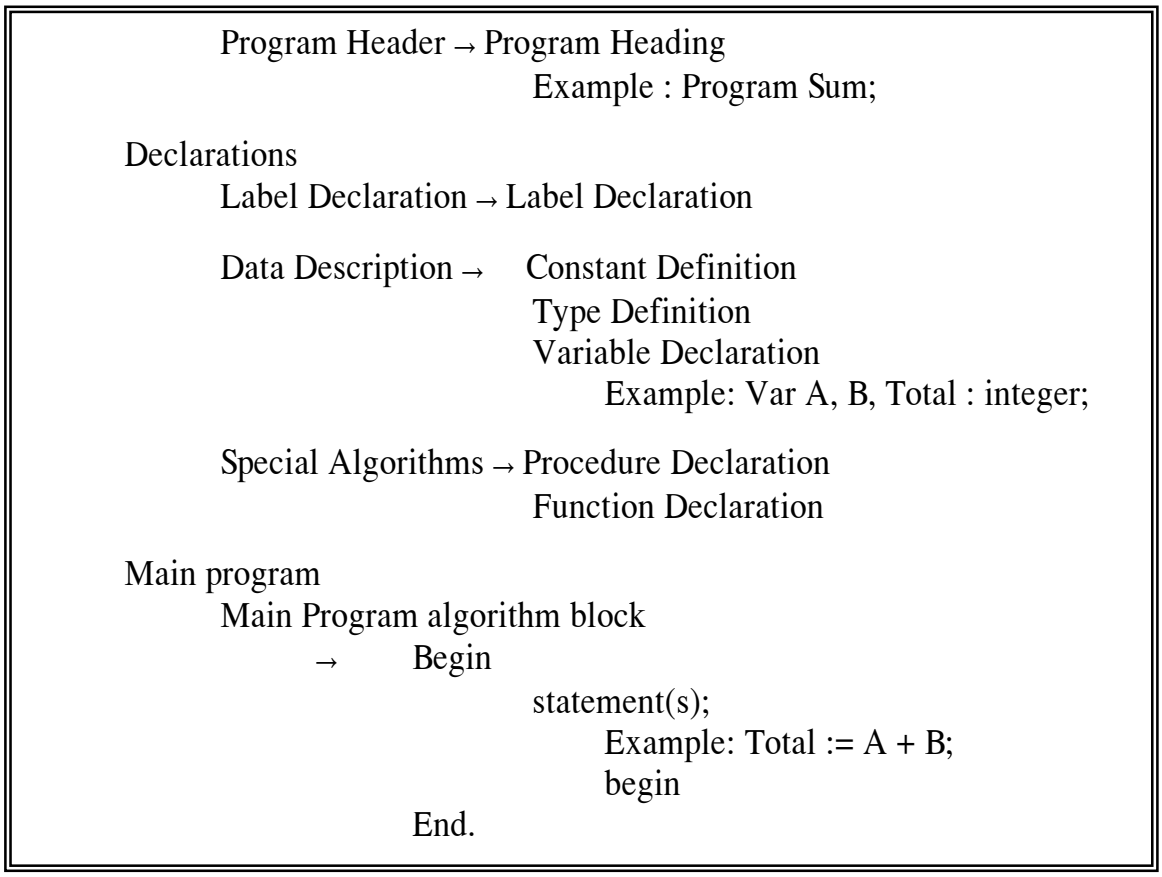


Figure 2. Detailed organization of a Pascal program

Let us look more closely at the declarations and the main body.

1.2.1 DECLARATIONS

The various kinds of declarations in Pascal must appear exactly in the order in which they are shown in figure 2; first, the labels, then the constants, etc. However, declarations are all optional.

1.2.2 THE EXECUTABLE PROGRAM BODY

The program body, shown in Figure 2, contains the sequence of statements that will execute the proper algorithms. Several types of statements can be used in Pascal. The three most important types of statements are:

1) Assignment statements
2) Input and Output statements
3) Control statements

Other types of statements include procedure calls, '**GOTO**□, and '**WITH**□ statements.

1.2.3 PROGRAM ORGANIZATION SUMMARY

In summary, each Pascal program must contain at least a program heading and a statement. In addition, a program may contains several declarations or definitions following the heading (in the proper order), as well as any number of statements. Comments, addition blanks, and indentions may be placed anywhere in a program to improve readability.

1.2.4 PARTS OF A PASCAL PROGRAM

A Pascal program or subprogram can be divided into three parts: The *header* (optional in Turbo Pascal but required in standard Pascal) names the program and can be used to document its purpose.

Definitions and declarations, which are also optional, can be categorized as

Constant definitions
Types definitions
Variable definitions
Label declarations
Subprogram definitions

Statements define the action of the program. Assignment statements, flow control statements, and so on, fall under this category. The statement part starts with the reserved word begin and ends with the reserved word end. Because the simple program has only one statement part, end also indicates the end of the program. This is indicated by a period(.) following the closing end. Every Pascal program or subprogram must have a statement part.

THE HEADER

Every program should begin with a program header. The program header begins with the reserved word program followed by a name and a semicolon(;). The program header announces the program and gives it an identity. For example, you can see from a glance that the following is a complete program:

```
program Display Greeting;
begin
writeln('Greetings ! ),
end.
```

The header indicates that this is a program, not just a procedure or function, and not only identifies the program but gives you an idea of what it does. Incidentally, procedures and functions also have headers, which are similar to program headers.

DEFINITIONS AND DECLARATIONS

To write useful programs, you almost always need some data. Data is any piece of information that will be manipulated by a program and will require storage space in the computer when you execute the program. Data can be constant (unchanging) or variable (its value may be altered as the program executes). In either case, you need a way to refer to these data items, and you must reserve space for the variables in memory. You must define constants or declare variable data items before you can use them. The purpose of *constant definitions* and *variable declarations is* to assign names, or *identifiers,* to specific data items and, in the case of variables, to reserve the required storage resources. The definition and declaration sections appear between the program header and the program statement block. Unlike standard Pascal, Turbo Pascal does not require that the definition and declaration sections appear in any particular sequence. Before you examine the predefined data types used in the declaration and definition parts, you should know a few things about identifiers. Pascal permits the use of long, meaningful names for variables, constants, headers, subprograms, and other program parts.

- Identifiers must begin with an alphabetic character.

- The first character may be followed by up to 126 additional characters, which must be alphabetic characters, numeric characters, or the underline character. The first 63 characters determine the uniqueness of the identifier.

- The case (upper or lower) of alphabetic characters is not significant. For example, TURBO, turbo, and Turbo are considered to be identical.

The rules concerning identifiers are liberal enough that you can use a meaningful name to describe each data item. Using descriptive identifiers will make your programs easier to understand, which will result in better programs with fewer errors. Although, you are only limited by your imagination when creating identifiers, Turbo Pascal reserves the identifiers listed in Table 1.1 for its own use. If you attempt to use any of those words as user-defined identifiers, the compiler will issue an error message.

Identifier	Standard Pascal	Usage
absolute	No	Variable declaration
and	Yes	Boolean and arithmetic operator
array	Yes	Type definition
begin	Yes	Logic control; used with end
case	Yes	Logic control; used with of, else, and end
const	Yes	Marks start of constant definition
div	Yes	Arithmetic operator
do	Yes	Logic control; used with for, while, and with
downto	Yes	Logic control; used with for
else	Yes	Logic control; used with if and case
end	Yes	Logic control; used with begin and case Type definition; used with record
external	No	Subprogram definition
file	Yes	Type definition
for	Yes	Logic control; used with to downto, and do
forward	Yes	Subprogram definition
function	Yes	Subprogram definition
goto	Yes	Logic control
if	Yes	Logic control; used with then and else
implementation	No	Unit definition
in	Yes	Set operator
inline	No	Marks start of inline machine code
interface	No	Unit definition
interrupt	No	Subprogram definition
label	Yes	Marks start of label declarations
mod	Yes	Arithmetic operator
nil	Yes	Pointer to nothing
not	Yes	Boolean and arithmetic operator
of	Yes	Logic control; used with case Type definition; used with array, file and set
or	Yes	Boolean and arithmetic operator
overlay	No	Subprogram definition
packed	Yes	Not used in Turbo pascal
procedure	Yes	Subprogram definition
program	Yes	Program definition; optional in
record	Yes	Type definition
repeat	Yes	Logic control; used with until

set	Yes	Type definition
shl	No	Arithmetic operator
shr	No	Arithmetic operator
string	No	Type definition
then	Yes	Logic control; used with if
to	Yes	logic control; used with for
type	Yes	Marks start of type definitions
unit	No	Unit definitions
until	Yes	Logic control; used with repeat
uses	No	Marks start of type definitions
var	Yes	Marks start of variable declarations
while	Yes	Logic control; used with do
with	Yes	Record variable field access; used with do
xor	No	Boolean and arithmetic operator

1.2.5 A SIMPLE TURBO PASCAL PROGRAM

The best way to begin is by writing your first program. To start Turbo Pascal, make sure you are logged in to the drive and directory in which the **TURBO.EXE** file resides. At the DOS prompt, type **TURBO** and press enter. On your screen, you see the Turbo Pascal integrated development environment. At the top of the screen is the main menu, which gives you access to all of Turbo Pascal's features. Below the main menu is the Edit window, in which you will type your programs. And below the edit window is the watch window, which is used in debugging programs.

To write your first program, press F10 to activate the main menu and then press E (for edit). Now the cursor will appear in the edit window, ready for you to begin typing in the following Turbo Pascal program, which will display one line of text on your computer's screen:

```
Program progl;
Begin
        writeln('This is my first program.□);
                readl n;
End.
```

If you make a typing mistake, use the arrow keys on the numeric keypad to position the cursor at the error, press DEL to delete the error, and then type the correct liters. Once you have completely typed the program, press F10 again to activate the main menu. Then press R to select the RUN menu, and press R again to run your program. Turbo Pascal will now execute the program you just wrote - your monitor will show this message:

This is my first program.

When you are ready to return to the integrated development environment, press ENTER. While it is small, this program contains elements common to all Turbo Pascal programs. It has a program heading, **Program Prog1**, which identifies the program. It also has a program block that starts with **Begin** and terminates with **End**, as shown below:

```
Begin
        Writeln ('This is my first program');
        Readln;
End.
```

1.2.7 ADDING VARIABLES TO A PROGRAM

Programs that merely write messages are not very interesting. To be truly useful, a program must process data, and that requires the use of *variables;* places in your computer's memory that hold values, such as numbers or strings.

To define a variable, you must give it a name and a type. You can give a variable almost any name you want, but it is best to choose a name that describes the information the variable holds. For example, you might call a variable that holds the name of a customer **CustomerName** and define it as follows:

```
Var

CustomerName : String[50];
```

CustomerName, the name variable, is also referred to as the variable identifier because it defines by name the location in memory where the value is stored. **String [50]** identifies the variables as a string and includes that the length of the string cannot exceed 50 characters.

Var is a Turbo Pascal *reserved word* that indicates the beginning of variables declarations. Reserved words are central to the Turbo Pascal language, and therefore, you cannot redefine them. These examples illustrate illegal attempts to use reserved words as variable identifiers:

```
Var
        Begin : Integer;
        Real :String;
```

Integer and *Real* variables can hold numbers; *char* variables can hold single characters; *String* variables can hold groups of characters; and *Boolean* variables contain true/false indicators. While they are designed for different purpose, these

variable types share one common characteristic-their values can be changed (or varied) in a program by using an *assignment statement*.

Assignment statements set variables to particular values. For example, the statements:

CustomerName := 'John Doe';

Takes the group of characters "John Doe" and stores them in the string variable **CustomerName**. Note that the assignment statement uses the := operator, which is known as the **assignment operator**.

1.2.7 VARIABLES AND INPUT

The assignment statement is just one way to set the value of a variable; the **Readln procedure** is another. But unlike assignment statements, **Readln** gets its value from a source outside the program, such as the person using the program or a disk file. When a program encounters a **Readin** statement, it stops and waits until the user types in the data and press ENTER. **Readln** then takes the input and assigns it to a variable named in the **Readln** statement. For example, the Pascal input statement **Readln(CustomerName)** waits for the user to type in a string of characters, accepts a string, and stores it in the variable **CustomerName.**

The following sample program demonstrates how **Readln** obtains input and stores it in variables.

```
Program Prog2;
Uses Crt;
Var
        i : Integer;
        s : String;
Begin
        Clrscr;
        Writeln('Enter a number: □,)
        Readln(i);
        Writeln('Your number is □,i);
        Writeln('Enter a string: □);
        Readln(s);
        Writeln('Your string is □,s);

        Readln;
End.
```

1.3 ASSIGNMENT

In Pascal language, there is a special way of placing a value in a memory location referenced by a variable. This action is accomplished by using := the assignment symbol. Consider the following Pascal instruction:

HoursWk := 42

This Pascal instruction reads *"HoursWk is assigned (.-=) the value 42*. "When this instruction is executed, the integer 42 is placed in the memory location referenced by the identifier HoursWk.

The assignment instruction can be used with any variable type. For example, if Name is declared a variable of type String[91, then the statement *Name : = 'Gary' is* a valid Pascal instruction. The Pascal reserved words **READ** and **READLN** also are used to assign a value to a variable. However, **READ** and **READLN** allow the program user to make the assignment by entering information from the keyboard during the execution of the program.

1.4 Simple Data Types

Identifier	Description,
Boolean	true and false
Integer	signed integer values -32768... 32767
Longint	signed integer values - 2147483648 ... 2147483647
Real	real numbers; accurate to 11 digits
Double	real numbers; accurate to 16 digits
Char	ASCII set of characters plus #128 through #225

1.5 Input and Output keywords

A Turbo Pascal program always starts execution at the first **Begin** statement of the main program block and continues until it reaches the final **End** statement. The program block in the example just given contains only two statements.

Writeln('this is my first program');
Readln;

The **Writeln** statement displays the string, and **Readln** makes the computer wait until you press enter. If you run a program in the integrated development environment ODE), you can use the **Readln** statement to stop a program before the screen switches back to the editor screen. When in the IDE, you can always view the output screen by pressing ALT-F5.

Writeln is a Turbo Pascal standard procedure that displays numbers and strings of characters to the screen, prints them on a printer, or writes them to a disk file. It also adds two special characters, carriage return and line feed (ASCII codes 13 and 10), to the end of the line. These special characters, often referred to in programming shorthand as CR/LF, signal that the end of a line of text has been reached, and that any additional text should start on the next line. Like **Writeln**, the procedure write also displays strings and numbers; but **Write** does not add CR/LF characters to the end of the line. When you use **Write**, the cursor remains on the same line as the information written, . while **Writeln** moves the cursor to the beginning of the next line.

1.6 Formatting

1.6.1 Formatting an Integer Value

Value	Format	Printed Output
234	:4	▌234
234	:5	▌▌234
234	:6	▌▌▌234
-234	:4	-234
-234	:5	▌-234
-234	:6	▌▌-234
234	:Len	▌▌▌234(if Len is 6)
234	:1	234

1. Edit Window

```
Program CountCoins;
Var
        Nickels, Pennies, Coins, Cents : Integer;
Begin
        Writeln('How many nickels do you have? ');
        Readln(Nickels);
        Write('How many pennies do you have? ');
        Readln(Pennies);
        Coins := Nickels + Pennies;
        Cents := 5 * (Nickels + Pennies);
        Writeln('You have ',Coins :3, 'coins.');
        Writeln('Their value is ',Cents :4,'cents.');
End.
```

2. Output Window

```
How many nickels do you have? 3
How many pennies do you have? 2
You have 5 coins.
Their value is 25 cents.
```

1.6.2 Formatting Real Values

Value	Format	Printed Output
3.14159	:5:2	▌3.14
3.14159	:4:2	3.14

3.14159	:3:2	3.14
3.14159	:5:1	▮▮ 3.1
3.14159	:5:3	3.142
3.14159	:8:5	▮ 3.14159
3.14159	:9	▮ 3.142+00
0.1234	:4:2	0.12
-0.006	:4:2	-0.01
-0.006	:9	-6.00E-03
-0.006	:8:5	-0.00600
-0.006	:8:3	▮▮ -0.006

1. Edit Window

```
Program Trip;
Var
   Speed, Time, Distance, Mileage, Gallons : Real;
Begin
        Write('Enter distance in miles:');
        Readln(Distance);
        Write('Enter time of trip in hours:');
        Readln(Time);
        Speed:= Distance /Time;
        Writeln('Average speed in MPH was: ',Speed:5:1);
        Writeln;
        Write('Enter gallons used: ');
        Readln(Gallons);
        Mileage:= Distance /Gallons;
        Writeln('Miles per gallon was: ',Mileage :5:1);
End.
```

2. Output Windows

```
Enter distance in miles: 100
Enter time trip in hours: 1.5
Average speed in MPH was: 66.7
Enter gallons used: 25
Miles per gallon was : 4.0
```

1.6.3 Formatting Strings

A **String** value is always printed right-justified in its field. Therefore, blank spaces precede a string if the field in which it is printed is bigger than the string. If *Field width* is too small to accommodate a string value, it is expanded so that the entire string can be displayed.

String	Format	Printed Output
' * '	:1	*
' * '	:2	▮ *
' * '	:3	▮▮ *
'ACES'	:1	ACES
'ACES'	:2	ACES
'ACES'	:3	ACES
'ACES'	:4	ACES
'ACES'	:5	▮ ACES

CHAPTER 2

2 KEYWORDS IN PASCAL

2.1 TYPES OF OPERATORS

The following example program demonstrates how arithmetic is used in 'turbo Pascal programs and introduces another data type called **Real**. Like **Integer** variables, **Real** variables are numbers; unlike, Integers, they can have decimal places. They can also be much larger than Integers: the maximum value for an **Integer** variable is 32,767, while for a **Real** variable it is LOU, or a 1 with 38 zeroes after it.

```
Program Prog3;

Uses CRT;
Var
        Numberl, Number2,
        AddResult, SubResult,
        MultResult, DivResult : Real;
Begin
        CIrScr;
        Write('Enter a number: ');
        Readln(Numberl);
        Write('Enter another number: ');
        Readln(Number2);

        AddResult := Numberl + Number2;

        SubResult := Numberl - Number2;

        MultResult := Numberl * Number2;

        DivResult := Numberl / Number2;

        Writeln;
        Writeln('Numberl + Number2 = ',AddResult);
        Writeln('Numberl - Number2 = ',SubResult);
        Writeln('Numberl * Number2 = ',MultResult);
        Writeln('Numberl l Number2 = ',DivResult);
        Writeln;
        Writeln('NumberI + Number2 = ',AddResult :10:3);
        Writeln('Numberl - Number2 = ',SubResult :10:3);
        Writeln('Numberl * Number2 = ',MultResult :10:3);
```

```
        Writeln('Numberl / Number2 = ',DivResult :10:3);
        Writeln;
        Write('Press ENTER...');
        Readln;
End.
```

Prog3 asks the user to enter two numbers, which are assigned to **Real** variables **Number1** and **Number2** and then used in four arithmetic operations: addition, subtraction, multiplication, and division. After performing the computations, **Prog3** writes out the results in two different formats scientific and decimal.

Scientific notation, used only for **Real** variables is a short-hand way of expressing large values. For example, the result of the following calculation: **5342168903247 x 24729234798734** expressed in scientific notation, would be 1.3210774914E+26. The first part of the number (1.3210774914) contains the significant digits; the second part (E+26) is the power of 10 to which the first part is raised. In other words, the number 1.3210774914+26 can be expressed as 1.3210774914 times 10 to the 26th power.

Scientific notation is the default format for the value of Real variables in Turbo Pascal. You can, however, also write Real values in decimal format. For example, in the statement

Writeln('Numberl + Number2 = ', AddResult:10:3);

The variable **AddResult** is followed by the format specification :10:3, which tells turbo Pascal to print the **Real** variables right justified in a file that is 10 spaces wide and allows 3 decimal places. If the resulting number were equal to 5, the number would be displayed as 5.000. If the number printed requires more than the 10 spaces allocated, the program prints the entire number, taking as many spaces as needed.

2.1.1 ARITHMETIC

In arithmetic, the numbers 0, 1, 2, 3..... are called whole numbers. When this collection of numbers is combined with its additive opposites (0,-1,-2,-3.....), the resulting collection of numbers is called the **Integers**. In Pascal, when a variable is declared to be of type **Integer**, the value the variable holds must bean integer. That is, the number must be the collection (**-MAXINT-1..... 2,1,0,1,...,MAXINT**). The identifier **MAXINT** is a predefined Pascal constant whose value is 32767.

Consider this VAR section:

```
VAR HoursWk : INTEGER
```

The variable HoursWk has been declared, and type is stated as **INTEGER.** This means that any number placed in the memory location referenced by the variable HoursWk must be an integer (that is, a number like -6 or 18 or -56 or 913).

2.1.1.1 OPERATIONS ON INTEGER DATA

The operations available for INTEGER data are addition, subtraction, multiplication, and division. The operations of addition, subtraction, multiplication are indicated by use of the symbols +, -, and • , respectively. The operation of division can be indicated by the / or slash symbol. Consider the following Var section and execution section:

```
VAR     NumA : INTEGER;
        NumB : INTEGER;
        NumC : INTEGER;

Begin
        NumA : _ -18;
        NumB : = 5;
        NumC : = NurnA / NumB;
End.
```

When the instruction **NurnC := NumA / NumB** is executed, the operation of division is performed on the data in the memory locations referenced by variables **NumA** and **NumB.** In this case, division will be performed on -18 and 5, producing a result of -3.6. An attempt is made to assign this value (-3.6) to the **INTEGER** variable **NumC**. At this point, an error (type conflict) is produced, since the value -3.6 is not an integer.

The slash division always yields a decimal number. For example, 18/6 yields 3.0 (a non-integer). To avoid this problem, another form of division is available in the Pascal language. This division is indicated by the Pascal reserved word **DIV** and is used with data of type **INTEGER.** This operation can best be explained by using an example.

Consider 23 divided by 5. You reported an answer of 4 with a remainder of 3. The number 4 was called the quotient. The quotient is always an integer. The above division is indicated by the Pascal reserved word **DIV**. Thus, 23 **DIV** 5 yields 4. The remainder is indicated by the Pascal word **MOD**. Thus, 23 **MOD** 5 yields 3. In general, **Number = quotient * divisor + number MOD divisor**

To better understand the operations /, **DIV** and **MOD**, observe the following comparisons:

Computation	Results	Computation	Results	Computation	Results
-7/2	-3.5	-7 DIV 2	-3	-7 MOD 2	-1
11/5	2.2	11 DIV 5	2	11 MOD 5	1
2814	7.0	28 DIV 4	7	28 MOD 4	0
-55/8	-6.875	-55 DIV 8	-6	-55 MOD 8	-7

2.1.1.2 TYPE REAL

In the Pascal language, any number written as a decimal is said to be of type **REAL**. A digital computer cannot store all real numbers in memory. For example, the number it and the number 1/3 cannot be stored in memory. A decimal approximation is used for real numbers that cannot be stored in memory. However, using approximations in computations with data of type **REAL** can lead to some surprises.

Observe the following **CONST** and **VAR** sections:

```
CONST Angle = 0.56;
VAR SideA : REAL;
      Perimeter : REAL;
      NurnSides : INTEGER;
```

The constant Angle has been declared and its value stated as the real number 0.56. The variables SideA and Perimeter have been declared and their type stated as REAL. Notice the form used with the decimal number in Angle = 0.56. The Pascal language is very strict about the form of the decimal name used. The name must have a digit on both sides of the decimal point. That is, the number 1/2 as a decimal must be written 0.5 (.5 will not work). Likewise, the number 7 as a decimal must be written 7.0 (7. will not work).

The operations available for variables of type REAL are addition, subtraction, multiplication, and division. These operations are indicated in a Pascal program by use of the symbols +,-,* and /. In computations involving more than one operation, the order-of-operations rules are followed (see table). Consider the following:

3.2 - 5.1 * 6.0 = 3.2 - 30.6 = -27.4

and

(3.2 - 5.1) * 6.0 = -1.9 * 6.0 = -11.4

Operation	Precedence
( )	Highest

*, / , DIV, MOD
+, - Lowest

The operations of multiplication and division are done first; the operations of addition and subtraction are done second. If parentheses are present, the computation in the parentheses is done first.

Assignment of a value to a variable of type REAL is accomplished by use of READ, READLN, or the assignment instruction : = . Consider the following VAR section and execution section:

```
VAR     SideA : REAL;
        Perimeter : REAL;
        NumSides : INTEGER;

Begin
        Write('Enter number of sides: ');
        readln(NumSides);
        Write('Enter length of side: □);
        Readln(SideA);
        Perimeter := NumSides * SideA;
        Writeln('Perimeter is : □,Perimeter);
End.
```

Here, assignment to the variables NumSides and SideA is accomplished with the READLN instruction. Assignment to the variable Perimeter is accomplished with the assignment operation, (:=)

Sample Execution

Enter number of sides : 5
Enter length of side : 10.2
Perimeter is : 5.10000000000E+01
The number 5.10000000000E+01 is E notation for 51.

2.1.2 BOOLEAN EXPRESSIONS - RELATIONAL

The values represented by type **BOOLEAN** are **True** and **False**. That's right: there are only two values. These values are used to give the truth value of a statement. For example, the statement 13 = 5 + 3 has the truth value False. As simple a type as **BOOLEAN** is, it is essential to computer programming. In the next section of this chapter, you will see an introduction similar to this:

If BOOLEAN expression is true
THEN < instructionA >
ELSE < instructionB >

And, in the next lecture, you will see an instruction similar to this:

WHILE BOOLEAN expression is true DO
< instruction >

To understand and use these instructions, it is essential that you understand **BOOLEAN** expressions. (**BOOLEAN** is named for George Boole, a nineteenth century English mathematician and the inventor of the calculus of logic). One source of **BOOLEAN** expressions is the relational operators, or relational: =, > , > =, <, < =, and < >. These operators are read as shown in Table 2.4.

Operator	Example	Reads as
=	15 = 3(5)	15 is equal to 3(5)
>	13 > -5	13 is greater than —5
> =	a > = b	a is greater than or equal to b
<	x < -3	x is less than —3
< =	a+b < = c	a+b is less than or equal to c
< >	-3 < > 3	-3 is not equal to 3

Table 2.4.

When these relational operators are applied to numbers (of type **REAL** or **INTEGER**), characters, or strings of characters, a **BOOLEAN expression** results. Consider the **BOOLEAN** expressions and their values shown in table 2.5.

BOOLEAN expression	Value
17=13+4	TRUE
t7 < > 13 + 4	FALSE
-13 > 5	FALSE
2.3 < = 3 * 814	TRUE
‘A’ < ‘B’	TRUE
‘John’ > ‘Joan’	TRUE
‘New’ < ‘New York’	TRUE

Table 2.5.

With **numbers**, the BOOLEAN value is arrived at through arithmetic. With characters, the BOOLEAN value is arrived at by comparing the character's *ASCII* value (a standardized value given to each keyboard character).

With strings of characters, the comparison is made character by character. For example, with the strings 'Joan', the BOOLEAN expression 'John' > 'Joan' has the BOOLEAN Value TRUE. The first character (J) of the strings is the same; the second character (o) is also the same. In the case of the third character, h has a greater ASCII value than character a. Thus, 'John' > 'Joan' has the BOOLEAN value TRUE. The same argument applies to 'John' > 'Joanna', which has the BOOLEAN value TRUE. In the case of the expression 'New' < 'New York', the first three characters are equal, but the length of the string 'New' is less than the length of the string 'New York'. Thus 'New' < 'New York' has BOOLEAN value TRUE.

2.1.2.1 TYPE BOOLEAN IN PROGRAMS

Consider the following program section:

```
VAR Ans :BOOLEAN;
    count: INTEGER;
```

Here the identifier Ans has been declared a variable, and its type is listed as BOOLEAN. This means a memory location is created and the value stored in memory location Ans will be one of the values TRUE or FALSE. Of course, once a variable has been declared and its type stated as BOOLEAN, the variable can be assigned a value (by use of the assignment operator, :=). The value assigned the variable must be of type BOOLEAN. Consider the following:

```
READ(count);
    Ans := (count DIV 3 = 0);
```

Suppose a value of 13 is assigned the variable Count by the READ instruction. Then the value assigned to the variable Ans would be FALSE. That is, the value FALSE would be stored in the memory location referenced by the **BOOLEAN variable Ans**. Variables of type **BOOLEAN** must be assigned a **BOOLEAN** value (just as variables of type **CHAR** must be assigned a character). The **READ** and **READLN** procedures *cannot* be used to obtain a value for a variable of type **BOOLEAN**. That is, Ans is of type **BOOLEAN**, then READ(Ans) will yield an error.

A BOOLEAN value can be written to the screen. The instruction WRITELN(Ans) will write to the screen the value TRUE if the BOOLEAN variable Ans contains the value TRUE.

2.1.2.2 OPERATIONS FOR TYPE BOOLEAN

The operations available for data of type BOOLEAN (**BOOLEAN** operations) are the logic operations: **NOT**, **AND**, and **OR**. This is much like the operations available for data of type **INTEGER** are: **+, -, □, DIV,** and **MOD**. The operation **NOT** operates on one piece of BOOLEAN data; the operations of AND and OR operate on two pieces of BOOLEAN data. Consider the following BOOLEAN expression: **NOT (17 > 3)**. The expression 17 > 3 has BOOLEAN value TRUE. Thus, the expression **NOT (17 > 3)** has BOOLEAN value FALSE. When NOT operates on a piece of BOOLEAN data, it simply yields the other piece of BOOLEAN data (remember, there are two pieces of BOOLEAN data: TRUE and FALSE).

The BOOLEAN operation **AND** operates on two pieces of BOOLEAN data. Consider the following: **(17 > 3) AND (8 < 9)**. When **AND** operates on two pieces of data, it yields the BOOLEAN value TRUE if both pieces of data are TRUE. Otherwise, it yields the BOOLEAN value FALSE. In the previous example, the expression (17 > 3) has Boolean value TRUE. The expression (8 < 9) has Boolean value TRUE. Thus, the operation yields a BOOLEAN value of TRUE. That is, the BOOLEAN expression (17 > 3) AND (8 < 9) has the BOOLEAN value TRUE. Consider Table 2.7. As you can see from the table, the only time the BOOLEAN operation AND yields of TRUE is when both pieces of data have BOOLEAN value TRUE.

Expression1	Value
'A' < 'H'	TRUE
17 > 8	TRUE
('A' < 'H') AND ('17' > '8')	TRUE
23 < 15	FALSE
(23 < 15) AND (5 > 1)	FALSE

Table 2.7.

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Table 2.8.

Table 2.8. is the truth value table for the BOOLEAN operator AND. The BOOLEAN operation OR also operates on two pieces of BOOLEAN data. Consider the following: (17 > 3) OR (5 > 2 + 3). If both BOOLEAN expressions have BOOLEAN value FALSE, then the BOOLEAN operation OR yields the BOOLEAN value FALSE.

Otherwise, the operation OR yields the value TRUE. In the BOOLEAN expression just given, the expression (17 > 3) has value TRUE. The expression (5 > 2 + 3) has BOOLEAN value FALSE. Thus, the BOOLEAN expression (17 > 3) OR (5 > 2 + 3) has BOOLEAN value TRUE. Consider Table 2.9.

Expression1	Value
'A' < 'H'	TRUE
17 > 8	TRUE
23 < 15	FLASE
('A' < 'H') OR ('17' > '8 ')	TRUE
(8 = 3) OR (7 < > 7)	TRUE

Table 2.9.

As you can see, the only time the BOOLEAN operation OR yields a value of FALSE is when both of the BOOLEAN values are FALSE. Table 2.10 is the truth value table for the operator OR.

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Table 2.10.

The **BOOLEAN** operations **NOT, AND, and OR** have a set of precedence rules, just as +,-,*, and have precedence rules(an order of operations). The order (precedence) is

NOT	**Always done first**
***,/, DIV, MOD, AND**	**Done next**
+,-, OR	**Done next**
< , > , =, < =, > =, < >	**Done next**

Thus, 17 < 5 + 20 yields TRUE. However, 7 > 10 OR 15 < 20 yields an error. Since the operation OR is performed first, an attempt is made to OR the non-BOOLEAN values 10 and 15. To avoid this problem, parentheses are used. Remember, any operations in parentheses are done first.

(15 < 20) OR (7 > 10)	Is a valid BOOLEAN expression and has the value TRUE.
NOT (15 < 20) AND (7 > 10)	Is a valid BOOLEAN expression and has the value FALSE.
NOT ((15 < 20) AND (7 > 10)	Is a valid BOOLEAN expression and has the value TRUE.

2.1.3 PRECEDENCE RULES

For the next topic on integers, order-of-operation rules (precedence rules) will be discussed. Consider this arithmetic expression: **160 DIV 20 - 5 * 2 + -5.** A person might do this computation using one of the two methods shown:

160 DIV 20 - 5 * 2 + -5	**160 DIV 20 - 5 * 2 + - 5**
8 - 5 * 2 + -5	**160 DIV 20 - 5 * -3**
3 * 2 + -5	**160 DIV 20 - - 15**
6 + - 5	**160 DIV 35**
1	**4**

Obviously, one of these computations is incorrect (actually, they both are). Applying the order of operation rules to the computation just given, the following result is obtained:

160 DIV 20 - 5 * 2 + - 5
= 8 - 10 + - 5
= - 2 + - 5
= - 7

Thus, 160 DIV 20 - 5 * 2 + -5 is -7.

CHAPTER 3

3. STATEMENTS INVOLVING CONSTRUCTS (I)

3.1 SELECTION

Fundamental to any computer language is the ability to alter the flow of execution of instructions in a program. Up to this point, the Pascal programs discussed have executed each instruction listed in the execution section from first to last (Sequentially). You are now ready for an instruction that allows for skipping another instruction. That is, a decision can be made to specify which of two instructions to execute. Such an instruction is called a decision control instruction.

Sequential execution
INSTRUCTION A
INSTRUCTION B
.....
.....

3.1.1 USING IF-THEN-ELSE

A Pascal instruction that allows for decision making is the **IF THEN-ELSE** instruction. Consider the following **IF THEN-ELSE** instruction:

```
If NumA > = 0 Then
        write (NumA , ' is good.')
Else write (NurnA , ' is no good.')
```

Suppose the variable **NumA** contains the value 8. The Boolean expression **NumA > = 0** has Boolean value true. Thus, the output is 8 is good. Suppose the variable **NurnA** contains the value -13. The Boolean expression **NumA > = 0** has Boolean value False. Thus, the output is -13 is no good. In the program in Figure 1, the **IF-THEN-ELSE** instruction is used to make a decision about the size of two numbers. The program has for input two numbers. The programs is to output the two numbers in order from smallest to largest.

```
Program Order;
Var
        NurnA, NumB : Real;
Begin
        Write ('Enter first number');
        Readln(NumA);
```

```
        Write ('Enter second number');
        Readln(NumB);
        If NumA < = NumB Then
                writeln(NumA : 8 : 2 , NumB : 8 : 2)
        Else writeln(NumB : 8 : 2 , NumA : 8 : 2);
        Writeln ('Thank You!');
End.
```

Figure 1.

Sample Execution

```
Enter first number: 23.7
Enter second number: 19.8

Numbers in order are:        19.80          23.70
Thank You!
```

Style And Punctuation

The style for writing the **IF-THEN-ELSE** instruction follows:

```
IF Boolean expression Then
        < InstructionA >
Else < InstructionB >
```

COMMENTS:
The notation < InstructionA > represents a single valid Pascal instruction.

```
IF numA MOD 3 = 0 THEN
                write (NumA, ' is divisible by 3.')
ELSE writeln ('3 is not a factor of ', NumA)
```

Figure 2

As you can see in Figure 2, there are no adjacent boxes. The boolean expression box is separated from the THEN instruction box by the word THEN. The THEN instruction box is separated from the ELSE instruction box by the word ELSE. Thus, the **IF-THEN-ELSE** instruction requires no semicolons for separation. (Of course, if another Pascal instruction follows the **IF-THEN-ELSE** instruction, a semicolon will be needed to separate the two instructions.) The **ELSE** part of the **IF-THEN-ELSE** instruction can sometimes be left off, if desired by the programmer.

The program's instruction are executed in order until the **IF-THEN** instruction is reached. At that point, the Boolean expression is evaluated. If it has the value TRUE, InstructionA is executed. If it has the value **FALSE**, instructionA is skipped. After **IF-THEN** instruction, the flow of execution returns to sequential, and InstructionC is executed. Consider the following instructions to output the change from a change machine for a quarter and/or a dime:

```
Write (□Enter amount (quarter, dime): □);
Readln(Amount);
If Amount = 'quarter'
      Then writeln('Change : 2 dimes, 1 nickel.');
If Amount = 'dime'
Then writeln('Change : 2 nickels');
```

Figure 3

Sample execution:

Enter amount (quarter, dime): *quarter*
Change: 2 dimes , 1 nickel.

Sample *execution:*

Enter amount (quarter, dime): *dime*
Change: 2 nickels.

When the user entered the string quarter, the Boolean expression Amount = 'quarter' was evaluated as TRUE. Thus, the message "Change : 2 dimes, 1 nickel." was written to the screen.

When the user entered the string dime, the Boolean expression Amount = 'dime' was evaluated as TRUE. Thus, the instruction Writeln ('Change : 2 nickels.') was executed. To increase your understanding of a series of **IF-THEN-ELSE** instruction, study the following example.

Example:

Write a Pascal program that will act as a change machine. The program should take as input a value (dollar, quarter, dime, or nickel). The program will produce an output of the change based on the following:

dollar: 3 quarters, 2 dimes, 1 nickel
quarter: 2 dimes, 1 nickel

dime: 2 nickels
nickel: no change

Solution:

Look over and think about the problem. Next, sketch out a design that will solve the problem:

Design: *Get amount to be changed*
if dollar then write 3 quarters, 2 dimes, 1 nickel
if quarter then write 2 dimes, I nickel
if dime then write 2 nickels
if nickel then write no change
stop

Your first sketch of a design portrays your thoughts and words as to how to solve the problem. Carefully walk through the design to make sure it will work. If the design does not work, discard it and start over. It is better to start over at the design stage than after time and effort have' been invested in getting the program to the computer.

After the design has been tested (by hand simulation), convert it to Pascal language:

Get amount to be changed

```
Write('Enter amount for change (dollar, quarter, dime, nickel):')
    Readln(Amount)
```

If dollar then write 3 quarters, 2 dimes, 1 nickel

```
If Amount = 'dollar' Then
    writeln('Change is 3 quarters, 2 dimes, 1 nickel')
```

If quarter then write 2 dimes, I nickel

```
If Amount = 'quarter' Then
    writeln('Change is 2 dimes, 1 nickel')
```

If dime then write 2 nickels

```
If Amount = 'dime'  Then
    writeln('Change is 2 nickels')
```

If nickel then write no change

If Amount = ‘nickel’ Then
writeln(‘Sorry, I don’t do nickels’)

From the design, it can be seen that a variable of type **STRING** will be needed. The rest of the design can be handled in the execution section. The Pascal program is shown in Figure 4.

```
PROGRAM Changeit;
VAR
    Amount : String [80];
BEGIN
    write (□Enter amount for change □);
    readln (amount);
    if amount = □dollar□ then
        writeln (□Change is 3 quarters, 2 dimes, 1 nickel□);
    if amount = □quarter□ then
        writeln (□Change is 2 dimes, 1 nickel□);
    if amount = □dime□ then
        writeln (□Change is 2 nickels□);
    if amount = □nickel□ then
        writeln ('Sorry, I don"t do nickels');
END.
```

Figure 4.

Sample execution:

Enter amount for change (dollar, quarter, dime, nickel): *quarter*
Change is 2 dimes, 1 nickel

3.1.2 DECISIONS WITH REAL DATA

Consider these instructions:

```
A := 1;
B := 3;
If (A/B. + A/B + A/B = 1) Then
    writeln (□1/3 + 1/3 + 1/3 = 1□)
Else writeln (□Error due to approximation□);
```

The execution of these instructions yields Error due to approximation. The problem is with the Boolean expression and the way the computer must use approximations for some real numbers. In the **If-Then-Else** instruction, the

Boolean expression is 1/3 + 1/3 + 1/3 = 1. This expression should yield the value TRUE. However, since the computer converts 1/3 to a decimal, the expression is **0.333.... 3 + 0.333... 3 + 0.333...3 = 1** This expression yields the value FALSE, since the computer compares 0.999...9 and 1.

When comparing real values for equality, we could ask if the two values are very close to one another. In this situation, we write the Boolean expression as **ABS ((A/B + A/B + A/B) - 1) < 0.001** That is, the difference between A/B + A/B + A/B and 1 is small, making the two values nearly equal. Consider the following problem.

PROBLEM

Write a Pascal program that allows the user to enter a numerator and denominator for a fraction. The user will enter a closeness factor. Finally, the user will enter an approximation for the fraction. The output of the program will be whether or not the approximation was close enough.

Sample Execution:

Demonstration of REAL data in decisions.
Enter numerator of fraction: 2
Enter denominator of fraction: 3
Enter a small positive number for closeness: 0.001
Enter approximation of fraction: 0.67

Sorry, not close enough.

Solution

Design: Get input data
IF (a/b - Approx) < CLOSE THEN
WRITELN ('Close enough.')
ELSE WRITELN ('Sorry, not close enough.')

Walking through the design with the data from the sample execution yields (2/3 - 0.67) < 0.001 for the Boolean expression. This expression has the value TRUE, since 2/3 - 0.67 is negative. Thus, the output will be

Close enough. < < < < < < design error! > > > > > >

In fact, a value entered for Approx like *7* or 143.6 would yield an output of Close enough. The design error is with the Boolean expression. This expression needs to be **ABS (a/b - Approx) < Close**

After testing this alteration to the design, a Pascal program is written as shown in Figure 5. In working with values or variables of type REAL in a Boolean expression, be careful with the use of the relational = . The reason for this is the approximations used by the computers for most real numbers.

```
PROGRAM RealDemo;
Var
      a, b          : Integer;
      Close, Approx : Real;
Begin
      Writeln(□Demonstration of REAL data in decisions.□);
      Write(□Enter numerator of fraction: □);
      Readln (a);
      Writeln(□Enter denominator of fraction: □);
      Readln(b);
      Writeln(□Enter a small positive number for closeness: □);
      Readln(Close);
      Write (□Enter approximation of fraction: □);
      Readln(Approx);
      If ABS (a/b - Approx) < Close Then
            Writeln (□Close enough.□)
      Else wri.teln(□Sorry, not close enough.□);
End.
```

Figure 5.

3.1.3 USING CASE

In this section, you will study the decision control instruction **CASE-OF-END**. This decision control instruction can be used as an alternative to some nested IF-THEN-ELSE instructions.

3.1.3.1 MULTIPLE DECISION MAKING

The IF-THEN-ELSE instruction provides for selecting one or two instructions to execute. To make a selection from one of several instruction to execute, the **CASE-OF-END** decision control instruction can be used. An example of a **CASE-OF-END** instruction is

```
CASE Code of
      1 : writeln(□Freshman□);
```

```
                2 : writeln(□Sophomore□);
                3 : writeln(□Junior□);
                4 : writeln(□Senior□)
                Else writeln(□Error in code.□)
End.
```

The above **CASE-OF-END** instruction evaluates the Boolean expression code = 1. If code = I has value TRUE, the string *Freshman is* output, and the execution of the **CASE** instruction is completed. If code = 1 has value **FALSE**, the Boolean expression code = 2 is evaluated. If code = 2 has value TRUE, the string *Sophomore is* output, and the execution of the **CASE** instruction is finished. This process continues until the reserved word **ELSE** is reached. If the **ELSE** is reached, the **ELSE** instruction is executed, and the execution of the **CASE** instruction is finished. The **CASE-OF-END** instruction just presented is an alternative to the following nested **IF-THEN-ELSE** instruction:

```
If Code = 1 Then
        writeln('Freshman')
                Else If Code = 2 Then
                        writeln('Sophomore')
                        Else If Code = 3 Then
                                writeln('Junior')
                                Else If Code = 4 Then
                                        writeln('Senior')
                                        Else writeln('Error in Code.');
```

The general form is as follows:

```
CASE expression of
        ValueA : < InstructionA >
        ValueB : < InstructionB >
        .
        .
        ValueN : < InstructionN >
        ELSE < InstructionX >
END.
```

The *expression,* above, is a variable or a valid combination of variables, operations and values. The expression must yield a value type INTEGER, CHAR, or BOOLEAN. Type REAL and type STRING cannot be used. The values listed must be of the same type as the expression. Only one instruction will be selected by the **CASE-OF-END** instruction. Also, the **ELSE** can be omitted. If so, the instruction after the **CASE** instruction is executed. The expression is known as the case sector.

The style for the instruction requires the selection to be indented under the **CASE** line. The **END** has the same indentation as the **CASE.**

3.1.3.2 ADDITIONAL FEATURES

In addition to the form just given, the **CASE-OF-END** also allows more than one value for a selection. The following **CASE-OF-END** instruction is a valid Pascal instruction:

```
CASE Grade OF
        □A□: Writeln(□Very GOOD!□);
        □B□ : Writeln(□Good□);
        □C□ : Writeln(□You made it□);
        □D□ : Writeln(□You should consider retaking the course.□);
        □F□, □W□, □I□ : Writeln(□Better luck next time!□);
End.
```

The last selection choice means that if the Boolean expression **(Grade = □F□) OR (Grade = □W□) OR (Grade = □I□)** is TRUE, then the instruction WRITELN('Better luck next time!') is executed. (As usual, a comma is used to separate the items in the list : 'F', 'W', 'I' A sequence of values can be listed for a selection in which case the symbol ... is used. For example,

```
CASE Score OF
    90..100: LetGrade := □A□;
    80..89: LetGrade := □B□;
    70..79: LetGrade := □C□;
    60..69: LetGrade := □D□
    ELSE LetGrade := □F□
END;
```

In this **CASE-OF-END** instruction, if Score has a value of 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, or 100 then LetGrade is assigned the character A. In addition to allowing multiple values for the case selector, the CASE-OF-END also allows a compound instruction for a selection instruction. Consider the following example

```
CASE Class OF
    1 :   BEGIN
              Over : = NumGrade - 70;
              NumGrade := 70 + Over DIV 3;
              WRITELN('Your makeup grade is : ',NumGrade);
          END;
    2 :   WRITELN ('Your makeup grade is : ',  NumGrade);
```

```
        3:      BEGIN
                    WRITELN('Your test score was ', NumGrade);
                    WRITELN('You should take another makeup test.');
                END;
END;
```

CHAPTER 4

4 STATEMENTS INVOLVING CONSTRUCTS (II)

4.1 REPETITION

In some programming problem, if some form of physical count is to be maintained, the execution requires the main part1body of the program to be repeated. This is sometimes called a loop where execution will branch back to a loop depending on a certain condition. Repetitive statements causes one or more statements to repeat. There are three repetitive statement types: WHILE, REPEAT and FOR.

4.1.1 USING FOR LOOP

A For loop produces more concise programming than WHILE or REPEAT loops. It has the starting value and the ending value. For loops can count forwards and backwards.

EXAMPLE 1: Write a program to count to ten.

```
PROGRAM FORCOUNT;
VAR
      COUNTER :INTEGER;
BEGIN
      WRITELN('FOR COUNT');
      FOR COUNTER := 1 TO 10 DO
            WRITELN(COUNTER);
END.
```

RESULT

```
FOR COUNT
1
2
3
...
10
```

PROGRAM EXPLANATION:

COUNTER starts at one and goes up to ten, and, for each successive COUNTER value, the loop should do a statement, in this case the Writeln statement. The For

statement initializes the COUNTER and increases it by one each time through the loop, and test for done when COUNTER is greater than ten.

EXAMPLE 2: Write a program to count from ten down to one.

```
PROGRAM FORCOUNTDOWN;
VAR
      COUNTER: INTEGER;
BEGIN
      WRITELN('FOR COUNT DOWN');
      FOR COUNTER := 10 DOWNTO 1 DO
            WRITELN(COUNTER);
END.
```

RESULT

FOR COUNT
10
9
8
...
1

PROGRAM EXPLANATION:

COUNTER starts at ten and goes down to one, and, for each successive COUNTER value, the loop should do a statement, in this case the Writeln statement initializes the COUNTER and decreases it by one each time through the loop, and test for done when COUNTER is not greater than ten.

4.1.2 USING WHILE LOOP

This statement enables a general condition to be checked before the operations within it is executed and the branching back to a loop depending on a general condition.

SYNTAX: **WHILE expression DO**
< statement >

While the expression is true, the statement under the while will be executed and following which while condition is to be tested and the execution will repeat until one specific point, the condition will become false and normal execution will proceed in a sequential manner.

EXAMPLE: Write a program to count to then and stops.

```
PROGRAM WHILECOUNT;
VAR
      [1]COUNTER : INTEGER;
BEGIN
      WRITELN('WHILE COUNT');
      [2]COUNTER := 1;
      WHILE COUNTER < = 10 DO
      [3]BEGIN
            WRITELN(COUNTER);
            COUNTER := COUNTER + 1;
      END;
      WRITELN('VALUE OF COUNTER =',COUNTER);
END.
```

RESULT:

```
WHILE COUNT
1
2
3
...

10
VALUE OF COUNTER = 11
```

PROGRAM EXPLANATION:

1 Variable COUNTER is called *a control variable. A* control variable controls how many times the WHILE statement executes. It can also be a character.

2 This line initializes the control variable to a starting value, setting COUNTER to one.

3 The compound statement executes while the value of COUNTER is less than or equal to ten. It displays COUNTER's value simultaneously. Then adds one to COUNTER each time through the loop. Finally, COUNTER becomes eleven, making the expression (counter < 10) false, ending the WHILE statement and, subsequently, the program.

4.1.3 USING REPEAT LOOP

A loop control structure similar in concept to the WHILE statement except that the condition will only be tested after the first execution of the statement block.

SYNTAX:

```
REPEAT
    statement1
    statement2
UNTIL condition1
statement3
```

This statement enables statement1 and statement2 to be executed once and only then the condition will be evaluated. If the condition is still true, statement1 and statement2 will be executed. Otherwise normal execution will resume with statement3.

EXAMPLE: Write a program to count to ten.

```
PROGRAM REPEATCOUNT;
VAR
    COUNTER: INTEGER;
BEGIN
    WRITELN('REPEAT COUNT');
    COUNTER : = 1;
    REPEAT
        WRITELN(COUNTER);
        COUNTER : = COUNTER + 1;
    UNTIL COUNTER > 10;
END.
```

RESULT:

```
REPEAT COUNT
1
2
3
...
10
```

PROGRAM EXPLANATION:

The program uses an integer control variable, COUNTER, initialized to one. The program writes successive values of COUNTER to the loop. The loop ends when COUNTER becomes greater than ten.

5 PROGRAM DESIGN

5.1 TOOLS

5.1.1 Comments

Comments define something the program must actually do. For example, they define data structures, or describe aspects of the program to the human reader. Long comments describe large data structures or command sequences; we write these in block format:

```
*
* This is a long comment in block format
* Blank comment lines aid legibility
*
```

In Pascal:

```
PROGRAM Trip;
{Calculate the fare for each trip}
...
```

Or

```
PROGRAM Trip;
(* Calculate the fare for each trip *)
```

Short comments may go on a single line, eg to describe individual data declarations or statements:

```
    Var
        payment : boolean; { 0 = not paid; 1 = paid }
```

Use comments freely to describe what is happening within a program; but remember to update them whenever you change the associated statements.

5.1.2 PSEUDOCODE & STRUCTURE CHART

Pseudocode is nearest to program languages and therefore easy to code from. Loop and selection structures are straight forward and clear. You can show levels and there is a certain freedom of expression. It is easy to learn and to produce quickly. You can also use it as an intermediate step when design charts are not easy to code. In many ways structure diagrams are a good solution when there are complex programs consisting of many modules. Levels and relationships are clear and you can see the problem at a glance.

Jackson Structured Programming (JSP) is a process applicable to Commerical Programming where the structure of the program often reflects the structures of the input and output data. For example, if we have to process a series of stock level records to produce a re-order list, the input data are an iteration of stock records. The program structure will consist of an iteration of 'process a stock level record'. At a lower level, a stock level may be above or below the re-order level ie. the input data represent a selection of 'below level records' or 'above level records'. This will be reflected in the program structure by having a selection statement:

IF level = Reorder level then ...

Scientific and Technical Programs tend to be 'logic driven' rather than 'data driven' so we cannot use the same technique. We can adapt it however, by using the same diagrammatical representation for the program, and analysing our program in terms of the basic constructs of:

The stages are:

SEQUENCE **SELECTION** **INTERATION**	Produce a Structure Chart Generate the Pseudo-Code Translate into Pascal

5.1.2.1 BASIC CONTROL STRUCTURE

Sequence - a number of operations performed one after the other. Each operation is performed exactly one time.

A consists of one B	**Lunch consists of Soup**
followed by one C	**followed by Main Course**
followed by one D	**followed by Coffee**

The pseudo-code is:

```
Begin (A-seq)                Begin (Lunch-seq)
B;                               Drink Soup
C;                               Eat Main Course
D;                               Drink Coffee
End; (A-seq)                 End (Lunch-seq)
```

Selection is a choice between two or more operations. Associated with each operation is a condition. Conditions are chosen so that ONE ONLY will be true at any one time, so only the operation associated with that condition will be performed. It is not a good idea to use a general 'else' condition as this may lead to

conditions being overlooked. Use a combination of 'not' conditions instead. The 'else' may be used at implementation time.

X and Y	**'if rich' and 'if poor'**
are conditions.	**are conditions.**
A consists of	**Main Course consists of**
one B or one C	**'eat salmon' or 'eat paste'**

The pseudo-code is:

Begin (A-Select)	**Begin (Main Course-Select)**
If X then	**If rich then**
B	**eat salmon**
else if Y then	**else if poor then**
C	**eat paste**
end; (A-Select)	**End (Main Course-Select)**

Iteration involves performing an operation zero or more times. The number of times the iteration is performed is specified by a condition associated with the operation:

1) **While condition Do action**
   The pseudo-code is:
   Begin (A-iter)
   While X
   B
   End; (A-iter)

2) **Repeat action Until condition**
   The pseudo-code is:
   Begin (A-iter) Repeat
   B
   Until X
   End; (A-iter)

3) **For no of times Do action**
   The pseudo-code is:
   Begin (A-iter)
   For n times
   B
   End; (A-iter)

5.1.2.2 STRUCTURES WITH MORE THAN ONE LEVEL

Sequence is indicated by moving across from left to right. For example, Christmas consists of Christmas Eve, on which Dinner is eaten, presents are exchanged, then rice pudding is served. On Christmas Day the church service is followed by Christmas gathering. Carolling Day follows Christmas Day.

The pseudo-code is:

```
Begin (Xmas-seq)
    Begin (Xmas Eve-seq)
        Eat Dinner
        Exchange Presents
        Eat Rice Pudding
    End (Xmas Eve-seq)
    Begin (Xmas Day-seq)
        Go to church
        Gathering
    End (Xmas Day-seq)
        Carolling Day
End (Xmas-seq)
```

5.1.2.3 COMBINING CONSTRUCTS

Different constructs cannot be mixed at the same level ie each level must be expressed as a sequence, or a selection, or an iteration. For example, A consists of B followed by C and then D. C expresses as an iteration, and D expresses as a selection. We cannot say 'A is B followed by an iteration of C, followed by D or E'. Thus, we say 'A is sequence of B, then C, then D. C is an iteration of E and D is a selection of F or G'.

Note that often an iteration requires the condition to be initialised, especially if it is a while loop while condition do. Condition needs to be established before entering the loop. This means that there will have to have a sequence above the iteration.

For example, a program reads in an integer N, followed by N integer values. It is to count the number of negative, zero and positive numbers and print out a count of each.

5.1.3 PRODUCING THE STRUCTURE CHART

Stepwise refinement is used. Most programs start with an initialisation section, followed by a body, followed by a termination section. Working down from here is more difficult, and may require some 'inspiration'. It will help to write down actions and conditions. .

Initialisation Actions

Zeroise negative, zero and positive counts
Read N
Write user explanation

Body Actions

Read a number
Increment negative count
Increment zero count
Increment positive count

Conditions

For N times
If negative
If zero
If positive

Termination Actions

Write header
write counts

This give the pseudo-code

```
Begin (Freq-seq)
        Begin (Init-seq)
                write user message
                zeroise negative, zero and positive counts
                read N
        End (init-seq)
        Begin (Body-iter)
                For N times
                        Begin (Process number-seq)
                                read number
                        Begin (Increase count-select)
                                if number < 0 then
                                        increase negative count
                                else if number = 0 then
                                        increase zero count
                                else if number > 0 then
                                        increase positive count
                        End (Increase count-select)
                        End (Process number-seq)
        End (Body-iter)
        Begin (Term-seq)
                write header
```

```
            write counts
        End (Term-seq)
End (Freq-seq)
```

CHAPTER 6

6 SETS

6.1 TYPES OF SETS

As another example of structured data, consider the data structure called **SET**. In mathematics, a set is a collection of objects. The objects might be numbers, colours, etc. In Pascal, a set is a collection of objects all of the same data type. The objects must be from an ordinal data type (CHAR, INTEGER, etc). The data structure **SET** provides a structuring of a mathematical set on a simple data type, such as characters.

To create a data type whose values are sets, the TYPE section can be used : *TYPE BaseValue = SET OF* 2.. 16. The data type following the Pascal reserved words SET OF is called the set's base type. The base type must be an ordinary data type. In this example, it is a subrange of INTEGER type data. Also, the number of values in the base type (its cardinality) is limited to 256 in Turbo Pascal.

Once the data type BaseValue is declared, it can be used in a variable declaration. For example, VAR *GoodNum : BaseValue* creates a variable whose values will be sets. The sets will contain as elements the integers from 2 through 16 inclusive. As with any variable, *GoodNum* has no value until it is initialized.

An example of an assignment instruction involving *GoodNum is GoodNum : = 12, 4, 8, 161*. This assignment instruction stores in the memory locations referenced by GoodNum the set whose elements are the numbers 2, 4, 8 and 16. In mathematics, the symbols (and) are used to denote a set. However, in Pascal, these symbols are used for comments, so the symbols [and] are used to bracket the elements in a set. The elements listed for the set are each separated by a comma. Also, elements can be indicated by using subrange notation. For example, GoodNum : = [5..8, 12] assigns to GoodNum the set whose elements are 5, 6, 7, 8 and 12.

The Pascal language offers the programmer a wide range of data type. The variables are almost limitless. Consider *TYPE People = (John, Sue, Bill, Jose, Amy);*

Committee = SET OF People;

VAR Possible: Committee;

The first instruction in the TYPE section, *People = (John, Sue, Bill, Jose, Amy)* creates an enumerated data type whose values are identifiers. The second

instruction in the TYPE section, *Committee = SET OF Peopleuses* the enumerated data type People to create a SET data type whose values are sets. The sets will contain as elements the identifiers John, Sue, Bill, Jose, and Amy. The identifier Possible is declared a variable, and its type is listed as Committee; that is, the variable Possible will hold values that are sets. An assignment to Possible could be *Possible : = [Sue, Bill, John]*

Analysis of SET Data

Just as with a standard data type, SET data can be analyzed by observing its values, operations, and predefined procedures and functions. It is helpful in this discussion to have an example:

TYPE People = (John, Sue, bill, Jose, Amy);

Committee = SET OF People;

Values : Subsets of the set declared. In this case, some of the values of the data type Committee are

[ ]; [John]; [Sue, Amy]; [Bill, Jose, John]; etc.

Operations :

Assignment (: =) : A value of type Committee can be assigned to a variable of type Committee.

Example : CashGroup := PartyGroup.

The variable CashGroup is assigned the value in the variable PartyGroup. Both variables must be of type Committee.

SET relational : <, < =, >, > =, =, < >
Example : [John] < [Sue, Bill, John] has BOOLEAN value TRUE.
[Sue, Amy] = [Amy, Sue] has BOOLEAN value TRUE.

One value of Committee is less than another value if it is a subset. Two values of Committee are equal if they have elements that are the same (the order of listing of the elements in a set is ignored).

SET arithmetic : union (+), intersection (*), difference(-)
Example : [John, Bill] + [Sue, Bill] yields [John, Sue, Bill]
[Jose, Amy] * [Sue, Amy] yields [Amy]
[Sue, Bill, John] - [John] yields [Sue, Bill]

The union (+) of two sets produces the set that contains all the elements in the two sets. No element is ever listed more than once. The intersection (*) of two sets produces the set that contains the elements common to both sets. If the sets have no elements in common, the intersection produces [ ] (the empty set). The difference (-) of two sets (A - B) produces the set that contains the elements in the first set (A) that are not in the second set (B).

6.1.1 RELATIONAL OPERATORS

A relational not mentioned in the previous list of SET relational is the relational IN. This relational tests for set membership. A BOOLEAN expression using this relational is 'A' IN ['a' .. 'z']. The value of this BOOLEAN expression is FALSE, since the character 'A' is not a member of the set ['a' .. 'z'].

6.1.2 LOGICAL OPERATORS

A common use of the relational IN is to replace BOOLEAN expressions like **(Letter > = □a□) AND (Letter < = □z□)** with a BOOLEAN expressions like Letter IN LowerCase Of course, LowerCase would have to be declared a SET variable and assigned the SET value ['a' .. 'z'].

6.1.3 USING SETS

Consider the procedure GetSelection in Figure 1, which is to return a valid selection (a character from A to E exclusive).

```
PROCEDURE GetSelection column, row : integer; Var select : char);
VAR
        validCh : set of Char;
BEGIN
        validCh : = ['A' .. 'E', 'a' .. 'e'];
        repeat
                gotoxy(column,row); (* screen location *)
                clreol;                 (* clear to end of line *)
                readln (select);
        until select IN validCh;
        select := UPCASE (Select);(* make sure capital letter *)
END;
```

Figure 1

The major uses of the data type SET involve the relational IN. To make sets easier to use, a SET value can be listed (created) in an execution section, provided its base

type is a predefined ordinal type or' an ordinal type declared in the TYPE section. That is, the procedure GetSelection could be written as shown in Figure 2.

```
PROCEDURE Getselection (column, row : integer;  Var select : char);
VAR
        VallidCh : set of Char;
BEGIN
        validCh : = [□A□ .. □E□, □a□ .. □e□];
                repeat
                        gotoxy(column,row); (* screen location *)
                        clreol;                 (* clear to end of line *)
                        readln (select);
                until select IN [□A□ .. □E□, □a□ .. □e□]
                        select : = UPCASE (Select); (* make sure capital
letter *)               END;
```

Figure 2

There are Situations where the programmer wishes to test for an element *not* in a set. Suppose an . instruction is to be executed while the value in the CHAR variable Ch is not in the set [‘A’..’E’J. To write the BOOLEAN expression for this WHILE-DO instruction requires caution :

*WHILE NOT (**Ch IN [□A□.. □E□**]) DO*
< Instruction >

Ch IN [‘A’ .. ‘E’] is a BOOLEAN expression. Thus, NOT (Ch IN [‘A’ .. ‘E’]) is a BOOLEAN expression. A common error involving this BOOLEAN expression is to write

Ch NOT IN [‘A’ .. ‘E’] < < < error > > >

6.1.4 CHARACTER SETS

The SET data structure is used to improve the readability of a program.. A variable of type SET cannot be used with READ, READLN, WRITE or WRITELN. That is, for LowerCase, a variable of type SET OF CHAR, instructions like WRITELN(LowerCase) and READLN(LowerCase) are invalid.

An additional weakness of the data structure is that the elements of a SET value cannot be accessed directly (unlike the elements of a STRING value). Finally, the base type of SET is quite limited.

For example, SET OF REAL, SET OF INTEGER, and SET OF STRING are all *invalid*. However, the use of sets and the relational IN can be used to improve the readability of some complex BOOLEAN expressions.

CHAPTER 7

7 TOP-DOWN PROGRAMMING (I)

7.1 FUNCTIONS

A function is used to define a new operation. Functions and procedures are both viewed as subprograms in Pascal.

7.1.1 USER-DEFINED FUNCTIONS

Consider the predefined (built-in) function SQR. This function has a number as input. The function SQR takes the number and multiplies it times itself. The result (output) of the function is stored in the function identifier SQR. In general, a function defined by the programmer

1. Has input values (value parameters).
2. Has an execution section to operate on the input values.
3. Produces an output value and assigns this value to the function identifier to be returned to the calling execution section.

When a programmer defines a function, the process follows the general path of defining a procedure. For example, suppose the programmer wanted a function that would cube a number. The definition would appear as follows

```
FUNCTION Cube (Num : INTEGER) : INTEGER;
BEGIN
        Cube : = Num * Num * Num;
END;
```

A typical call to the function Cube would be Volume := Cube(6). The function definition starts with the function header. This section of the function starts with the reserved word **FUNCTION**. The word FUNCTION indicates that the function identifier will be defined as an operation. Following the reserved word FUNCTION is the function identifier. Next in the **function header** is a parameter list in parentheses. The last item in the function header is a data type for the function identifier.

A function can have a CONST section and a VAR section. It must have an execution section. The function execution section is bracketed by the reserved words BEGIN and END. In the execution section, the instructions that define the function are

listed. One of these instructions (usually the last one) must assign a value to the function identifier of the type declared in the function header.

There are two major differences between FUNCTION blocks and PROCEDURE blocks:

1. A function identifier has a data type declared for it in the header. A procedure identifier does not. **FUNCTION Cube (Num : INTEGER) : INTEGER**

2. In the execution section of a function, the function identifier must be assigned a value. A procedure identifier *cannot* be assigned a value. The reason for this is that a function is designed to perform list of instructions, then return exactly one value. A procedure is designed to perform a list of instructions and possibly return several values (using variable parameters).

Another major difference between procedures and functions occurs in the **function call**. A function executes a set of instructions, then assigns a value to the function identifier. Thus, a function identifier in the calling execution section represents a value. The function identifier must be used like a value in the calling execution section. A procedure identifier must be used as an instruction in the calling execution section.

EXAMPLE

This function is to find the average (mean) of a collection of five numbers. The input to the function will be the five numbers. The output of the function is the average of the collection of numbers. Also, give a typical call to this function.

SOLUTION

```
FUNCTION MeanOf5 (Num1, Num2, Num3, Num4, Num5: REAL) : REAL;
VAR
      Sum: REAL;
BEGIN
      Sum := Numl + Num2 + Num3 + Num4 + Num5;
      MeanOf5 := Sum/5;
END.
```

A typical call to this function would be **WRITE (MeanOf5 (12.3, 14, 19, 23.8, 3) : 7 : 3)**

Consider the following function *NewPrice*. The input to this function consists of a character indicating whether price is to be marked up or marked down, the original price and the percent of change. The function is to return the new price.

```
FUNCTION NewPrice (Which : CHAR;OrigPrice,Percent : REAL;
VAR
        Amt :REAL;
BEGIN
        Amt := PerCent * OrigPrice;
        If (Which = 'U') OR (Which = 'u') THEN
                NewPrice := OrigPrice + Amt
        ELSE NewPrice : = OrigPrice - Amt;
END.
```

A possible call to NewPrice is : **ThePrice := NewPrice('D',9.87,0.30)**

A function is boxed much as a procedure is boxed. In the header box, there *must* be an identifier for the function and a data type for the identifier. In the parentheses in the header box, the parameters for the function and their types are listed. The rest of the function is boxed in the usual way.

7.1.2 USING FUNCTIONS

Here are some guidelines for using functions in Pascal programs:

1. If the subprogram is to return more than one value to the calling execution section, do not use a function. Use a procedure.

2. Use only value parameters in the function header. (To use a variable parameter would return more than one value and would confuse the readers of the 'program.)

3. Remember to list a data type for the function identifier in the function header.

4. Remember to assign a value to the function identifier in the function execution section. This assignment often is the last instruction in the function execution section.

5. A function call must list actual parameters that match the formal parameters in the function header both in number of parameters and in data type.

6. A function call is a call to have the instructions in the execution section of the function executed using the value(s) listed. The call also represents a value (of the type listed for the function in its header) and must appear in the calling execution section in an assignment instruction, BOOLEAN expression, etc.

For example, NumA := Cube (31) or WRITELN (Cube(31))

To further your understanding of functions and to see functions used in a Pascal program, consider the following example.

EXAMPLE

Write a Pascal program to process a checking account. The program is to allow the user to enter a name and the starting balance for the account. The program will then process a collection of withdrawals and deposits for the account. The output of the program will be the account name and the starting and ending balances of the account.

SOLUTION

Design : **GetName**
WHILE Transaction < > Q DO
Get Starting Balance
Get Transaction
Update Balance
Report Results

Refinement of : ***Update Balance***
If Withdrawal
Then Balance = Balance - Amount
If Deposit
Then Balance = Balance + Amount

COMMENT

A function will be used for this module. The input will be type of transaction and Amount. The function will return the Balance.

Refinement of : ***Report Results***
Give Account Name
Give Starting Balance
Give Ending Balance

PROGRAM CheckAccount;
VAR

```
        Name: STRING[30];
                StartBal, Amount, Balance : REAL;
                        Transaction : CHAR;
FUNCTION Update (OldBal, Amt : REAL; Trans : CHAR) : REAL;
        BEGIN                                                   {Update}
                IF Trans = 'D'
                        THEN Update := OldBal + Amt;
                IF Trans = 'W'
                        THEN Update := OldBal - Amt;
        END;                                                    {Update}
PROCEDURE Report (FirstBal, LastBal : REAL);
        BEGIN                                                   {Report}
                WRITELN('Starting Balance $', FirstBal : I :2);
                WRITELN('Ending Balance $', LastBal : 1 : 2);
        END;                                                    {Report}
BEGIN                                                   {CheckAccount}
        WRITE('Please enter account name: ');
        READLN(Name);
        WRITE('Please enter starting balance: $');
        READLN(StartBal);
        Balance := StartBal;
        Transaction := ' ';                     {initialize Trnsaction to blank}
        WHILE Transaction < > 'Q' DO                            {WHILE-DO}
                BEGIN
                        WRITE('Please enter transaction (W/D/Q):□);
                        READLN(Transaction);
                        IF (Transaction = 'W') OR (Transaction = 'D')
                          THEN
                                BEGIN                   {IF-THEN instruction}
                                  WRITE('Please enter amount: $');
                                  READLN(Amount);
                                  Balance: =Update(Balance,Transaction,Amount);
                                END;                    {IF-THEN instruction}
                END;                                            {WHILE-DO}
                WRITELN;
                WRITELN(Name);
                Report(StartBal, Balance);
END.
```

Consider the conditional instruction and the previous program.

IF (Transaction = 'W') OR (Transaction = 'D')
THEN ...

What problems might arise if the BOOLEAN expression were replaced by (Transaction < > 'Q')? Suppose the user entered an 'S' for the variable Transaction. The function **Update** would be called. The value 'S' would be passed to the actual parameter Trans. The instructions in the execution block of **Update** would be executed, but no value would be assigned to the function identifier **Update**. An error would be produced, since the function identifier must be assigned a value.

7.1.3 PREDEFINED FUNCTIONS

A function can be thought of as an operator such that we given an input value (or values), the value is operated on to yield an output value. The output value is stored in a memory location identified by the function name.

Consider the predefined Pascal functions SQR (short for *square*) in the following instruction : **A := SQR (7)**. The function SQR is given an input of the value 7. The function takes this value and multiplies it times itself. The result (49 in this case) is output by the function SQR. This output value is then assigned to the variable A.

Consider the following:

Instruction	Result
Hyp := SQR (4) + SQR(3)	Hyp is assigned the value 25.
WRITE (SQR(0.3))	The real number 0.09 is output as = = 9.0000000000E-02

If the input value to the Pascal function SQR of type INTEGER, then the output value is of type INTEGER. Likewise, an input value of type REAL yields' an output value of type REAL.

Another Pascal predefined function is ROUND. consider this Pascal instruction : **A := ROUND(123.4567)** The function ROUND is given the input value 123.4567. The function takes this value and rounds it off to the nearest integer (123 in this example). The output value is then assigned to the variable A. Consider the following instructions and their result :

Instruction	Result
StepF := ROUND(9.67)	The integer 10 is assigned to the variable StepF
WRITE (ROUND(-15.5)):5)	The integer -16 is output as = = -16

The function ROUND takes a value of type REAL or INTEGER as an input value and outputs a value of type INTEGER.

A function closely related to ROUND is TRUNC. Consider the following instruction: **B := TRUNC(123.4567)** The predefined function TRUNC is given the input value 123.4567. The function TRUNC takes this value and truncates (cuts off or lops) the decimal part of the value. The output value (123 in this case) is then assigned to the variable B.

Consider the following instructions and their result :

Instruction	Result
StepF : = TRUNC(9.67)	The integer 9 is assigned to the variable StepF
WRITE (TRUNC(-15.5) : 5)	The integer -15 is output as == -15

The function TRUNC takes as input a value of type REAL or INTEGER and outputs a value of type INTEGER. Yet another predefined (built-in) function that has input data of type INTEGER or REAL is ABS.

Consider this instruction: **Number := ABS(-13)**. When this instruction is executed, the variable Number will be assigned the absolute value of the number -13. The absolute value is computed as follows

1. If the number is greater than or equal to zero, then the absolute value of the number is the number.

2. If the number is less than zero, then the absolute value of the number is -1 times the number.

Thus, Number will be assigned -1 *-13 (since -13 is less than 0), or the value 13. One other predefined function dealing with data or type REAL is the SQRT function (SQRT is Pascal shorthand for square root).

Consider this instruction : **Hyp := SQRT(16)**. The result of this assignment instruction is that the variable Hyp will be assigned the input value 4.0. The variable must be declared and its type stated as REAL since the function SQRT always returns a value of type REAL.

Consider the following instructions and their results :

Instruction	Result
Discrim : = SQRT(323.0)	Discrim is assigned the value 17.9722.
Hyp := SQRT(SQR(4) + SQR(3))	Hyp is assigned the value 5.0
WRITE(SQRT(100))	The real number 10.0 is output as = =

1.0000000000E+01

CHAPTER 8

8 TOP-DOWN PROGRAMMING (I1)

8.1 PROCEDURES

In this lesson, we will explore a method of physically separating the major steps and their refinements from the main execution section of the program. That is, the program will be constructed as a collection of modules with the main execution sections coordinating the execution of the modules. This approach to program development is called the **modular approach.**

The program mirrors the solution design through the use of modules(subprograms). A program becomes a list of modules and a main execution section that describes the sequence in which the modules are to be executed (called). The modules can be developed, tested, and debugged independently of one another, then assembled to form the program. This well-organized approach to program development leads to programs that are easier to read, test, debug and maintain.

In Pascal, modules fall into two classes : procedures and functions. We will start discussing procedures.

8.1.1 PROCEDURES AND PROBLEM SOLVING

A procedure is like a small program. Just as a program tells the computer how to solve a problem, a procedure tells the computer how to do one of the major steps (subtasks) in the solution. The program layout becomes as follows :

1. list the major steps in the execution section

2. define how to do a complex step in a procedure block

8.1.2 DECLARING A PROCEDURE

A procedure block in a program looks very much like the program itself. The block begins with a procedure header section. This section begins with the reserved word **PROCEDURE**, which is followed by the procedure's identifier. The word PROCEDURE tells the compiler that the identifier will be defined as an instruction. Following the procedure header section, a CONST section and a VAR section can appear. Finally, the **procedure execution** section appears. The execution section *is* bracketed by the words BEGIN and END and contains the instruction that define the procedure.

The position of the procedure block is between the VAR section and the execution section of the program block (see figure 1.).

PROBLEM

Write a Pascal program that allows the user to select from *a* menu listing : triangle, rectangle and trapezoid. After the selection is made, the user enters the information necessary to compute the area of the shape. The output of the program is the area measurement of the selected shape.

```
PROGRAM proDemo;                              {program header section}

VAR variable_A : real;                        {program VAR section}

PROCEDURE Demo;                               {procedure HEADER section}
BEGIN                                         {procedure execution section}
      (* instructions that define the procedure *)
END;

BEGIN                                         {procedure execution section}
      (* instructions that define the procedure *)
END;
```

Figure 1.

SOLUTI0N

As usual, the starting place is the construction of a design.
Design :

```
REPEAT
      Clear screen
      Present menu
      Read selection
      Clear screen
      CASE selection OF
            'A' : DoTriangle
            'B' : DoRectangle
            'C' : DoTrapezoid
      END
UNTIL user wants to quit
```

The next step is to. refine the main steps in the solution that needs explanation. The steps "Clear Screen" and "Read selection" do not require further refinement.

Refinement of : **Present menu**

```
WRITELN('The geometric shapes available are:')
WRITELN('A. Triangle')
WRITELN('B. Rectangle')
WRITELN('C. Trapezoid')
WRITELN
WRITE('To select shape enter A,B,C or Q = Quit:')
```

Refinement of : **DoTriangle**

```
WRITE('Enter height of triangle :')
READLN(Height)
WRITE('Enter base of triangle :')
READLN(Base)
WRITE('The area measure is :')
Area := (Base * Height /2)
WRITELN(Area)
WRITELN
WRITE('Press RETURN key for main menu.')
READLN
```

Refinement of : **DoRectangle**

```
WRITE('Enter length of rectangle:')
REALD(Lenghth)
WRITE('Enter width of rectangle :')
READLN(Width)
WRITE('The area measure is :')
Area : = (Length * Width)
WRITELN(Area)
WRITELN
WRITE('Press RETURN key for main menu')
READLN
```

Refinement of : **DoTrapezoid**

```
WRITE('Enter length of major base of rtapexoid')
READLN(Bigbase)
WRITE('Enter length of major base of rtapexoid')
READLN(Smallbase)
WRITE('Enter height of trapezoid : ')
READLN(Height)
WRITE('The area measure is : ')
Area : = (1/2 * (Bigbase + Smallbase) * Height)
WRITELN(Are)
```

```
WRITELN
WRITE('Press RETURN key for main menu')
READLN
```

Sample execution.

Geometric shapes aailable are:

A. Triangle
B. Rectanhle
C. Trapezoid

Please select by entering A,B,C or Q = Quit:A

Result.

Enter height o triangle: ***12.3***
Enter base of triangles: ***5.2***
Area measure of triangle is: 36.98

Press RETURN key for main menu

After testing this design by walking through it with several examples, convert the design to a Pascal Program. The program will have the form shown in Figure 2.

```
PROGRAM GeometricShapes;
VAR Select : CHAR;

PROCEDURE PresentMenu,
    BEGIN
        {instructions for this module step go here};
    END;

PROCEDURE DoTriangle;
    VAR {variables this procedure needs go here};
    BEGIN
        {instructions for this module step go here};
    END;

PROCEDURE DoRectangle;
    VAR {variables this procedure needs go here};
    BEGIN
        {instructions for this module step go here};
    END;
```

```
PROCEDURE DoTrapezoid;
    VAR {variables this procedure needs go here};
    BEGIN
        {instructions for this module step go here};
    END;

BEGIN
    REPEAT
        CLRSCR;
        PresentMenu;
        READLN(Seiect);
        CLRSCR;
        CASE Select OF
            'A' : DoTriangle;
            'B' : DoRectangle;
            'C' : DoTrapezoid;
        END;
    UNTIL Select = 'Q'

END.
```

Figure 2.

The form of the program GeometrieShapes shown in Figure 2 is called a stub. That is, the program has been stubbed in. The stub can be compiled and executed (and often is, as a test).

It is now time to fill in the procedure blocks with their instruction. Again, the design and its refinements enter the process. The definition of each procedure is much like writing a small program. The refinement of the step (developed in the design of the solution) is used as a guide to write the procedure.

From the Presentmenu refinement, the procedure will be as follows:

```
PROCEDURE PresentMenu;

BEGIN
    WRITELN(□The geometric shapes available are : □);
    WRITELN(' A. Triangle');
    WRITELN(' B. Rectangle');
    WRITELN(' C. Trapezoid');
    WRITELN;
```

```
        WRITE(□To select shape enter A,Br C or Q = Quit : □);
    END;
```

COMMENT

Much as WRITELN by itself results in the output of a blank line to the screen. READLN by itself results in halting the program until the RETURN key is pressed. This allows the user to read the output of the procedure. Otherwise, the results would vanish from the screen almost as soon as they were printed (due to CLRSCR).

From the refinement of DoTriangle, the procedure will need variables Height and Base. Thus, the procedure produced from this refinement is:

```
    PROCEDURE DoTriangle;

    VAR Height, Base : REAL;

    BEGIN
        WRITE(□Enter height of triangle : □);
        READLN(Height);
        WRITE(□Enter base of triangle : □);
        READLN(Base);
        WRITE(□The area measure is : □);
        WRITELN(Base*Height/2);
        WRITELN;
        WRITE(□Press RETURN key for main menu □);
        READLN;
    END;
```

A procedure is given everything it needs to do the task for which it is designed. The procedure DoTriangle needs variables Height and Base. Thus, they are declared in the VAR section for the procedure block. This procedure is said to be self-contained (or modular). Continuing the process, the program GeometricShapes is as shown in figure 3.

```
PROGRAM GeometricShapes;

VAR Select: CHAR;

PROCEDURE PresentMenu;
    BEGIN
        WRITELN(□The Geometric Shapes available here□);
        WRITELN(□      A.    Triangle□);
```

```
            WRITELN(□      B.    Rectangle□);
                WRITELN(□      C.    Trapexoid□);
                WRITE (□To select shape enter A, B, C, Q = Quit : □);
            END;
PROCEDURE DoTriangle;
    VAR Height, Base : REAL;
    BEGIN
        WRITE(□Enter height of triangle : □);
            READLN(Height);
            WRITE(□Enter base of triangle : □);
                READLN(Base);
                WRITE(□The area measure is : □);
                WRITELN(Base*Height/2);
                WRITELN;
                WRITE(□Press RETURN key for main menu : □);
                READLN;
            END;

PROCEDURE DoRectangle;
    VAR Length, Width: REAL;
    BEGIN
        WRITE(□Enter length of rectangle : □);
        READLN(Length);
        WRITE(□Enter width of rectangle : □);
            READLN(Width);
            WRITE(□The area measure is : □);
            WRITELN(Length*Width);
            WRITELN;
            WRTTE(□Press RETURN key for main menu .. □);
                READLN;
        END;

PROCEDURE DoTrapezoid;
    VAR BigBase, SmallBase, Height : REAL;
    BEGIN
        WRITE(□Enter major base of trapezoid : □);
            READLN(BigBase);
            WRITE(□Enter minor base of trapezoid : □);
                READLN(SmallBase);
                WRITE(□Enter height of trapezoid : □);
```

```
                    READLN(Height);
                    WRITE('The area measure is : ');
                    WRITELN(0.5*(BigBase + SmallBase)*Height);
                    WRITELN;
                    WRTTE('Press RETURN key for main menu .. ');
                        READLN;
                    END;

BEGIN
    REPEAT
        CLRSCR;
        PresentMenu;
        READLN(Select);
        CLRSCR;
        CASE Select OF
            'A' : DoTriangle;
            'B' : DoRectangle;
            'C' : DoTrapezoid;
        END;
    UNTIL Select = 'Q'
END.
```

Figure 3.

COMMENT

To read a Pascal program, always start reading at the program execution section. One reason for using procedures is to allow the reader to read the program in as much or as little detail as desired. The reader can read the main steps in the program execution section, and if he or she desires to see how step is done, he or she can look up in the program to the desired step definition in the appropriate procedure block.

When the program is run, it starts executing the instructions in the main execution section. When a procedure identifier is encountered, like DoTriangle or PresentMenu, control is sent to the procedure's execution section. The procedure is said to be called (or executed). That is, the program calls (executes) the procedure PresentMenu.

8.1.3 STYLE AND PUNCTUATION

A procedure (module) block *must* contain a procedure header and an execution section. It *may* contain a CONST section and a VAR section. The word

PROCEDURE is a pascal reserved word (thus, it is capitalized). The sections of a procedure are indented under the procedure. This indentation indicates that these sections belong to the procedure. A procedure is boxed much the same way as a program is boxed.

1. The word PROCEDURE starts the procedure box and the procedure header box
2. An identifier is sought and boxed.
3. The procedure header box is closed.
4. If the word CONST is found, the CONST sections is boxed.
5. If the word VAR is found, the VAR section is boxed.
6. The word BEGIN is sought and starts the execution section box.
7. The instructions in the execution section are boxed.
8. The word END is sought, and the execution section box is closed.
9. The procedure box is closed.

From Figure 4., you can see that a semicolon is needed to separate the procedure header section from the VAR section, and a semicolon is needed to separate the VAR section from the execution section. Also, semicolons are needed to separate the instructions within the procedure execution section.

```
PROCEDURE doTriangle;
        VAR Height, Base, Area : Real;
BEGIN
        write (□Enter height of triangle: □);
        readln (height);
        write (□Enter base of triangle: □);
        readln (base);
        write (□The area measure is: □);
                area := (height * base);
                writeln (area:0:2);                    {with 2 decima
places}         writeln;
                write (□Press RETURN key for main menu-□);
                readln;
           END.
```

Figure 4

A semicolon will be needed to separate one procedure section from another. Also, a semicolon will be needed to separate the last procedure block from the program execution section. For this reason, every procedure block must be followed by a semicolon.

8.1.4 DOCUMENTATION

Like a program, a procedure should have a comment section. In this comment section, the purpose of the procedure and a brief description of its input and its output should be recorded. As an example, see the procedure in Figure 5.

```
PROCEDURE doTriangle;
{Purpose:    To compute the area of a triangle.
Input:       The base and height of a triangle will be entered by the
program user.
Output:      The area will be written to the screen,}
        VAR Height, Base, Area : Real;
BEGIN
        write (□Enter height of triangle: □);
        readln (height);
        write ('Enter base of triangle: □);
        readln (base);
        write ('The area measure is: □);
        area := (height * base);
        writeln (area:0:2);                          {with 2 decimal places}
        writeln;
        write ('Press RETURN key for main menu-□);
        readln;
END.
```

Figure 5

When the program is compiled (converted to object code), the comments are ignored.

COMMENT

Just as a program should have a comment section, so should a procedure. Often a programmer waits until the program is tested and debugged before inserting comments. Before a program is finished, it must be documented properly.

8.2 TYPE OF VARIABLES

8.2.1 GLOBAL IDENTIFIERS

Consider the example Pascal program in figure 6. When the program *Demo is* compiled, memory locations are set aside for the program variables *NumA*, *Name*

and *Results*. These variables are called **global variables** or **global identifiers**. That is, they are known (can be used) throughout the program. This means that one of instructions in the procedure Trial could use one or more of the variables identifiers *NumA*, *Name* or *Results*.

```
PROGRAM Demo;
Var
        NumA : integer;
        Name : string [30];
        Results : real;
procedure Trial;
Var
        fNum, sNum : integer;
Begin
        {procedure instructions}
End;

Begin { main program }
        {program instructions}
End.
```

Figure 6

8.2.2 LOCAL IDENTIFIERS

When the procedure Trial is called, memory locations are set aside for the procedure variables *(FNum* and *SNum*). After the instructions of the procedure have been executed, these memory locations are taken back by the system (their contents are lost). They are known (can be used) only when the procedure Trial is executing, and cannot be used in the execution section of the program Demo. For this reason, the variables *FNum* and *SNum* are said to be local variables or **local identifiers**.

Local variables are destroyed (the memory locations are taken back by the Pascal system and used for other purposes) after the instructions of the module are executed. Thus, any local variables should be initialized each time the module is called (since the old values are lost). For example, suppose you made a call to the procedure Trial and at the end of Trial's execution, *FNum* contained the value 5. Do not assume that FNum will contain the value 5 when Trial is called again !

What happens if a variable identifier is declared in a module that has the same name as a global identifier? When the module is called (executed), the system reserves a different memory location than the one used for the global identifier.

For example :

```
PROGRAM Example;

VAR NumB : INTEGER;
PROCEDURE One;
        VAR NumB : INTEGER;
        BEGIN
                NumB := 3;
                WRITELN('NumB is', NumB : 3);
        END;
BEGIN
        NumB := 12;
        One;
        WRITELN('NumB is', NumB : 3);
END.
```

CHAPTER 9

9 TOP-DOWN PROGRAMMING (III)

9.1 PARAMETERS

The use of procedures usually will include the use of parameters. Parameters are used to allow the transfer of data to and from procedures. The parameters used with procedures fall into two classes value and variable. Both classes will be discussed, as well as procedure calls involving parameters.

9.1.1 ACTUAL AND FORMAL, PARAMETERS

In defining a procedure, any valid identifier can be used in the parameter list. The identifier does not have to be declared before it is listed int he procedure header. This allows procedures to be written independently (perhaps by another programmer). The identifiers listed in the procedure header are called **formal parameters**. The word formal, in this context, means the outward from of something rather than its content.

Comment

A variable parameter and the variable in the procedure call share the same memory location. That is the *actual* memory location being used is the variable in the procedure call. The variable parameter is just a *formality*.

The identifiers used in the procedure call must be declared prior to this use. The list of parameters in the procedure call are known as actual parameters. The list of actual parameters must match the list of formal parameters (the ones in the procedure heading). That is,

1. The number of actual parameters in the call must be the same as the number of formal parameters in the procedure header

2. The types of the parameters must match

3. A variable parameter in the procedure header must be matched with a variable identifier in the procedure call.

Any reference to a (formal) parameter within the procedure execution section uses the identifier listed in the procedure header. When writing a procedure, think and write in a local sense. That is, act as if you know nothing of the program except what is required of the procedure you are writing.

To illustrate these points, consider the following procedure headers and the corresponding calls to the procedure

Example 1.

```
PROCEDURE ProcessPair (VAR Ans : BOOLEAN;
                       FNum, Snm : INTEGER) {formal}

Call : ProcessPair (First, 13, -29)          {actual}
```

The actual parameter First must be a variable whose type is BOOLEAN.

Example 2.

```
PROCEDURE checkList (StudentID, Course : INTEGER;
                     VAR Grade: CHAR;
                     VAR NumHrs : INTEGER);

Call : CheckList (StudentN, 1104, Grade, Hours)
```

StudentN must contain a value whose type is INTEGER, Grade must be a variable identifier whose type is CHAR, and Hours must be a variable identifier whose type is INTEGER.

9.1.2 VALUE, VARIABLE AND MIXTURE

9.1.2.1 VALUE PARAMETERS

A collection of parameters is used to define a particular instance of a process. For example, a linear function has the general form of y = mx + b. Suppose m has the value 6 and b has 'the value 12. Then you have a particular instance of a linear function, y=6x+ 12. the symbols m and b are called parameters for a linear function. To introduce parameters and their use in programming, consider the following problem.

Problem

Write a Pascal program that accepts as input a base and an exponent (both positive integers). The program is to output the resulting base to the exponent power.

Solution

After thinking about the problem, develop the following design:

Design : ReadBase&Exponent
DoPower

Refinement of : **ReadBase&Exponent**

```
WRITE('Please enter base: ')
READLN(Base)
WRITE('Please enter exponent : ')
READLN(Exponent)
```

Refinement of : **DoPower**

```
Count : = 1
Power : = 1
WHILE Count < = ExpNum DO
        Power : = Power * BaseNum
        Count : = Count + 1
WRITELN('The power is: ',Power)
```

To convert the step **DoPower** to a procedure, the procedure will require the transfer of values from the program, namely, a value for BaseNum and a value for ExpNum.

To perform this transfer, parameters are used. Consider the conversion of the step DoPower to a procedure presented in figure 1.

As you can see in Figure 1, the procedure header has been changed. Additional information has been added. The information enclosed in parentheses,

BaseNum, ExpNum : INTEGER

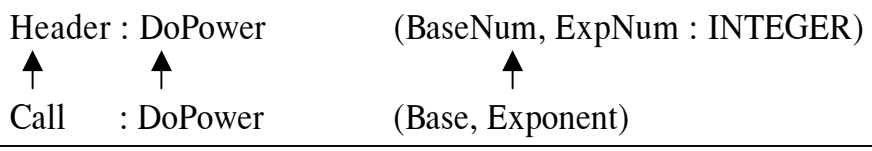
is called a **parameter list** for the procedure. The list indicates that two values will be sent to the procedure DoPower when it is called. The first value will be placed in the variable BaseNum; the second value will be placed in the variable ExpNum. The values sent must be of type INTEGER, since the variables are declared to hold integer values. The parameters BaseNum and ExpNum are called value parameters.

```
PROCEDURE DoPower (baseNum, ExpNum : integer);
VAR
        count : integer;
        power : real;
BEGIN
        count := 1; power := 1;
        While count < = ExpNum Do
        Begin
                power := power + baseNum;
                count := count + 1;
        End;
        Writeln ('The power is : □, power);
END;
```

Figure 1

With this change in the procedure header, there is a change in the call to the procedure DoPower. The following would be a valid all to the procedure DoPower :

DoPower (Base, Exponent)



A valid call must include two values of type INTEGER. In the call to the procedure, DoPower(Base,Exponent), the value in the variable Base is passed to the procedure and is then placed in the variable BaseNum. The value in the variable Exponent is passed to the procedure and placed in the variable ExpNum.

When the procedure DoPower is called, memory locations are set aside for the variables BaseNum and ExpNum. The values passed are stored in these memory locations. When the execution of DoPower is finished, the variable BaseNum and ExpNum are forgotten (along with any variables or constants listed in the procedure's VAR and CONST sections); that is, the memory space allotted to store the values of these variables is taken back by the system.

The conversion of the design to a Pascal program is shown in figure 2

COMMENT

The procedure call DoPower(Base,Exponent) in figure 2 is an instruction that tells the program to execute the instructions listed in the execution section of the procedure DoPower. The execution of these instructions should be done using the values sent in the procedure call (namely, the values in the program variables Base and Exponent).

In the previous example, the procedure DoPower will process any two numbers of type INTEGER. When the call is made to DoPower, the two values are passed to request a particular instance of the procedure DoPower. In the call

DoPower(Base,Exponent)

the values passed are the values stored in the program variables Base and Exponent

Actually, any two expressions could be passed. All the following calls are valid:

DoPower (16,7) **{16' is computed}**

DoPower (5, Exponent*3) **{$5^{Exponent\ -3}$ is computed}**

DoPower (Base*3,Exponent+4) **{Base*$3^{Exponent\ +4}$ is computed}**

The following calls are invalid :

DoPower(4) ***{needs a value for ExpNum}***

DoPower(3.2, 5) **{the first value must be of type INTEGER}**

```
PROGRAM powers;
VAR
        Base, exponent : integer;
PROCEDURE DoPower (baseNum, ExpNum : integer);
VAR
        count : integer;
        power : real;
BEGIN
        count := 1; power := 1;
        While count < = expNum Do
        Begin
                power := power * baseNum;
                count := count + 1;
        End;
        Writeln (□The power is : □, power);
END;

BEGIN
        write ('Please enter base: ');
        readln (base);
        write ('Please enter exponent: ');
        readln (exponent);
        DOPOWER(BASE,EXPONENT);
END.
```

Figure 2

The Procedure Header

The general form of a procedure header contains the procedure identifier, then a set of parentheses containing the parameter list. For example, suppose a procedure is to find the volume of a box and output the result. This procedure would require parameters for the height, width, and length of the box, since these values determine the particular box. A header for this procedure might be

PROCEDURE Volume (Nigh, Long, Wide: REAL)

A call to this procedure would require values for each parameter. Thus, a call to this procedure might be

Volume (18.2, 20.12, 5.2)

Notice that the items in the parameter list for the procedure header and in the procedure call are separated by a comma. The identifiers in the procedure header parameter list do not have to be previously declared in the program.

Consider the following procedure header

PROCEDURE Example (Num : INTEGER; ChA, ChB : CHAR)

A call to this procedure must include values for the parameters Num, ChA and ChB. For example,

Example (19, 'A', 'G')

As you can see, a parameter list is stated much like declared variables in a VAR section. In fact, the procedure header just given could be presented as follows

PROCEDURE Example (Num : INTEGER; ChA, ChB : CHAR)

Also notice, in the call to the procedure, that the values sent to the procedure parameters are ordered. That is,

Example ('A', 19, 'Q')

would yield an error, since an attempt would be made to assign to Num a value of type CHAR and to assign to ChA a value of type INTEGER.

9.1.2.2 VARIABLE PARAMETERS

The procedures that have been shown can be classified as procedures without parameters, and procedures with value parameters. The first class of procedures simply defines a set of instructions to be executed. The second class defines a set of instructions to be executed based on a set of values (parameters) sent in the call to the procedure.

Procedures with variable parameters can do the following :

1. Define a set of instructions to be executed
2. Allow for a set of values to be sent when the procedure is called
3. Allow for a set of values to be returned after the instructions have been executed.

Consider the procedure DoPower that shown previously.

The procedure's action can be limited to one thought. DoPower finds the power and writes the result to the screen. The instruction

WRITELN('The power is:', Power)

in the procedure DoPower can be moved to the main execution section. However, the variable Power would have to be declared a program variable. Additionally, a method of transferring the value for Power back to the calling execution section must be devised. This is due to the fact that the program variable Power and the procedure variable Power reference different memory locations. The transfer of the information back to the calling execution section can be accomplished by using a variable parameter.

9.2 PARAMETER PASSING WITH PROCEDURE

A variable parameter is used to return a value to the calling execution section. The return of the value is accomplished by the procedure placing a value in a memory location referenced by a variable from the calling execution section. That is, the variable parameter shares a memory location with a variable from the calling execution section. Thus, a variable parameter must be given a variable identifier when the procedure is called. The instructions defined in the procedure execution section place a value in the location named by the variable identifier. Once the procedure execution is completed, the value in the variable parameter remains. (With value parameter, the values in memory are lost after the procedure execution). The procedure DoPower using a variable parameter to return the results to the calling execution section is shown in figure 3.

```
PROCEDURE dopower (baseNum, expNum : integer; VAR results : real);
VAR
        count : integer;
BEGIN
        count := 1;
        results : = 1;
        While count < = expNum Do
        Begin
                results := results * baseNum;
                count := count + 1;
        End;
END;
```

Figure 3

indicate variable parameters, the word VAR must precede the variable parameter identifiers when they are listed in the procedure header. In a call to a procedure that has variable parameters, variable identifiers must be sent to the variable parameters.

CHAPTER 10

10 INTRODUCTION TO ARRAYS

10.1 USE OF ONE DIMENSIONAL ARRAY

10.1.1 ARRAY DECLARATION

As with any data structure, the RECORD data structure provides a structuring on simpler data types. The structuring provided is that of a record. The simpler data types can be any valid Pascal data type (except TEXT).

10.1.2 USING TYPE STATEMENT

A good analogy of the Pascal data type RECORD is a record like your student record that is in the personnel department. On this record is your name, your parents,' names, courses you are taking, scores you have made on standardized tests, comments made by some of your earlier teachers, and so forth. This collection of data is organized into a form. The pieces of data are of different types. Some of the information is string-type data, some is numerical data. Other everyday examples of record are credit record, military records and social security records.

To define a RECORD type in a TYPE section, the following form is used

```
TYPE
    identifier = RECORD
        feldl identifier : type;
        feld2 identifier : type;
            .
            .
        fieidn identifier : type;
    END;
```

An example of a definition of a RECORD type is as follows:

```
TYPE
    Person =RECORD
        Name: STRING[40];
        Address : STRING[40];
        Age: INTEGER;
        Income: REAL;
    END;
```

The identifiers Name, Address, Age and Income are called fields of the record. With the type, Person, defined, a variable could now be declared and its type stated as Person.

For example,

VAR Customer : Person

declares the identifier Customer to be a variable and states that the values it will hold are of type Person (a record). The following instructions would assign a value to the variable Customer:

Customer.Name := 'John Small'
Customer.Address : = '123 Boxwood Av Helena, Montana';
Customer.Age := 26;
Customer.Income := 32000

A variable of type RECORD is assigned a value by assigning a value to each field of the record. A period is used to specify which field of the record is being accessed. The identifiers separated by a period (such as Customer.Age) are called field selectors.

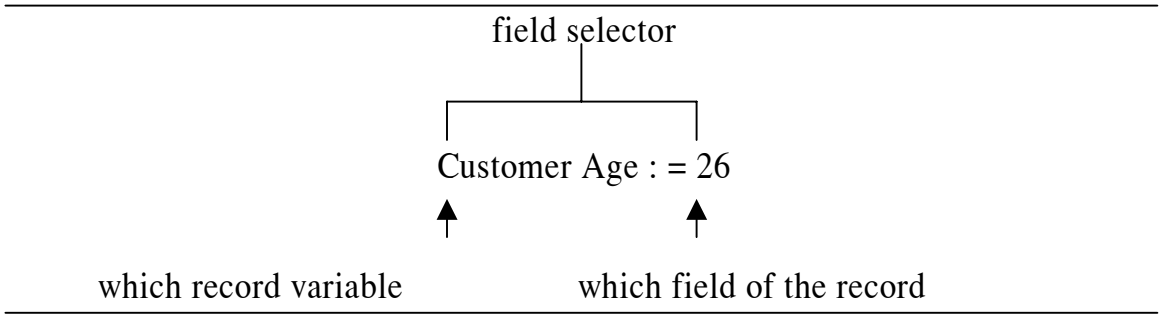


Figure 1

COMMENT

All variables of type person have the field identifiers Name, Address, Age and Income. That is, if Supplier is also of type Person then Supplier.Name := 'Raoul Bliss' is a valid assignment to the Name field of Supplier.

EXAMPLE

Define a type *PartRec,* to be a record with fields *PartName, PartNum, Quantity,* and *Ordered.* Declare Parts to be a variable of type PartRec. Write instructions to assign a value to each field of the variable Parts. Allow the user to verify the assignment. That is, the code should echo the field assignments, then allow the user to re-enter the values if assignments are incorrect.

SOLUTION

```
TYPE Digits = 1000..9999;
     WholeNum = 0..MAXINT;
     PartRec = RECORD
          PartName : STRING[40];
          PartNum : Digits;
          Quantity : WholeNum;
          Ordered : BOOLEAN;
     END;
VAR
     Ans, OK : CHAR;
     Parts  : PartRec;

BEGIN
     REPEAT
          WRITE('Enter part name: ');                    {Fill Record}
          READLN(Parts.PartName);
          WRITE('Enter part number (1000.9999):□;
          READLN(Parts.PartNum);
          WRITE('Enter quantity on hand:□);
          READLN(Parts.Quantity);
          WRITE(% part ordered? (Y/N):□);
          READLN(Ans);

          IF (Ans = 'Y') OR (Ans = 'y')     THEN
                    Parts.Ordered := TRUE
          ELSE Parts.Ordered := FALSE;
          WRITELN;
          WRITELN('The record entered is: ')             {Show Record}
          WRITELN('Name: ', Parts. PartName);
          WRITELN('ID Number: ', Parts.PartNum);
          WRITELN('Quantity: ', Parts.Quantity);
          WRITE('On order: ');

          IF Parts.Ordered THEN
               WRITELN('Yes')
          ELSE WRITELN('No');

          WRITELN;
          WRITE('Is record correct? (Y/N): □);           {Verify Record}
```

```
        READLN(OK);
    UNTIL (OK = 'Y') OR (OK = 'y');
END.
```

10.1.3 USING FOR STATEMENT

Parallel arrays with different component types can be combined into a single data structure. This is accomplished by using the user-defined data structure array of records. Consider this parallel arrays

Index	State	SeatBelt	Deaths
1	FL	true	0.021
2	NY	true	0.008
3	NJ	true	0.005

These three lists can be combined into an array of records as follows

```
TYPE StateRec = RECORD
                    State : STRING[2];
                    SeatBelt : BOOLEAN;
                    Deaths : REAL
                END;

RecList = ARRAY[1..50] OF StateRec;

VAR StateList : RecList
```

The identifier **StateList** has been declared a variable and its type stated as **RecList**. Thus, the values **StateList** will hold are one-dimensional arrays in which the components are of type **StateRec**. Each component of the array will be a record. An assignment to component 3 of **StateList** would be as follows:

```
StateList[3].State := 'NJ';
StateList(3].SeatBelt := TRUE;
StateList[3].Deaths := 0.005;
```

The following instruction could be used to assign a record value to each of the fifty cells of the variable StateList

```
FOR Index := 1 to 50 DO
    BEGIN
        WRITE('Enter state name ', Index,□:□);
        WEADLN(StateList[Index].State);
        WRITE('Does state have seat belt law (Y/N) : □);
        READLN(Ans);
        IF Ans = 'y' OR Ans = 'Y' THEN
            StateList[Index].SentBt : = TRUE
        ELSE StateList[Index].SeatBt := FALSE;
        WRITE('Enter highway death rate : ');
        READLN(StateList [Index].Deaths);
    END;
```

Next, consider this problem involving an array of records and a TEXT file.

Problem

A local bank has a summary of its customer records in a TEXT file. The lines of the TEXT file are similar to the following:

Polset, Thomas M. < end-of-line marker>
546-0145 < end-of-line marker>
-23.56 < end-of-line marker >
Pursey, Lucy J. < end-of-line marker >
567-1234 < end-of-line marker >
378.65 < end-of-line marker >

Each set of three lines contains a name, an account number and a balance. The bank needs a Pascal program that allows the user to select from the menu shown.

Menu

A. Find name
B. Find overdrawn accounts
C. Find sum of all accounts
Q. Quit

Solution

First a data structure is selected. For this problem, a list of records is chosen:

```
Customer = RECORD
        Name: STRING[40];
        AccNum : STRING(8];
        Balance: REAL;
END;

ListType = ARRAY[ I..IndexMax] OF Customer
```

Design :
```
REPEAT
        Present menu
        Get selection
        CASE Select OF
                'A' : Find Name
                'B' : Scan Balance
                'C' : Sum Balance
        END
UNTIL Select = 'Q'
```

Refinement of : ***FillList***

This procedure reads three lines of TEXT file to fill one cell of List

```
Open file
Index . = 0
WHILE NOT EOF(InFile) DO
        Index : = Index + 1
        READLN(InFile, List[Index}.Name)
        READLN(InFile, List[Index).AccNum)
        READLN(InFile, List[Index}.Balance)
Close file
```

Refinement of : ***FfndName***

```
WRITE('Enter name of customer')
WRITE('(Iast, first, middle initial)': ')
READLN(Name)
BinarySearch(List,Length,Name,Found)
IF Found = 0
        THEN WRITELN('Name not found')
        ELSE WRTTELN('Name: ',List[Found].Name)
                WRITE('Account number: ')
                WRITELN(List[Foundl.AccNum)
                WRITE('Balance: $')
                WRITELN(List[Found]. Balance: 1:2)
```

The procedure Binary Search must be altered to deal with a list of records

Refinement of : ***ScanBalance***

```
Index : = 1
```

```
WHILE (Index < = ListLen) DO
IF List[Index].Balance < 0
        THEN WRITELN('Name: ',List[Index].Name)
                WRITE('Aecount number: ')
                WRITELN(List[Index].AccNum)
                WRITE('Balance: $')
                WRITELN(List[Index].Balance:1:2)
Index : = Index + 1
```

Because this procedure and the procedure FindRecord must write the record to the semen, a procedure will be used to write the record to the screen

Refinement of : ***SumBalance***

```
Sum : = 0
FOR Index : = 1 TO ListLen DO
        Sum : = Sum + List[Index].Balance
WRITE('Sum of all accounts is: $', Sum: 1:2)
```

A structured diagram of this design is shown in Figure 2. The design yields the following Pascal program.

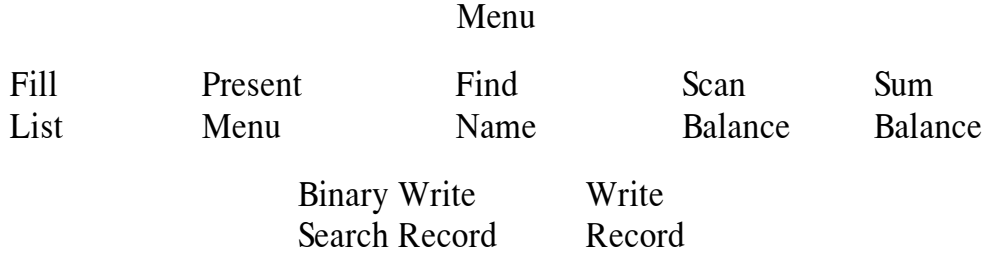


Figure 2

```
PROGRAM heckAccount;

CONST IndexMax = 200;

TYPE Customer = RECORD
                    Name: STRING[40];
                    AccNurn : STRING[S];
                    Balance : REAL;
                END;
     ListType = ARRAY[1..IndexMax] OF Customer;
     WholeNum = O..MAXINT;
     String40 = STRING[40];

VAR  List : ListType;
     Select: CHAR;
```

```
        InFle : TEXT;
        ListLen : WholeNum;

        PROCEDURE FillList (VAR InFile : TEXT, VAR List: ListType;
                                        VAR Index : WholeNum);
BEGIN                                                           {Fi1lList}
        ASSIGN(InMle, 'Accounts.TXT');
        RESET(InFile);
        Index : = 0;
        WHILE NOT EOF(InFile) DO
                BEGIN
                        Index := Index + 1;             {keep up with length of list}
                        READLN(InFle, List[Index}.Name);
                        READLN(InFile, List[Index}.AccNum);
                        READLN(InFle, List[Index}.Balance);
                END;
        CLOSE(InFile);
END;                                                            {FillList}

PROCEDURE PresentMenu;

BEGIN
        WRITELN;
        WRITELN('MENU');
        WRITELN('A. Find name');
        WRITELN('B. Find overdrawn account');
        WRITELN('C. Find sum of all accounts');
        WRITELN('Q. Quit');
        WRITELN;
END                                                     {PresentMenu}

PROCEDURE WriteRecord (TheRecord : Customer);

BEGIN                                                   {WriteRecord}
        WRITELN('Name: ', TheRecord.Name);
        WRITELN('Account number: ', TheRecord.AceNum);
        WRITELN('Balance: $', TheRecord.Balance: 1:2);
END;                                                    {WriteRecord}

PROCEDURE FindName (VAR List : ListType; ListLen : WholeNum);
```

```
VAR Name: STRING[40];
    Found : WholeNum;

PROCEDURE BinarySearch(ListLen : WholeNum; Target : String40;
                       List : ListType; VAR Found : WholeNum);

    VAR High, Low, Mid : INTEGER;

        BEGIN                                              {BinarySearch}
            Found := 0;
            High := ListLen + 1;
            Low . = 0;
            WHILE ((High - Low) > 1) AND (Found = 0) DO
             BEGIN                                          {WHILE-DO}
                Mid : = (High . + Low) DIV 2;
                IF List[Mid].Name = Target
                    THEN Found := Mid
                    ELSE IF Target > List[Mid].Name
                        THEN Low := Mid
                        ELSE High := Mid;
             END;                                           {WHILE-DO}
        END;                                               {BinarySearch}

BEGIN                                                       {FindName}
    WRITE('Enter name of customer: ');
    WRITE('(last, first, middle initial)': ');
    READLN(Name);
    BinarySearch(LIstLen,Name,List,Found);
    IF Found = 0
        THEN WRITELN('Name not found')
        ELSE WriteRecord(List[Found]);
END;    {FndName}

PROCEDURE ScanBalance (VAR List : ListType; ListLen : WholeNum);
 VAR Index : WholeNum;
 BEGIN                                                     {ScanBalance}
    Index := 1;
    WHILE (Index < = ListLen) DO
     BEGIN                                                 {WHILE-DO}
        IF List[Index].Balance < 0
            THEN WriteRecord(List[Index]);
```

```
            Index := Index + 1;
        END;                                                    {WHILE-DO}
  END                                                           {ScanBalance}

PROCEDURE SumBalance (VAR List : ListType; ListLen : WholeNum);
  VAR Sum : REAL;
      Index : WholeNum;
  BEGIN                                                         {SumBalance}
      Sum : = 0;
      FOR Index := 1 TO ListLen DO
            Sum := Sum + List[Index].Balance;
      WRITELN('Sum of all accounts is: $', Sum:1:2);
  END;                                                          {SumBalance}

BEGIN                                                           {Main Program}
      FlllList(Infile, List, ListLen);
      REPEAT
            PresentMenu;
            WRITE('Enter A, B, C, Q: ');
            READLN(Select);
            CASE Select OF
                  'A' : FindName(List, ListLen);
                  'B' : ScanBalance(List,ListLen);
                  'C' : SumBdlance(List, ListLen);
            END;
      UNTIL Select = 'Q';
END.                                                            {Main Program}
```

CHAPTER 11

11 SEARCHING

11.1 TYPES OF SEARCHING

A search, or a "table look-up" depend on the number and size of entries in the table, and whether or not the data stored is fixed or can vary during the program run (as in the case of a list of variable names generated during compilation of a source program).

11.1.1 LINEAR SEARCH

The simplest and most obvious approach to use a *linear search;* here the program looks at each table element in turn, starting with the first, until either it finds a match or it reaches the end of the table (in which case the key element was "missing", in other words, not in the table). It is usual for all elements of the table to have the same fixed size, so it is relatively simple to describe a standard technique for the linear table search.

To make our table search more elegant and easier to amend, we could rewrite it using two logical items "searching" and "found". The conditional expression "if found" means "if the value of logical item found is true". Similarly "while searching" means "while searching has the value true". Now our process looks like:

```
index ← 1
searching ← true;
while searching
 do
  if key = table [index]
    then
     searching ← false;
     found ← true;
  else if index = tabsize
     then
       searching ← false;
       found ←false;
     else
       index ← index + 1;
  endif;
```

```
 enddo;
if found
   then process record in Table [index];
   else process missing KEY value;
endif;
```

PROBLEM

Write a program to determine the title of book is found in a book listing.

```
Program Search;
uses crt;
var
        title: array [1..5] of string;
        a :integer;
        request_title : string;

begin
        clrscr;
        for a := 1 to 5 do
        begin
                write ('Title □,a,□);
                readin (title [a]);
        end;

        writeln ('Enter the title: □);
        readin (request_title);

        for a : = 1 to 5 do
                if request_title = title [a] then
                        writeln ('This is found in the list.');

End.
```

11.1.2 BINARY SEARCH

A binary search or logarithmic search compares the item in the middle of the table with the search key. If they match, then the required entry has been found. If not, the program tests on which side of the target (search key) the examined entry lies. If the table entry is lower in the sequence than the search key, then (because of the ordering of the table) so are all of the proceding entries.

On the other hand, if the examined table entry is higher in the sequence, so are all of the following entries. In either case, we have got rid of half of the table. We can now treat the remaining half table in the same way; half of it ($^1/4$ of the original table) is eliminated. This goes on until either a match is found or, by reducing the area of search to a single entry, the program finds that the required value is not in the table.

Since at each step, we reject 1h of the search area, a table of N items will require, at most, 2 x loge[sub]N comparisons to find a given key or that it is missing. This give the method its name. The algorithm (pseudocode program) shown below is to search a table of length tab-size for a value which matches key.

```
first ←;
last ← tabsize;
while first < > last
 do
  interval ← last - first + 1;
  half ← interval /2;
  pointer ← first + half;
  if key = table [pointer]
    then goto found;
  endif;
  (since no match, determine which half or table to keep)
  if key > table [pointer]
    then first ← pointer;
    else last ← pointer;
  endif;
 enddo;
 missing: (control comes here if key not in table)
 found: (process matching element)
```

Do not forget, thought, that we can use the logarithmic search only with an *ordered table of values*.

For example:

Refinement of : *FindName*

```
WRITE('Enter name of customer')
WRITE('(last, first, middle initial)': ')
READLN(Name)
BinarySearch(List,Length,Name,Found)
IF Found = 0
```

The procedure Binary search must be

```
        THEN WRITELN('Name not found')
        ELSE WRITELN('Name: ',List[Found].Name)
             WRITE('Account number: ')
             WRITELN(List[Found].AccNum)
             WRITE('Balance: $')
             WRITELN(List[Found].Balance:1:2)
```

altered to deal with a list or records

solution

```
PROCEDURE FindName (VAR List: ListType; ListLen : WholeNum);

 VAR Name: STRING[40];
     Found : WholeNum;

 PROCEDURE BinarySearch(ListLen : WholeNum; Target : String40;
                        List : ListType; VAR Found : WholeNum);

     VAR High, Low, Mid: INTEGER;

         BEGIN                                          {BinarySearch)
             Found := 0;
             High := ListLen + 1;
             Low := 0;
             WHILE ((High - Low) > 1) AND (Found = 0) DO
              BEGIN                                     {WHILE-DO}
                  Mid := (High + Low) DIV 2;
                  IF List[Mid].Name = Target
                       THEN Found := Mid
                       ELSE IF Target > List[Mid].Name
                            THEN Low := Mid
                            ELSE High : = Mid;
              END;                                      {WHILE-DO}
         END;                                           {BinarySearch}
```

For more details, please refer to Chapter 10 figure 2.

CHAPTER 12

12 BUBBLE SORT

12.1 BUBBLE SORT

Sorting is the process of arranging items in a desired sequence. An internal sort involves re-arranging the items of a table so that the values of an aprticular key are in order (either ascending or descending). Where we have to work with large volumes of data, such as massive files, we need an "external" sort.

12.1.1 AN INTERNAL SORT

Consider a linear array A having N elements, each holding an integer value. For example, if N is 10, the array looks like:

A	11	8	34	78	-5	8

After re-arranging, the contents of the array is in ascending order, as shown below:

A	-5	8	8	11	34	78

We will inspect the full list, looking for the lowest value. Having found it, we exchange that value with the value at the beginning of the list. Now we can ignore the first item in the list, and repeat the operation with the shorter list starting at the second item. Having found the lowest item in that list, we exchange it with the value in position two (the head of the shorter list). We repeat this until we have only two items left, exchange them if necessary - and the sort is complete.

Bubble Sort compares pairs of values (first with second, then second with third, and so on), and the elements of the pairs are exchanged if they are in the wrong order. This way, the smallest value moves up one position at each iteration through the table - and after one iteration, the largest value is at the end.

PROBLEM

Write a program to accept five book titles and sort them in ascending order.

```
Program Bubble;
uses crt;
```

```
var
        title : array [1..5] of string;
        a,b :integer;
        t : string;                     {temporary storage when swapping value}

Begin
        clrscr;
        for a := 1 to 5 do
                title [a] := ' ';                                   {initialise values}

        for a : =       1 to 5 do
        begin
                write ('Title of book ', a');
                readln (title [a]);
        end;

        for a := 1 to 4 do
                for b := 1 to (5-a) do
                begin
                        if title [b] > title [b+1] then
                        begin
                                t := title [b];
                                title [b] := title [b+1];
                                title [b+1] := t;
                        end;
                end;

        writeln ('Sorted list');
        writeln ('-----------------------------------');
        for a := 1 to 5 do                          {sorted in ascending order}
                writeln (title [a]);

End.
```

PROBLEM

Write a program to accept ten scores and sort them in descending order.

```
Program Bubble;
uses crt;
const
```

```
        maxno := 10;
var
        score : array [1 .. maxno] of integer;
        a,b :integer;
        t : integer;                    {temporary storage when swapping value}
Begin
        ..............
        for a := 1 to (maxno -1 ) do
                for b := 1 to (maxno-a) do
                begin
                        if score [b] < score [b+1] then
                        begin
                                t := score [b]
                                score [b] : = score [b+ 1];
                                score [b+ 1] : = t;
                        end;
                end;

        writeln ('Sorted list');
        writeln ('----------------------------------------□);
        for a : = 1 to maxno do                 {sorted in descending order}
                writeln (score [a]);

End.
```

CHAPTER 13

13 MORE ABOUT ARRAYS

13.1 USE OF TWO-DIMENSIONAL ARRAYS

There are many problems in which the data being processed can be naturally organised as a table. For example, suppose that water temperatures are recorded four times a data at each of three locations near the discharge outlet of the cooling system of a nuclear power plant. These temperature readings can be arranged in a table having four rows and three columns

Time	Location		
	1	2	3
1	68.5	68.7	62.0
2	68.8	68.9	64.5
3	70.4	69.4	66.3
4	68.5	69..1	65.8

In this table, the three temperature readings at time I are in the first row, the three temperatures at time 2 are in the second row, and so on.

13.1.1 ARRAY TYPE DEFINITION

These 12 data items can be conveniently stored in a *two-dimensional array*. The array declaration:

```
CONST
     MaxTimes = 4;
     MaxLocations = 3;

TYPE
     TemperatureTable = array[1..MaxTimes, 1..MaxLocations] of real;

VAR
     TempTab : TemperatureTable;
```

reserves 12 memory locations for these data items. The doubly indexed variable

TempTab[2,3]

then refers to the entry in the second row and third column of the table, that is, to the temperature 64.5 recorded at time 2 at location 3.

In general, TempTab[i,j] refers to the entry in the *i*th row and *j*th column, that is, to the temperature recorded at time *i* at location *j*.

An equivalent method of declaring a multidimensional array is as an ***arrays of arrays***, that is, an array whose elements are other arrays. To illustrate, reconsider the table of temperatures given earlier.

For each time there is an array of temperatures corresponding to one row of the table. The entire table might then be viewed as an array of these temperature arrays. Thus, if we define the types *Temperatures* and *Array0fTemperatures* by

```
TYPE
    Temperatures = array[1..MaxLocations] of real;
    ArrayOt Temperatures = array[1..MaxTimes] of Temperatures;
    or

TYPE
    Array0fTemperatures =
        array[1..MaxTimes) of array[L.MaxLocations) of real;
```

then the declaration

```
VAR
    Table : Array0fTemperatures;
```

established 12 memory locations of data items.

The single indexed variable Table[2] refers to the second row of the table, that is, to the list of temperatures at time 2 : **68.8 68.9 64.5**

The doubly indexed variable **Table[2)[3]** or equivalently, **Table[2,3]** then refers to the third temperature in this row, that is, the temperature **64.5** recorded at time 2 at location 3.

In general, either Table[i,j] or Table[i][j] refers to the jth entry in the ith row of the table, that is, to the temperature recorded at time i at jth location.

13.1.2 USING FOR LOOPS WITH ARRAYS

The elements of a multidimensional array can be read by using nested repetition structures; we simply place an input statement within a pair of nested *for* (or *while* or *repeat*) loops, each of whose control variables controls one of the indices

of the array. For example, to read the twelve entries of a 3 by 4 integer array *StudArray* in rowwise order so that it has the value

2	37	0	0
0	1	11	32
6	18	4	12

We could use the nested for loop statements

```
for row := 1 to 3 do
        for column : = 1 to 4 do
                read (StudArray[row,column]);
```

Thus, to calculate the average temperature at each of the four times, it is necessary to calculate the sum of the entries in each row and to divide each of these sums by 3. Example :

```
for Time := 1 to 4 do begin
        Sum := 0;
        for Loc : = 1 to 3 do
                Sum := Sum + TempTab[Time,Loc];
        Meantime := Sum/3;
        writeln          ('Mean Temperature at Time ',Time:1, ' is ',
MeanTime:3:1);
end; (* for *)
```

Nested repetition structures can also be used to copy the entries of one array into another array. For example, if ArrayA and ArrayB are 5 by 10 arrays, the nested for statements

```
for row : = 1 to 5 do
        for column := 1 to 10 do
                ArrayB[row,column] := ArrayA[row,column];
```

will copy the entries of ArrayA into the corresponding locations of ArrayB, assuming, of course, that the component type of ArrayA is compatible with the component type of ArrayB. If ArrayA and ArrayB have the *same types,* this can be done more simply with the array assignments statement ArrayB : = ArrayA;

CHAPTER 14

14 RECORDS

14.1 DECLARATIONS

As with most data structures in Pascal, the data structure RECORD can be built upon. That is, the fields of the record can be declared types like ARRAY or RECORD. Just as the instructions of a program can be refined by using procedures and functions, the fields of a record can be refined. Records which contain a field of type RECORD are called hierarchical records, nested record or tree structures. Consider the record type given below

```
TYPE
    Person = RECORD
        Name: STRING[40];
        Address : STRING[40];
        Age: INTEGER;
        Income: REAL;
    END
```

The field name could be refined as

```
NameRec = RECORD
    First : STRING[18];
    MidInit : STRING[2];
    Last : STRING[20];
END
```

Likewise, the field Address could be refined as

```
AddRec = RECORD
    Street : STRING[40];
    City: STRING[20];
    State : STRING[20];
END
```

Declaring variable Customer to be of type Person, a value would now be assigned to Customer as follows :

```
Customer.Name.First := 'John';
Customer.Name.Midlnit := 'G';
Customer. Name.Last := 'Small';
```

```
Customer.Address.Street := '125 Boxwood Av. ';
Customer.Address.City := 'Helena';
Customer.Address.State := 'Montana';
Customer.Age := 26;
Customer.Income := 31000;
```

Now the TYPE section would appear as shown in figure 1.

```
TYPE
    nameRec = RECORD
        first: string [18];
        midlnit : string [2];
        last: string [20];
    END;

    AddRec = RECORD
        Street : STRING[40];
        City : STRING[20];
        State : STRING[20];
    END;

    Person = RECORD
        Name: STRING[40];
        Address : STRING[40];
        Age: INTEGER;
        Income: REAL;
    END
```

Figure 1

14.2 ARRAY OF RECORDS

As you can see, to access the inner fields a path is followed. The field selector, Customer. Name. Last, accesses the record Customer, then the field Name, then the field Last. Assignments (with the assignment operator : =, READ, etc) with RECORD type data are done by fields. Thus, for variables Debtor and Customer of type Person, the following instructions are valid assignments

```
Debtor : = Customer;
Debtor.Name : = Customer.Name;
Debtor.Name.First := Customer.Name.First;
```

That is, two records of the same type and/or two fields of the same type can be used with the assignment operation. Also, for Debtor of type Person and Owner of type Name, the following are valid assignment :

```
Debtor.Name := Owner;
Debtor. Name.Last := Owner.Last;
```

14.3 USING WITH STATEMENT

The Pascal instruction WITH can be used to reduce the amount written to access a field of a record. For example, suppose you wanted to write to the screen the high and low temperature for the record value in the variable Today. Using the WITH instruction, this could be accomplished as follows:

```
WITH Today.Weather.Thermo DO
    BEGIN
        WRITELN('High temperature was : □, HiTemp);
        WRITELN('Low temperature was : □,      LoTemp);
        END
```

The record path (Today.Weather.Thermo) is stated once in the WITH instruction. Throughout the block bracketed by the BEGIN-END, only the field selector is sued. This WITH instruction replaces the instructions

```
WRITELN('High temperature was : ',Today.Weather.Thermo.HiTemp);
WRITELN('Los temperature was : ',Today.Weather.Thermo.LoTemp);
```

The general form of the WITH instruction is

WITH *record identifier* **DO**
    < instruction >

The *record identifier* can be the root of the record or part of the path up to the leaf of the record. In the example just given, Today.Weather.Thermo was a path of the record.

For a nested record such as

```
TYPE NameRec = RECORD
        First : STRING[18];
        MidInit : STRING[2];
        Last : STRING[20];
    END;
```

```
AddRec = RECORD
        Street : STRING[40];
        City : STRING[20];
        State : STRING[20];
END;

Person = RECORD
        Name: NameRec;
        Address : AddRec;
        Age: INTEGER;
        Income: REAL;
END;

VAR Customer : Person;
```

two possible forms of the WITH instruction are as follows

```
WITH Customer DO          and    WITH Customer.Address DO
    < instruction >                    < instruction >
```

The instruction part of the WITH instruction can be a simple or compound instruction.

For a compound instruction, BEGIN-END is used to mark the scope of the WITH instruction. Any reference to an identifier in the BEGIN-END block of the WITH instruction that matches a field of the record is a reference to the field selector. That is, suppose First is a program variable which is also used as a field selector for the RECORD-type variable Customer (this is not a recommended practice). In the instructions

```
First := 'Mary';
WITH Customer.Name DO
        BEGIN
                .
                First := 'John';       {assignment                to
Customer.Name.First}                    .
                                END;
                        WRITE(First);
{outputs Mary}
```

the field selector Customer. Name. First is assigned the value 'John'. However, the instruction WRITE(First) will output Mary. Outside the BEGIN-END block of the WITH instruction, the instruction

W RITE(Customer.Name.First)

must be used to output John. Inside the BEGIN-END block of the WITH instruction, the instruction WRITE(First) is all that is needed to output the value in the First field of Customer.Name.

CHAPTER 15

15 STANDARD LIBRARY FUNCTIONS (I)

15.1 TRIGONOMETRIC FUNCTIONS

Pascal provides a number of functions to help you develop programs. A function has a name and usually requires you to specify one or more arguments or parameters. The name of the function can be used in a mathematical expression or a write statement in the same way you use the name of a variable, a constant, or a numeric literal. The input to the functions are provided through arguments (these are the values that are placed in parentheses and separated by commas immediately after the function.) The function considers the inputs and then returns a single value. The value returned has a particular type. Some functions only return integers, some only return reals. Others can return either integer or real results.

15.1.1 ABS

ABS(x) returns absolute value of x. X is an integer-type or real-type expression. The result, of the same type as X, is the absolute value of X.

Example:

```
var
  r: Real;
  i: Integer;
begin
  r : = Abs(-2.3);       { 2.3 }
  i : = Abs(-157);       { 157 }
end.
```

15.1.2 EXP

EXP(x) computes e^x, where e = 2.1782818... The value e raised to the power of X, where e is the base of the natural logarithms. X is either a real or an integer value. The result type of X is real.

Example:

```
Begin
        Writeln('e = ',Exp(1.0));
end.
```

15.1.3 RANDOMIZE

Randomize initializes the built-in random generator with a random value and returns a real number greater than zero and less than one. The random value is obtained from the system clock. If Range is not specified, the result is a Real-type random number within the range 0 < = X < 1. If Range is specified, it must be an expression of type Word, and the result is a Word-type random number within the range 0 < = X < Range. If Range equals 0, a value of 0 is returned. The random number generator should be initialized by making a call to Randomize, or by assigning a value to RandSeed.

Example:

```
uses Crt;

 begin
       Randomize;
       repeat
             { Write random numbers }
             Writeln (Random(1000));
 until KeyPressed;
end.
```

15.1.4 RANDOM

RANDOM(X) returns a word less than x. If x is not specified, the result is a real random number within the range 0 -s x s 1. If x(range) is specified, it must be an expression of type integer, and the result is a word random number within the range 0 s x < range. If x is s 0, a value of 0 will be returned. The value of X must be a word (0..65,535).

Example:

```
uses Crt;
begin
 Randomize;
 repeat
   { Write random numbers }
   Writeln (Random(2000));
 until KeyPressed;
end.
```

15.1.5 LN

IN(X) computes the natural logarithm of x (x0). X is integer or real type. The result type of X is real.

Example:

```
var
  e . real;
begin
  e := Exp(1.0);
  Writeln('ln(e) = ',In(e));
end.
```

15.1.6 PI

PI returns the value of Pi, which is defined as 3.1415926535897932385. Precision varies, depending on whether the compiler is in 80x87 or software-only mode.

Example:

```
Begin
        Writeln('π = ',Pi);
end.
```

15.1.7 INT

INT(X) returns the integer part of X. X is a real-type expression. The result is the integer part of X; that is, X rounded toward zero.

Example:

```
Var R: Real;
begin
  R : = Int(123.456);                { 123.0 }
  R := Int(-123.456);          { -123.0 }
end.
```

15.2 NUMERIC FUNCTIONS

The predeclared numeric functions provide a method of performing many numeric operations without the tedium of deriving formulas and procedures yourself. These functions include not only the basic numeric functions but include also the trigonometric functions that provide the basis for construction of all trigonometric functions as mentioned in 15.1.

15.2.1 ROUND

ROUND(X) rounds the value of X to the nearest integer. X is a real-type expression. Round returns a Longint value that is the value of X rounded to the

nearest whole number. If X is exactly halfway between two whole numbers, the result is the number with the greatest absolute magnitude. A run-time error occurs if the rounded value of X is not within the Longint range.

Example:

```
begin
  Writeln(1.4, ' rounds to ', Round(1.4));
  Writeln(1.5, ' rounds to ', Round(1.5));
  Writeln(-1.4, ' rounds to ', Round(-1.4));
  Writeln(-1.5, 'rounds to ', Round(-1.5));
end.
```

15.2.2 SIN

SIN(X) returns the trigonometric sine of X. X is integer or real type. The result type is real. X is a real-type expression. Returns the sine of the angle X in radians.

Example:

```
Var
  R: Real;
Begin
  R : = Sin(Pi);
  Writeln ('The Sine of r is', R);
  Readln;
end.
```

15.2.3 SQR

SQR(X) computes the square of X. X is an integer-type or real-type expression. The result, of the same type as X, is the square of X, or X*X.

Example :

```
begin
  Writeln('5^2 is ', Sqr(5));
end.
```

15.2.4 SQRT

SQRT(X) returns the square root of X. X is a real-type expression. The result is the square root of X.

Example:

```
Begin
  Writeln('The square root of 2 is ', Sqrt(2.0));
end.
```

15.2.5 TRUNC

TRUNC(X) truncates X (ie. ignores the decimal part of X). X is a real-type expression. Trunc returns a Longint value that is the value of X rounded toward zero. A run-time error occurs if the truncated value of X is not within the Longint range.

Example:

```
begin
   Writeln(1.4, ' becomes ', Trunc(1.4));
   Writeln(1.5, ' becomes ', Trunc(1.5));
   Writeln(-1.4, ' becomes ', Trunc(-1.4));
   Writeln(-1.5, ' becomes ', Trunc(-1.5));
end.
```

15.3 CHARACTER FUNCTIONS

Pascal provides several functions for working with character and other scalar data. The ord and chr functions are mutual inverses. The upCase function converts a lower-case character to its uppercase equivalent.

15.3.1 ORD

ORD(C), where C is a char or other scalar value. If C is a character, the function returns an integer value that corresponds to the position of C in the ASCII character set. If C is a scalar, such as a user-defined enumerated type, the ordinal value (decimal integer that is used to encode the character C) of the scalar is returned. C is an ordinal-type expression. The result is of type Longint and its value is the ordinality of C.

Example:

```
type
  Colors = (RED,BLUE,GREEN);
begin
  Writeln('BLUE has an ordinal value of ', Ord(BLUE));
  Writeln('The ASCII code for "c" is ', Ord('c'), ' decimal');
end.
```

15.3.2 CHR

CHR(L), where L is of an integer value. The function returns a char value equal to the position of L within the character set. Chr returns the character with the ordinal value (ASCII value) of the byte-type expression, L.

Example:

```
var
  I: Integer;
begin
  for I : = 32 to 126 do
          Write(Chr(I));
end.
```

15.3.3 UPCASE

UPCASE(SOURCE), where SOURCE is a character. The function returns a character in which each character of SOURCE that is a lower-case character is converted to the corresponding uppercase character, and all other characters are unaffected.

SOURCE is an expression of type Char. The result of type Char is Ch converted to uppercase. Character values not in the range a..z are unaffected.

Example:

```
var
  source : string;
  i : Integer;
begin
  Write('Enter a string: ');
  Readln(source);
  for i : = 1 to Length(source) do
    source[i] : = UpCase(source[i]);
  Writeln('Here it is in all uppercase: ',source);
end.
```

CHAPTER 16

16 STANDARD LIBRARY FUNCTIONS (11)

16.1 STRING FUNCTIONS

Pascal provides several functions for use in manipulating text. The functions are designed for use with string data, but simple assignment allows character data to become string data.

16.1.1 CONCAT

Any number of strings S1, S2,... specifiied when calling **CONCAT(S1,S2,.)** returns a concatenated string of the sequence of strings S1,S2;....Sn. Each parameter is a string-type expression. The result is the concatenation of all the string parameters. If the resulting string is longer than 255 characters, it is truncated after the 255th character. Using the plus (+) operator returns the same result as using the Concat function: S : = 'ABC' + 'DEF';

Example:

```
var
  S: String;
begin
  S : = Concat('ABC', 'DEF'); { 'ABCDE' }
end.
```

16.1.2 LENGTH

L'ENGTH(SOURCE) returns the number of characters in the string, SOURCE.

Example:

```
var
  S: String;
begin
  Readln (S);
  Writeln('"', S, '"');
  Writeln('length = ', Length(S));
end.
```

16.2 OTHER FUNCTIONS

16.2.1 CLREOL

CLREOL clears all characters from the cursor position to the end of the line without moving the cursor. All character positions are set to blanks with the currently defined text attributes. Thus, if TextBackground is not black, the current cursor position to the right edge becomes the background color. **CLREOL** is window-relative. The following program lines define a text window and clear the current line from the cursor position (1, 1) to the right edge of the active window (60, 1).

```
Window(1, 1, 60, 20);
CIrEol;
```

Example:

```
uses Crt;
begin
  CIrScr;
  Writeln('Hello there, how are you today?');
  Writeln('Press <enter> key...');
  Readln;
  GotoXY(1,2);
  CIrEol;
  Writeln ('Glad to hear it!');
end.
```

16.2.2 CLRSCR

CLRSCR clears the active windows and returns the cursor to the upper-left corner. Sets all character positions to blanks with the currently defined text attributes. Thus, if TextBackground is not black, the entire screen becomes the background color. This also applies to characters cleared by C1rEol, InsLine, and DelLine, and to empty lines created by scrolling. **CLRSCR** is window-relative. The following program lines define a text window and clear a 60X20 rectangle beginning at (1, 1).

```
Window(1, 1, 60, 20);
ClrScr;
```

Example:

```
uses Crt;
begin
  Writeln('Hello William Royl. Please the <enter> key...');
  Readln;
  ClrScr;
```

```
end.
```

16.2.3 COS

COS(X) returns the trigonometric cosine of X. X is a real-type expression. The result is the cosine of X where X represents an angle in radians.

Example:

```
var R: Real;
begin
  R := Cos(Pi);
end.
```

16.2.4 DATETIME

Variables of **DATETIME** type are used in connection with the UnpackTime and PackTime procedures to examine and construct 4-byte, packed date-and-time values for the GetFTime, SetFTime, FindFirst, and FindNext procedures.

Example:

```
Type
        { Date & time recored used by PackTime }
        { and UnpackTime }
 DateTime = record
        Year, Month, Day, Hour, Min,Sec: Word;
 end;
```

16.2.5 DELAY

DELAYS a specified number of milliseconds. It is an approximation, so the delay period will not last exactly Ms milliseconds.

Example:

```
uses Crt,

begin
   Sound(220);         { Beep }
   Writeln ('Moorthy is waiting for Joyce at the gate!!!');
   Delay(200);         { For 200 ms }
   NoSound;            { Reliefl }
end.
```

16.2.6 DEC

X is an ordinal-type variable or a variable of type PChar if the extended syntax is enabled, and N is an integer-type expression. X is decremented by 1, or by N if N is specified; that is, **DEC(X)** corresponds to X : = X - 1, and **DEC(X, N)** corresponds to **X : = X - N. DEC** generates optimized code and is especially useful in a tight loop.

Example:

```
var
  Int Var: Integer;
  LongintVar: Longint;
begin
  Intvar := 10;
  LongintVar := 10;
  Dec(IntVar);        { IntVar := IntVar - 1 }
  Dec(LongintVar, 5);      { LongintVar := LongintVar - 5 }
end.
```

16.2.7 INC

INC(X,N) increments a variable, where X is an ordinal type variable, and N is an integer type expression. X is incremented by 1, or by N if N is specified; that is, INC(X) corresponds to X : = X+1, and INC(X,N) corresponds to X := X+N. **INC** generates optimized code and is especially useful in tight loops.

Example:

```
var
  Int Var: Integer;
  LongintVar: Longint;
begin
  Intvar := 10;
  LongintVar := 10;
  Inc(IntVar);        { IntVar := IntVar + 1 }
  Inc(LongintVar, 5);      { LongintVar := LongintVar + 5 }
end.
```

16.2.8 PRED

PRED(X) returns the predecessor of X. The value of X can be integer, char or boolean and the result of the predecessor returned will be same. X is an ordinal-type expression. The result, of the same type as X, is the predecessor of X.

Example:

```
Begin
   Writeln('The predecessor of 5 is ', Pred(5));
end.
```

16.2.9 SUCC

SUCC(X) returns the successor of X. The value of X can be integer, char or boolean and the result of the successor returned will be same. X is an ordinal-type expression. The result, of the same type as X, is the successor of X.

Example:

```
type
  Colors = (RED,BLUE,GREEN);
begin
  Writeln('The predecessor of 5 is ', Pred(5));
  Writeln('The successor of 10 is ', Succ(10));
  if Succ(RED) = BLUE then
    Writeln('In the type Colors, RED is ',
                'the predecessor of BLUE.');
end.
```

16.2.10 VAL

VAL(SOURCE) converts a string value, SOURCE, to its numeric representation, as if it were read from a text file with Read.

```
procedure Val(S; var V; var Code: Integer); where:
 S      string-type variable; must be a sequence of characters that
        form a signed whole number
 V      integer-type or real-type variable
        Code variable of type Integer.
```

Example:

```
var , Code: Integer;
begin
{ Get text from command line }
  Val(ParamStr(l), 1, Code);
  { Error during conversion to integer? }
  if code < > 0 then
     Writeln('Error at position: ', Code)
  else
     Writeln('Value = ', I);
```

```
  Readln;
end.
```

16.2.11 ODD

ODD(X) determines if X(integer) is odd or even. It will return a value of true if X is odd, false if otherwise.
Example:

```
begin
  if Odd(5) then
    Writeln('5 is odd')
  else
    Writeln('Something is odd');
end.
```

16.2.12 KEYPRESSED

KEYPRESSED determines if a key has been pressed on the keyboard. The key can be read using the ReadKey function. True If key has been pressed False If key has not been pressed

Example:

```
uses Crt;
begin
 repeat
   Write('Type A or a');
 until KeyPressed;
end.
```

16.2.13 WINDOW

WINDOW(XI, Y1, X2, Y2) defines a text window on the screen. X 1 and Y 1 are the coordinates of the upper left corner of the window, and X2 and Y2 are the coordinates of the lower right corner. The upper left corner of the screen corresponds to (1, 1). The minimum size of a text window is one column by one line. The default window is (1, 1, 80, 25) in 25-line mode, and (1, 1, 80, 43) itr 43-line mode, corresponding to the entire screen. All screen coordinates (except the window coordinates themselves) are relative to the current window. For instance, GotoXY(1, 1) will always position the cursor in the upper left corner of the current window.

Many Crt procedures and functions are window-relative, including C1rEo1, C1rScr, DelLine, GotoXY, InsLine, WhereX, WhereY, Read, Readln, Write, Writeln.

Example:

```
uses Crt,
var
  X, Y: Byte;
begin
  TextBackground(Black);  { Clear screen }
  ClrScr;
  repeat
    X : = Succ(Random(80));        { Draw random windows }
    Y : = Succ(Random(25));
    Window(X, Y, X + Random(10), Y + Random(8));
    TextBackground(Random(16));          { In random colors }
    CIrScr;
  until KeyPressed;
end.
```

16.2.14 TEXTCOLOR

TEXTCOLOR(COLOR) selects the foreground character color, COLOR. COLOR is an integer expression in the range 0..15, corresponding to one of the text color constants defined in Crt. There is a byte-type variable Crt--TextAttr--that is used to hold the current video attribute. TextColor sets bits 0-3 to Color. If Color is greater than 15, the blink bit (bit 7) is also set; otherwise, it is cleared.

You can make characters blink by adding 128 to the color value. The Blink constant is defined for that purpose; in fact, for compatibility with Turbo Pascal 3.0, any Color value above 15 causes the characters to blink. The foreground of all characters subsequently written will be in the specified color.

Example:

```
uses Crt;
begin
  { Green characters on black }
  TextColor(Green);

  WriteLn('Hey there!');
  TextColor(LightRed+Blink);
  WriteLn('Hi there!');
```

```
  { Yellow characters on blue }
  TextColor(14); { Yellow = 14 }
  WriteLn('Ho there!');
  NormVideo; { Original attribute }
  WriteLn('Back to normal...');
end.
```

16.2.15 TEXTBACKGROUND

TEXTBACKGROIJND(COLOR) selects the background color, COLOR. COLOR is an integer expression in the range 0..7, corresponding to one of the first eight text color constants. There is a byte variable in Crt--TextAttr--that is used to hold the current video attribute. TextBackground sets bits 4-6 of TextAttr to Color. The background of all characters subsequently written will be in the specified color.

Example:

```
uses Crt;
begin
  { Green characters on black }
  TextColor(Green);
  TextBackground(Black);
  WriteLn('Hey Uma!');
  { Blinking light-red characters }
  { on gray }
  TextColor(LightRed+ Blink);
  TextBackground(LightGray);
  WriteLn('Hi Bob Low!');
  { Yellow characters on blue }
  TextColor(14); { Yellow = 14 }
  TextBackground(Blue);
  WriteLn('Ho Linda!');
  NormVideo; { Original attribute }
  WriteLn('Back to normal...');
end.
```

PASCAL PROJECT

SCENARIO

A motor car manufacturer of Royal Motor Company has a network of approved dealers. Each approved dealer has a terminal connected to the manufacturer's central mainframe, and a locally attached printer. Approved dealers can also use their terminals to order cars with catalogue reference for their customers. Each transaction can only be for one car, if a dealer wishes to order several same catalogue reference he must place several orders. Every night a program on the mainframe checks these orders against the manufacturer's inventory system to decide whether the manufacturer has in stock a car that matches the specification. If the car is available from stock, a delivery note is printed at the despatch department, where arrangements are made for the car to be delivered to the dealer. Otherwise, the order is held over until the car is produced. Upon receiving an order from the customer, the dealer will make arrangements to get the car from the manufacturer and apply for registration number from the Registry of Vehicle. When the registration is done, the car will delivered to the customer details into the existing customer record where the car plate number will be regarded as a unique field. Orders are processed on a strictly "first come, first served" basis when allocating cars to customers.

SERVICING

Dealers may also use the manufacturer's mainframe to schedule servicing appointments. There are two types of appointment. The first is requested by the customer, and the second is a routine appointment generated based on the date of sales obtained from the sales file. The routine appointments are generated automatically by computer when a routine service is due. Routine servicing is due six months after a car has been sold, then again after a further six months, and annually thereafter. Routine appointments appear on the master schedule report along with customer-requested appointments. If a customer fails to bring the car in for a routine appointment, the guarantee on the car is automatically void. If an appointment involves any maintenance on the car, the details of this work will be recorded on the paper records of the car's service history (the mechanic involved). If a customer moves out of the area covered by an approved dealer, update the file. Normally the dealer will forward the manual records to the approved dealer nearest to the customer's new address. The appointment requested by the customer are removed from the computer files when the date of the appointment has passed. Routine appointments are kept on the manual file until the guarantee expires, five years after the purchase of the car. Dealers must arrange for appointments whose service cars not bought from this manufacturer, which are considered as new customers.

INPUT REQUIREMENTS

Car plate numbers are to be input to identify the owner. Date of required appointment is to be entered through the terminal. The validity of the appointment dates must be checked (eg. 29/02/1991 is invalid). Probations must be made for error corrections for each input requirement.

PROCESSING REQUIREMENTS

Appointment dates must be after user input current date. Car plate numbers are to be checked for existence before further processing. Upon validation, the name of the owner is to be displayed.

Service codes must also be validated. All correct codes must be accompanied with their respective description for confirmation of service. All services scheduled are to be deleted from the master schedule once the services are done. Only a maximum of 5 services can be allocated for one day. If the total number of services exceeds the allowable services per day, the user is required to *fix* another appointment. Appointment dates must be after user input today's date. The master schedule must be sorted in ascending order by dates prior to generation of the master schedule report.

OUTPUT REQUIREMENTS

Master Schedule Report

Request Date:	Car Plate no	Customer Name	Service Code
DD/MMIYYYY	AAA9999A	XXXXXXXXXXX	XXXXX
DD/MMIYYYY	AAA9999A	XXXXXXXXXXX	XXXXX
DD/MMIYYYY	AAA9999A	XXXXXXXXXXX	XXXXX

****** END OF MASTER SCHEDULE ******

On screen report is to be generated according to the output requirement.

MAJOR ASSIGNMENT 1

Write a program to generate a scheduled service appointment upon customer's request, with the following 5 options in the Main Menu, using *looping:*

MAIN MENU

1 or [A]ppointment entry
2 or [M]aster schedule report
3 or [C]ustomer enquiry
4 or [S]ervice enquiry
5 or [Q]uit

When the first option is selected, a further 3 sub-options are required as follows:

MAIN MENU

1 or E(X)ISTING CUSTOMERS
2 or [N]EW CUSTOMERS
3 or MAIN MEN[U]

MAJOR ASSIGNMENT_2

An extension on Major Assignment l. Write a subroutine or module *each* from the **two menus** using Procedures and Functions.

1) ROUTINE SERVICES

For customers who bought car from the dealer within 5 *years time*. Appointment dates are generated by the SYSTEM.

2) REQUEST SERVICES

For any customers who needs service from the company. Appointment dates are requested by the customers. (Maximum is 5 types of services)

3) TYPES OF SERVICES

The option displays all these service codes and its description:

Code 01 - Car Washing
Code 02 - Car Polishing
Code 03 - Upholstery Cleaning
Code 04 - Window Screen Repairing
Code 05 - Car Audio Repairing
Code 06 - Air Conditioner Repairing
Code 07 - Installation of Alarm Clock
Code 08 - Exhaust System Replacement
Code 09 - Battery and Tyre Servicing
Code 10 - Body Polishing and Repairing
Code 11 - Clutch and Automatic Overhauls
Code 12 - Brake Roller Test and Servicing
Code 13 - Electric Service
Code 14 - Radiator Machine Repairing
Code 15 - Engine Tuning

4) DATE

Checking for *valid date* and leap *year* when user input today's date.

5) SORTING DATA

By using Bubble Sort technique, compare the year, month then day of the request date.

MAJOR ASSIGNMENT 3

An extension on Major Assignment 2. Declare the records of the following into one-dimensional array. (Assuming there are 5 off & each). The actual data for EXISTING CUSTOMERS are stored in the Sales file from COBOL.

1) FIELDS DECLARATIONS

(1) For Existing Customers

Car Plate, Customer Name, Customer Address, Telephone Number,
Purchase Date (DD/MM/YYYY), Warranty Date (DD/MM/YYYY),
Routine Date (DD/MMIYYYY), Request Date (DD/MM/YYYY),
Service Code

(2) For New Customers

Car Plate, Customer Name, Customer Address, Telephone Number,
Request Date (DD/MM/YYYY), Service Code

(3) For Master Schedule Report

Service Date (DD/MM/YYYY), Car Plate, Customer Name, Service Code

(4) For Services

Service Code, Service Descriptions

NB:

1. When entering *service code,* check whether valid *service code or not.* Display the service description, if valid.

2. For Master Schedule Report, if existing customer has no request date, then by default use routine date. As for new customer, use request date.

PASCAL PROJECT
CUS'T'OMER RECORD FROM SALES FILE (COBOL)

RECORD LAYOUT FROM SALES FILE

FIELD NAMES	SIDE	DECIMAL
Order no	5	-
Dealer Code	8	-
Car Catalogue	8	-
Car Model	12	-
Car Price	9	-
Sales/Purchase Date	8	-
Delivery Date	5	-

SAMPLE VALID DATA IN SALES FILE (COBOL)

Order no:	A0011
Dealer:	DC-012-S
Car Catalogue:	CL16AAMP
Car Model:	COROLLA
Car Price:	12.000.00
Sales/Purchase Date:	09/04/1993
Delivery Date:	00001

SAMPLE VALID DATA FOR NEW CUSTOMERS

Car Plate:	SBA 1244A
Customer Name:	MD.SALIM B MAHMOOD
Address 1:	BLK 45, #12-11
Adderss 2:	BRAS BASAH RD
Address 3:	S'PORE 0208
Tel no:	4322100
Request Date:	10/12/1993
Service Code:	2 3 4 9 1

SAMPLE VALID DATA FOR EXISTING CUSTOMERS

Car Plate:	SBC 4388A
Customer Name:	TAN JEREMIAH
Address 1:	BLK 23, #09-11
Adderss 2:	VICTORIA ST

Address 3:	S'PORE 0718
Tel no:	2278234
Routine date:	*Generated by computer*
Request Date:	13/12/1993
Service Code:	8 2 15

PASCAL PROJECT
SYSTEM DOCUMENTATION
CONTENTS